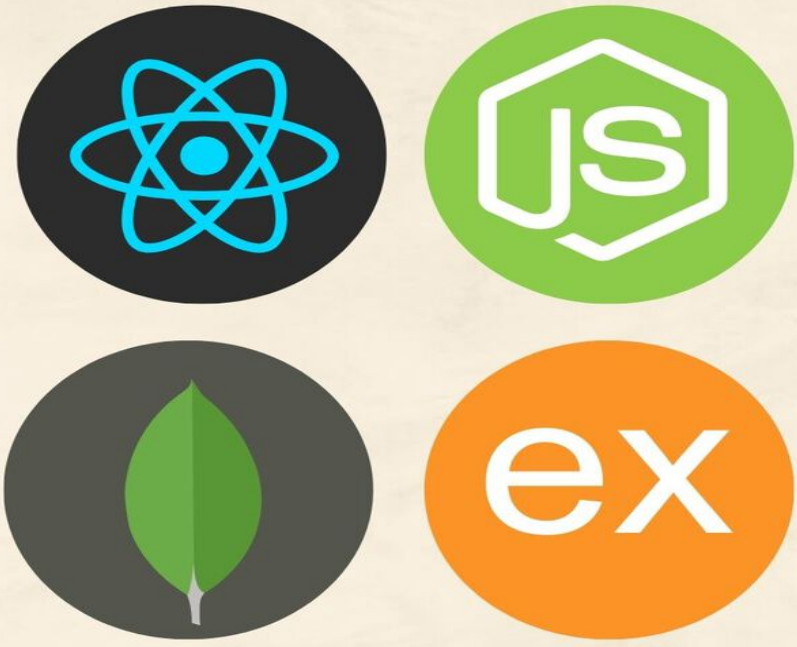
MERN

WEB DEVELOPMENT FOR BEGINNERS

A Step-By-Step Guide to Build a Full Stack Web Application With React, Express, Node.js, and MongoDB



NATHAN SEBHASTIAN

TABLE OF CONTENTS

MERN Stack Web Development For Beginners

A Step-By-Step Guide to Build a Full Stack Web Application With React, Express, Node.js, and MongoDB

By Nathan Sebhastian

PREFACE

The goal of this book is to provide gentle step-by-step instructions that will help you see how to develop web applications using the MERN Stack.

I'll teach you why MERN is a great choice to build full-stack web applications. We'll cover essential MERN topics like routing, authentication, network requests and database query and see how they are used to develop a feature-rich web application.

After finishing this book, you will know how to build and deploy a modern, high-performance web application using MERN.

Working Through This Book

This book is broken down into 18 concise chapters, each focusing on a specific aspect of MERN stack development. You're also going to practice what you've learned in each chapter by developing a Task Management application.

I encourage you to write the code you see in this book and run them so that you have a sense of what web development with

MERN looks like. You learn best when you code along with examples in this book.

A tip to make the most of this book: Take at least a 10-minute break after finishing a chapter, so that you can regain your energy and focus.

Also, don't despair if some concept is hard to understand. Learning anything new is hard for the first time, especially something technical like programming. The most important thing is to keep going.

Requirements

To experience the full benefit of this book, basic knowledge of JavaScript and React is required.

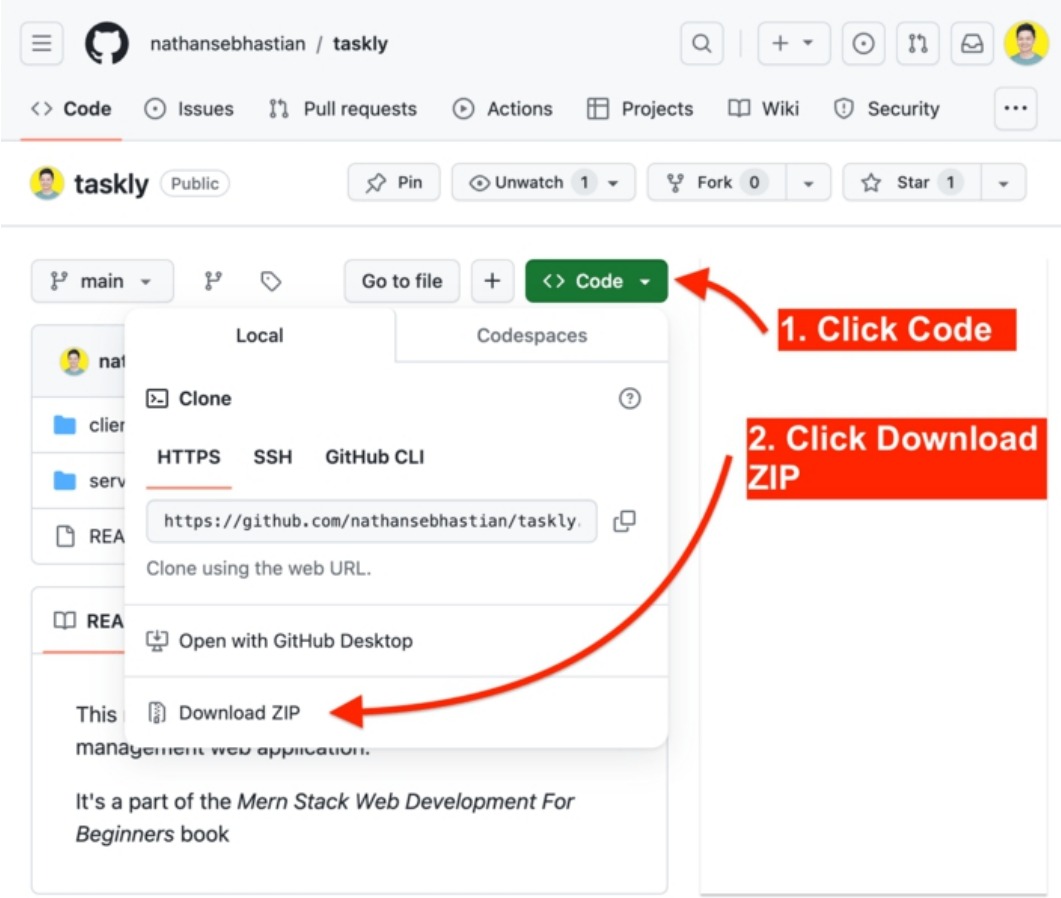
If you need some help in learning React, you can get my book at https://codewithnathan.com/beginning-react

Source Code

This book will help you learn the MERN stack by building a web application project.

In the Summary section of each chapter, you will see a link to download the code from GitHub.

You can download the source code as a ZIP archive as shown below:



This way, you can continue to the next chapter without getting left behind.

Contact

If you need help, you can contact me at nathan@codewithnathan.com.

You might also want to subscribe to my 7-day free email course called Mindset of the Successful Software Developer at https://g.codewithnathan.com/mindset

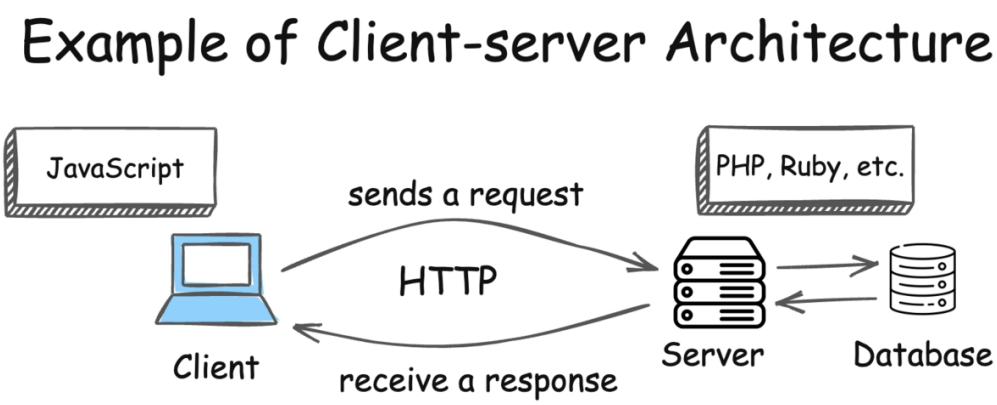
The email course would help you find the right path forward as a software developer.

CHAPTER 1: INTRODUCTION TO MERN STACK

The MERN stack is a collection of technologies that you can use to build a full-stack web application using JavaScript. MERN is short for MongoDB, Express, React, and Node.js.

If you're familiar with JavaScript, you might know that JavaScript was first created to make the web browser programmable. JavaScript exists and runs only inside web browsers.

A complete web application is made up of two parts: the frontend or client side, and the backend or server side. The client-server architecture below shows how a web application is developed:



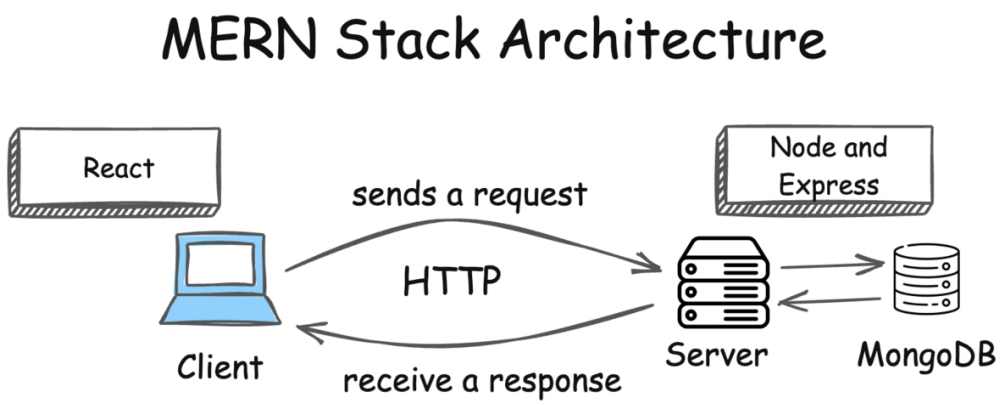
The client and server communicate using HTPP requests. When needed, a server might interact with the database to fulfill the request sent by the client.

Because JavaScript only runs in the browser, another programming language is required to build a web application. The most popular server-side programming languages are Python, Ruby, and PHP.

But all this changed with the invention of Node.js, which is a program that can run JavaScript outside of the browser.

By utilizing the power of Node.js, JavaScript can now be used to develop both the client and server side of a web application.

Using the MERN stack, you can develop a complete web application by using React to handle the client side, Node.js as the web server, Express as the web framework, and MongoDB as the database:



The MERN stack can be seen as a collection of libraries and tools that enables you to develop both frontend and backend systems using JavaScript.

By using the MERN stack, you gain the following benefits:

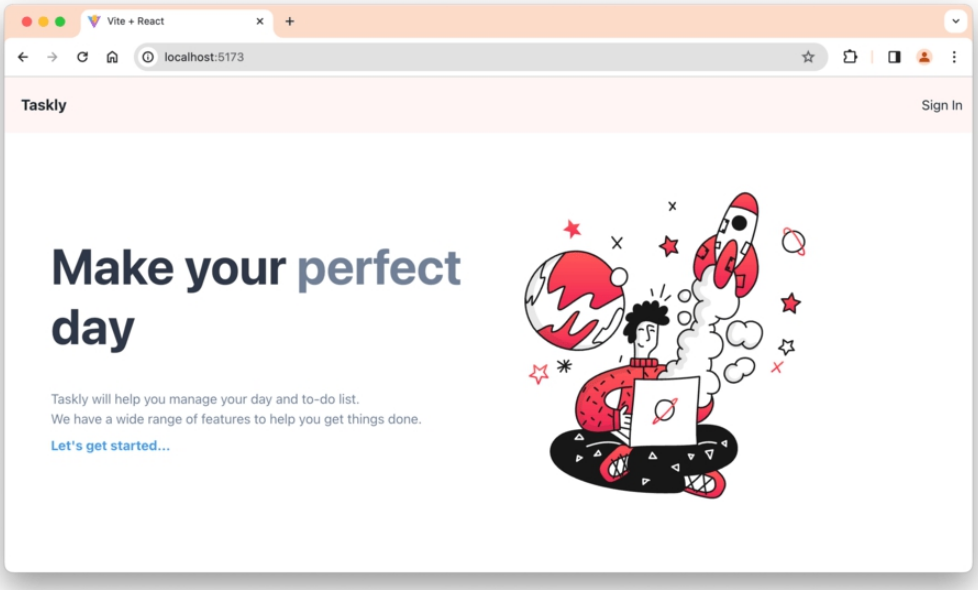
- Develop a complete web application using one language (JavaScript)
- Develop the client and server side in isolation
- Using open source technologies that are maintained by developers worldwide
- Great support ecosystem because developers like open source technologies

In short, Learning the MERN stack enables you to do web development using one language and opens more career opportunities.

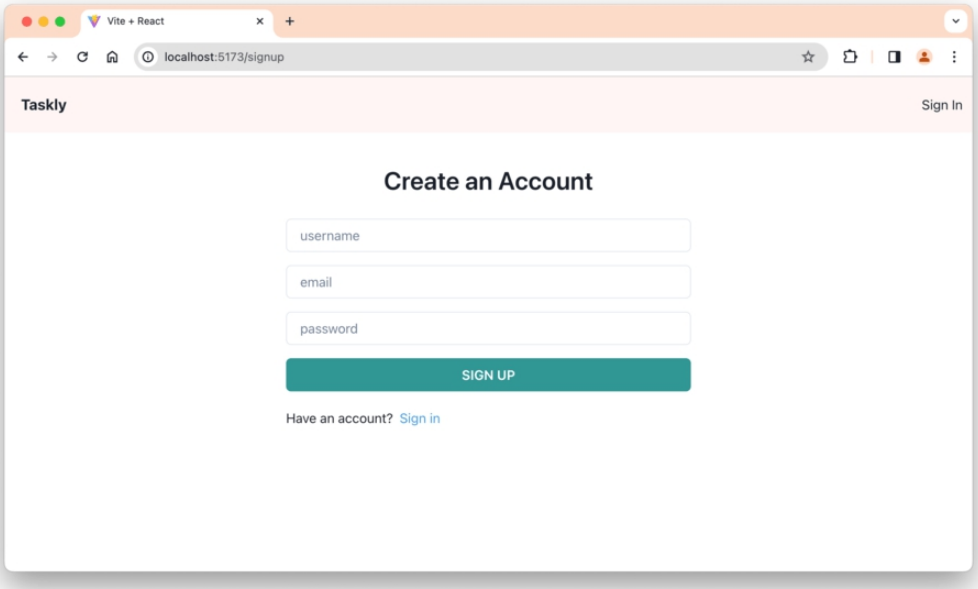
The Exercise Application

To make the learning practical, we're going to build a web application and deploy it to the public. The application we're going to build is a task management application.

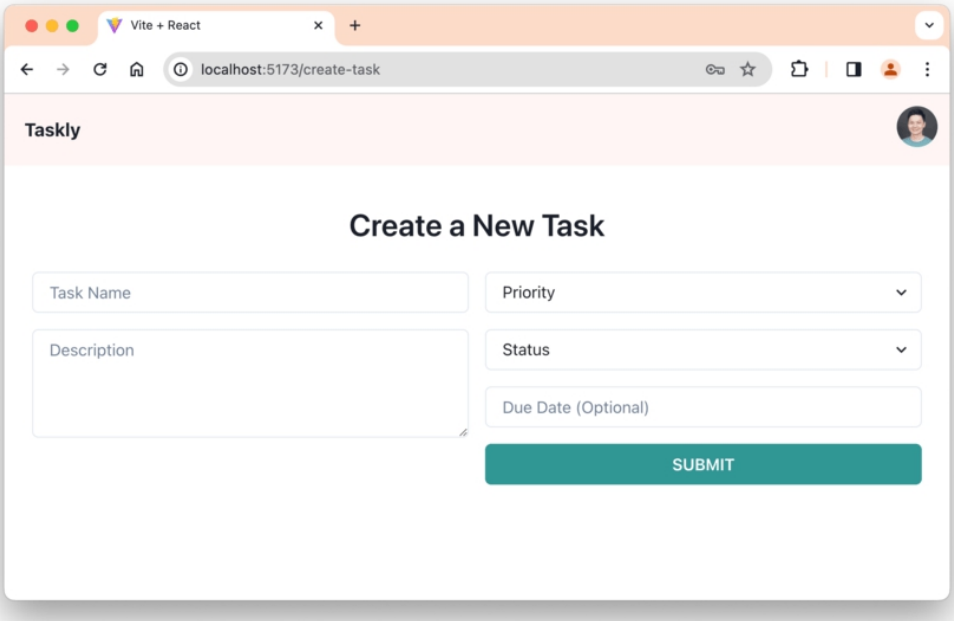
In this application, you'll be able to add, update, delete, and filter tasks by a specific attribute:



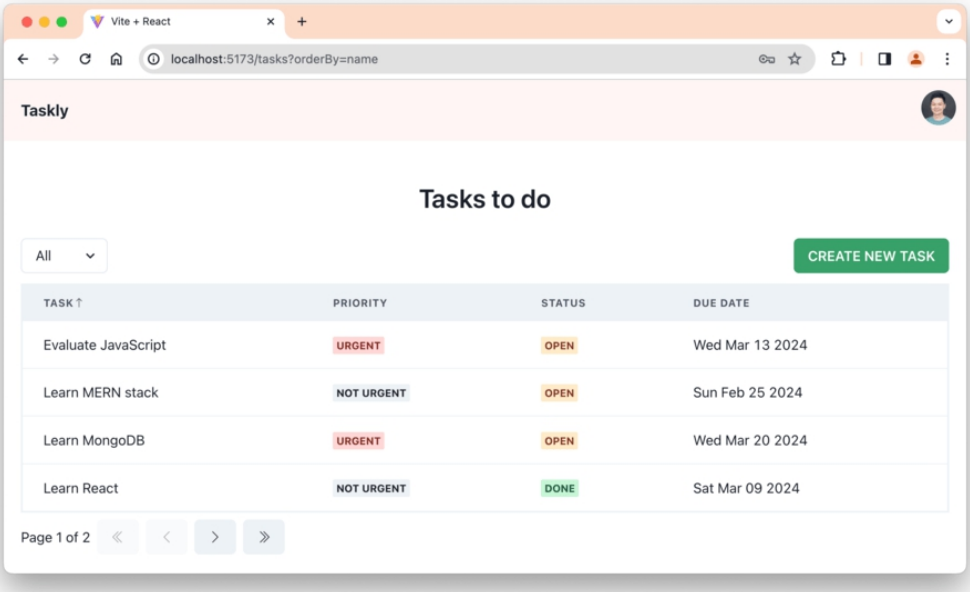
Before you can create a task, you need to register for an account using a username, email, and password:



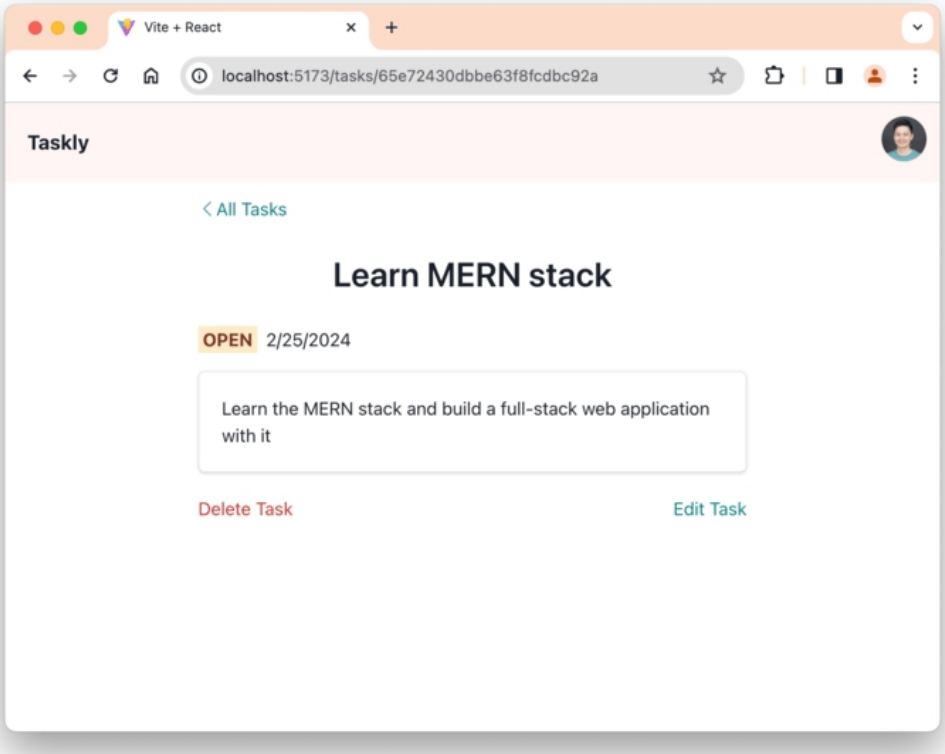
After signing up, you'll be able to add a new task to the application:



You can view all tasks created by the user. All common patterns in creating user interfaces like filter, paginations, and sort will be added to this page:



When you click on one of the tasks, you will see the full detail of the task. You can also update or delete the task:



With this practice, you will get the experience of developing an application using MERN stack from scratch, as well as understanding how to organize your code well.

Computer Setup

To start developing with the MERN stack, you need to have three things on your computer:

1. A web browser
2. A code editor
3. Node.js

Let's install them in the next section.

Installing Chrome Browser

Any web browser can be used to browse the Internet, but for development purposes, you need to have a browser with sufficient development tools.

The Chrome browser developed by Google is a great browser for web development, and if you don't have the browser installed, you can download it here:

https://www.google.com/chrome/

The browser is available for all major operating systems. Once the download is complete, follow the installation steps presented by the installer to have the browser on your computer.

Next, we need to install a code editor. There are several free code editors available on the Internet, such as Sublime Text, Visual Studio Code, and Notepad++.

Out of these editors, my favorite is Visual Studio Code because it's fast and easy to use.

Installing Visual Studio Code

Visual Studio Code or VSCode for short is a code editor application created for the purpose of writing code. Aside from being free, VSCode is fast and available on all major operating systems.

You can download Visual Studio Code here:

https://code.visualstudio.com/

When you open the link above, there should be a button showing the version compatible with your operating system as shown below:

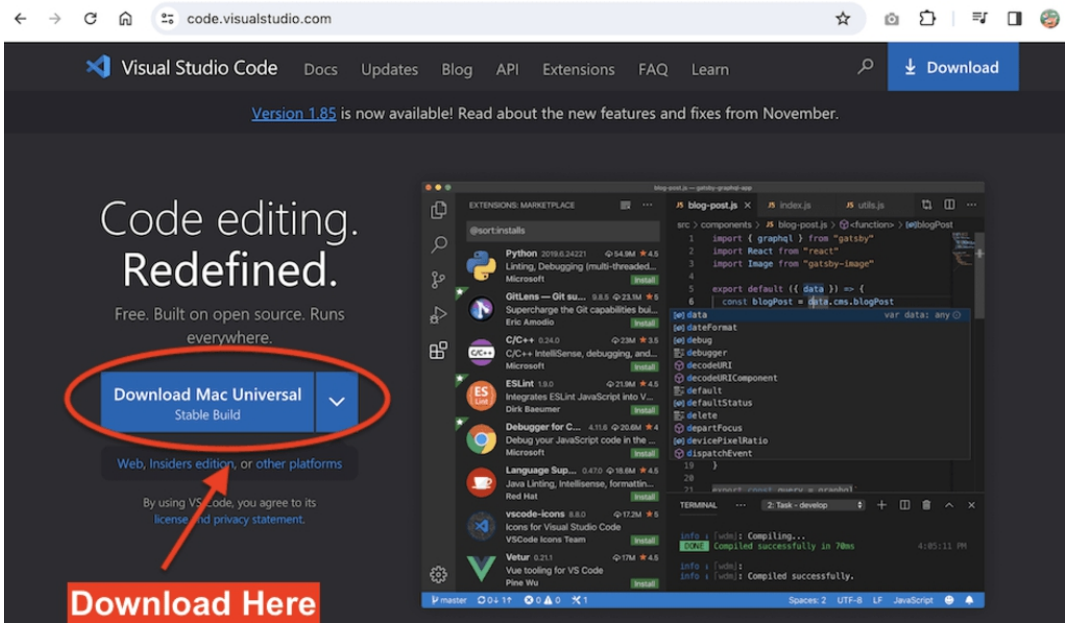


Figure 1. Install VSCode

Click the button to download VSCode, and install it on your computer.

Now that you have a code editor installed, the next step is to install Node.js

Installing Node.js

Node.js is a JavaScript runtime application that enables you to run JavaScript outside of the browser. We need this program to generate and run the web application that we're going to develop.

You can download and install Node.js from https://nodejs.org. Pick the recommended LTS version because it has long-term

support. The installation process is pretty straightforward.

To check if Node has been properly installed, type the command below on your command line (Command Prompt on Windows or Terminal on Mac):

```
node -v
```

The command line should respond with the version of the Node.js you have on your computer.

You now have all the programs needed to start developing a MERN web application. Let's create an Express application for the backend part in the next chapter.

Summary

In this chapter, we've learned what is the MERN stack and why it's a great choice, and then we installed the required tools to develop a MERN application.

If you encounter any issues, you can email me at nathan@codewithnathan.com and I will do my best to help you.

CHAPTER 2: YOUR FIRST EXPRESS APPLICATION

In this chapter, we're going to create the backend part of the project using Node.js and Express.

First, create a folder on your computer that will be used to store all files and code related to this project. You can name the folder 'taskly'.

Inside that folder, create a new folder named 'server'. Here's where you will code the backend part of the web application.

Inside the 'server' folder, open your terminal and run the npm command to create a new JavaScript project:

```
npm init
```

npm stands for Node Package Manager. It's a program used for creating and installing JavaScript libraries and frameworks.

The `npm init` command is used to initialize a new project and create the `package.json` file. It will ask several questions about your project, such as the project name, version, and license.

For now, just press Enter on all questions until you see the following output:

```
About to write to /taskly/server/package.json:

{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this OK? (yes)
```

Press Enter, and you should see a `package.json` file generated in the 'server' folder containing the same information as shown above.

Now you need to install the packages required for this project. Run the command below:

```
npm install express
```

The `express` package is the Express web framework that we need to create a web server.

When the installation is finished, you should see a `package-lock.json` file and a `node_modules/` folder generated.

The `package-lock.json` file records the exact version of every dependency you installed on the project.

The `node_modules/` folder store all packages you installed using the `npm install` command.

In the `package.json` file, you should also see the installed packages listed as follows:

```
{
  // ...Other info
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.3"
  }
}
```

The `dependencies` property contains the packages and their version numbers. This way, we can keep track of the installed packages and avoid breaking the application because of incompatible package versions.

If you open the `node_modules/` folder, you will see the `express` folder along with many other folders. These folders are packages used by `express` itself.

We don't know what these packages do, but this is one of the benefits in using npm packages. You can use other people's library and framework in your project without knowing all their details.

Add type:module to Package.json File

By default, Node.js supports importing packages in a JavaScript file using the CommonJS Module pattern.

The CommonJS Module enables you to import JavaScript packages using the `require()` function. For example, here's how to import the `express` package in your JavaScript file:

```
const express = require('express');
```

But nowadays, the EcmaScript Module (or ESM) is the standard way to import packages. The syntax is as follows:

```
import express from 'express';
```

To use the ESM syntax in our project, we need to add a `"type": "module"` property in our `package.json` file as shown below:

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",  // <-- Add type here
  // Other info...
}
```

This way, we can use the ESM syntax in our code.

Creating the Server File

It's time to create your first Express application. On the 'server' folder, create a new file named `server.js` and write the following code in it:

```
import express from 'express';

const app = express();
const PORT = 8000;
```

```
app.use('*', (req, res) => {
  res.status(404).json({ message: 'not found' });
});

// Start the Express server
app.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}`);
});
```

Here, the `express` package is imported, and we create a new Express server by calling the `express()` function.

A specific `PORT` is defined so that our Express application always runs on that port. Here, we specify `8000`.

Next, the `app.use()` function is called to create a wild card (*) route. This route is created so that Express can receive HTTP requests.

Express passes two arguments to the route handler, the `req` stands for request object, which contains data regarding the request received by Express.

The `res` argument stands for response, which you can use to send a response back to the client.

Here, the response for the route is specified as a JSON string ({ `message: 'not found'` }) with a `404` status.

After that, the `app.listen()` function is called to open a specific web port for connections.

Running Express Server

Let's run the server now. Open your `package.json` file, and change the `test` script into a `start` script as shown below:

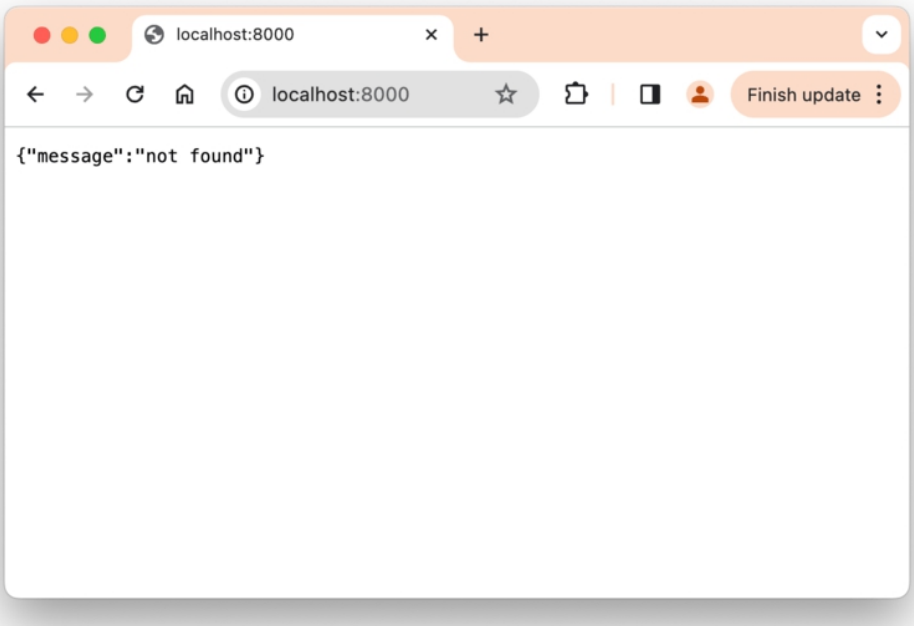
```
"scripts": {
  "start": "node server.js"
}
```

A `package.json` script is an alias of the command you write as the value of the script, this means that when you run the `npm start` command, the `node server.js` command will be executed.

In the terminal, run the `npm start` command and you will see the following output:

```
> server@1.0.0 start
> node server.js

Server listening on port 8000
```

Now open your browser and head towards 'http://localhost:8000', you will see the response as follows:



No matter what URL you request, the response will be the same because no other routes have been defined yet.

Adding an API Route

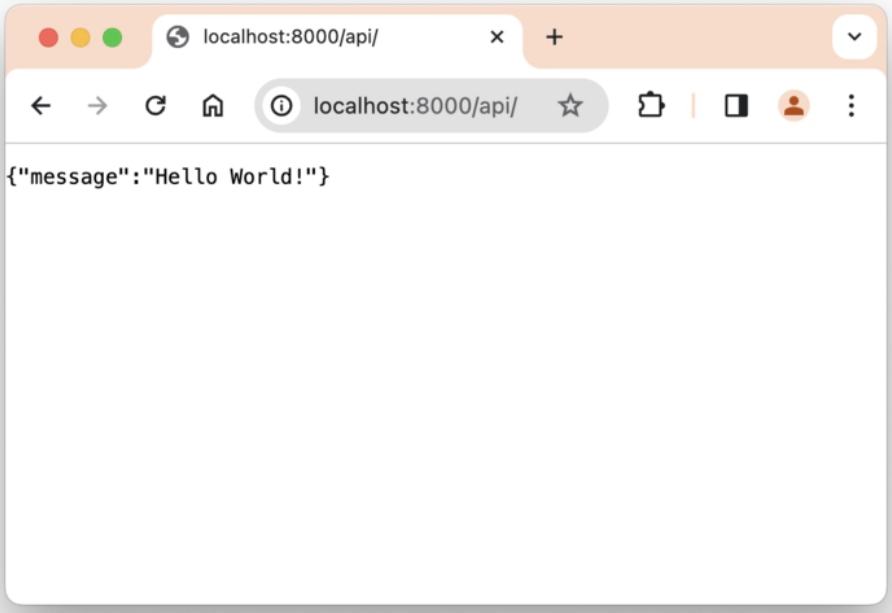
Let's create a new API route as follows:

```
// Create a new '/api' route
// Make sure its above the '*' route
app.use('/api', (req, res) => {
  res.status(200).json({ message: 'Hello World!' });
});

app.use('*', (req, res) => {
  res.status(404).json({ message: 'not found' });
});
```

The code you added to the `server.js` file won't take effect immediately. You need to stop the Express server using `Control`

+ C on the terminal, then run `npm start` again for the changes to work.

Now you can go back to the browser and refresh the page. You should see the following output:



As you can see, the new route is working because the `message` has been changed.

If you access other URLs such as localhost:8000/a or localhost:8000/b, you will get the 'not found' message again.

When creating an Express server, the order of the `app.use()` call matters. If you place the (*) route above any other route, all URL patterns will match that route first and return a 'not found' response.

Adding Nodemon for Development

In the previous section, you probably feel some inconveniences when creating the API route.

Notice that after adding the route and saving the changes, you need to run the `npm start` command again for the changes to work.

You're going to add more code to this project in the following chapters, and if you still use `node`, then you have to do the restart every time you make some changes.

To make the development more pleasing, let's use Nodemon to run the server instead.

Nodemon is a monitoring script that will automatically restart Node.js when there's any change to the project.

To use Nodemon, You need to install it using npm first:

```
npm install nodemon --save-dev
```

The `--save-dev` option is used to specify packages that are only needed for development purposes.

Packages marked as `dev` dependencies won't be installed when you deploy the application to a production environment later.

If you open the `package.json` file, you should see `nodemon` listed as `devDependencies` like this:

```
"dependencies": {
  "express": "^4.19.1"
```

```
},
"devDependencies": {
  "nodemon": "^3.1.0"
}
```

After installing Nodemon, add a `dev` script on the `package.json` file as shown below:

```
"scripts": {
  "start": "node server.js",
  "dev": "nodemon server.js"
}
```

Now you can run the `dev` script by running the `npm run dev` command.

Nodemon will execute the `server.js` file and show the following output:

```
[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
Server listening on port 8000
```

To see Nodemon in action, open the `server.js` file and change the message returned by the API route as follows:

```
app.use('/api', (req, res) => {
  res.status(200).json({ message: 'Hello There!' });
});
```

You will see Nodemon automatically restart the server in the terminal as follows:

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server listening on port 8000
```

Now change the message back to 'Hello World!', and you'll see the restart message again on the terminal.

If you don't use Nodemon, then you'll have to restart the server manually when you make changes to the project. Thank you Nodemon!

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-2

You've managed to create your first Express application and created a route that responds to a URL request. Good job!

In the next chapter, we're going to create a MongoDB database and integrate it into our Express application.

CHAPTER 3: CREATING A MONGODB DATABASE CLUSTER

When developing a web application, a database is essential for storing and managing important information.

In this chapter, you're going to learn about the MongoDB database software and how to create one

Don't worry if you never learned or used MongoDB before. I'll introduce how the database works in a nutshell and guide you through creating the database.

MongoDB Introduction

If you already know about MongoDB, you can fast forward to 'Setting Up MongoDB Atlas Cloud' section below.

MongoDB is one of many databases that you can use to store web application data. It's a document-oriented database that provides flexible and scalable solutions for storing and retrieving data.

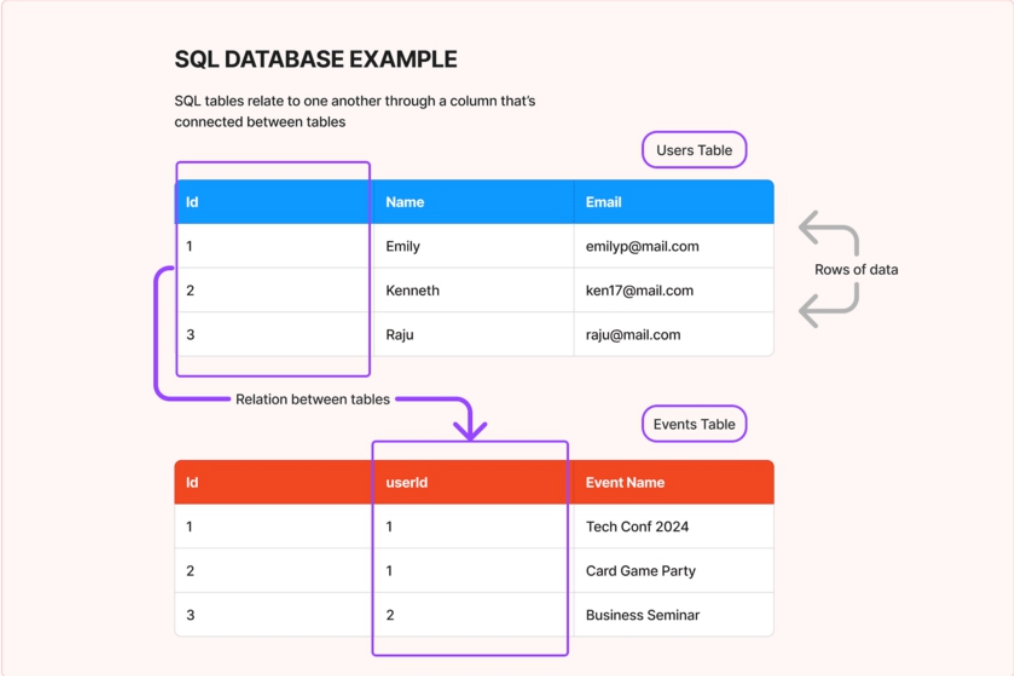
MongoDB is also known as a NoSQL database because it doesn't use the Structured Query Language (SQL) for accessing and

manipulating the stored data.

Some of the most popular SQL databases are MySQL, SQLite, and PostgreSQL. You might have heard or used them before.

In an SQL database, data is stored under tables and rows. Suppose we want to use the database to keep track of users and events that are created in our application.

Here's a representation of the data in an SQL database:



In the above example, the Users Table and the Events Table represent the `users` and `events` data respectively.

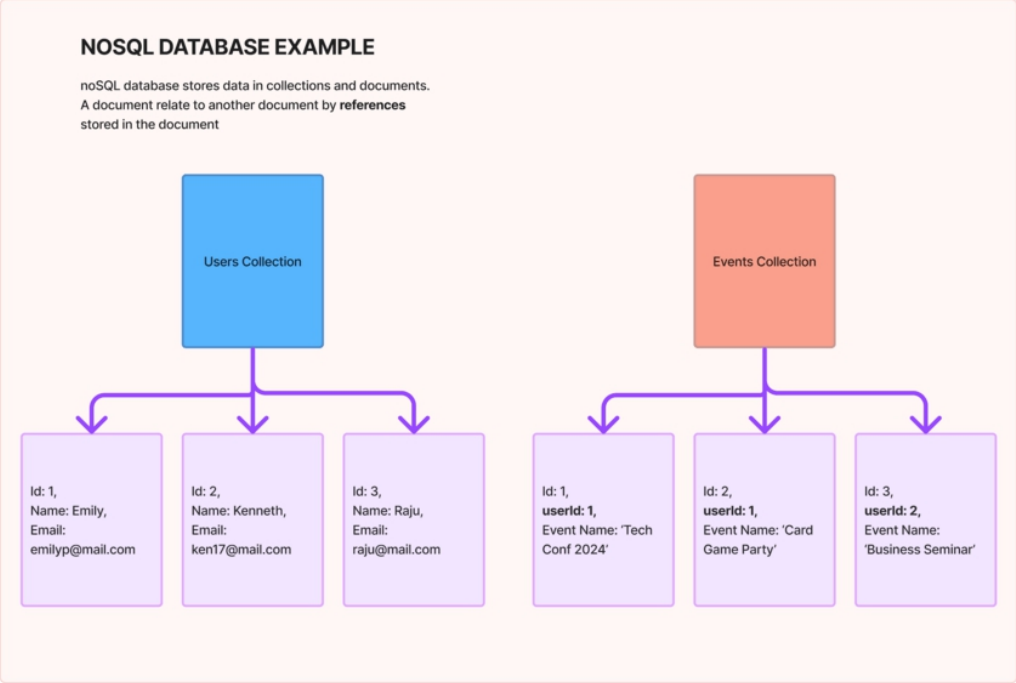
In each table, there are rows of data, and the `Id` column in the Users Table is related to the `userId` column in the Events Table.

Through the relation created in these tables, we can see that the first two events were created by the user 'Emily', while the third one was created by 'Kenneth'.

When you update the `Id` value in the Users Table, the `userId` value in the Events Table will be updated automatically.

By contrast, a NoSQL database like MongoDB represents the data as documents and collections.

A collection can have many documents, and each document can have as many entries of data. A document can store a reference to another document:



Since a NoSQL database is non-relational, the `userId` entry in the Events Collection won't be updated automatically when you update the `Id` entry in the Users Collection.

There are many articles available about the comparisons between NoSQL and SQL databases, but this will be enough for now.

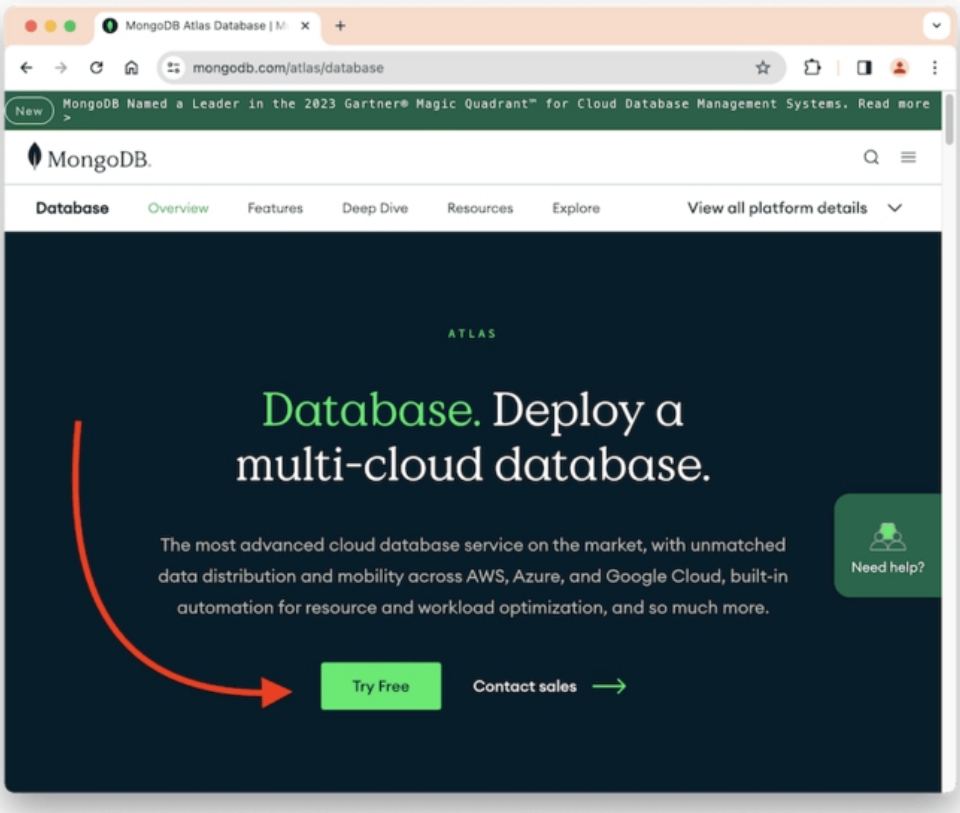
Let’s continue with setting up a MongoDB instance for our application to use next.

Setting Up MongoDB Atlas Cloud

The easiest and quickest way to get started with MongoDB is to sign up for its cloud service, which is MongoDB Atlas.

When using MongoDB Atlas, you don’t need to manually install and run MongoDB on your computer for development and testing. The cloud service will generate a server instance for MongoDB along with credentials to access the database.

MongoDB Atlas also has a free tier which is perfect for our Express server project, so head over to https://www.mongodb.com/atlas/database and click on the 'Try Free' button shown on the page:



You'll be taken to the registration page, where you can create an account for free:

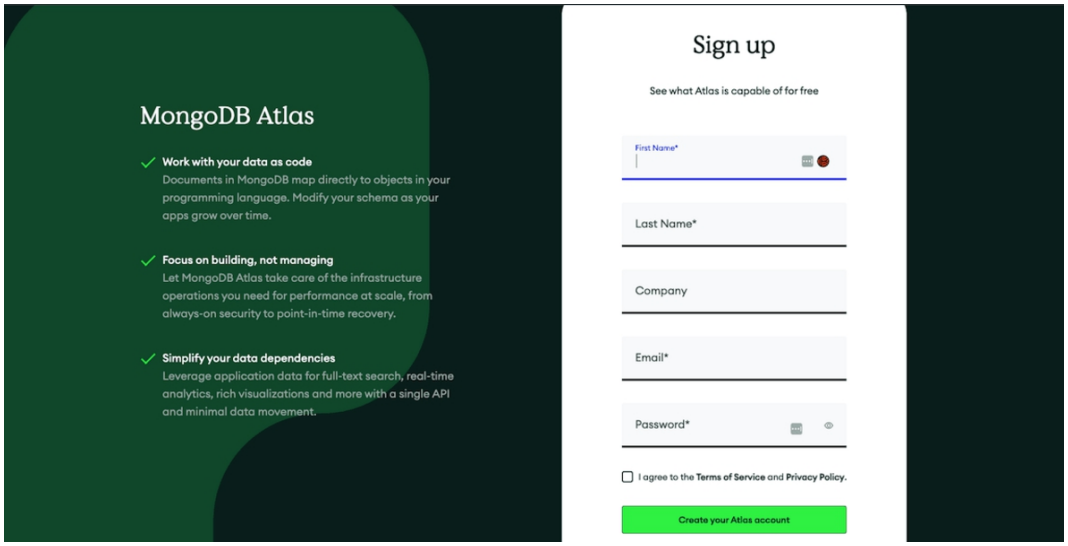


Figure 2. MongoDB Atlas Sign Up Page

1. Fill in the required fields.
2. When you’re done, click "Create your Atlas account".
3. Next, you will be sent an email verification. Verify your email by clicking the "Verify Email" button on that email as shown in the image below:

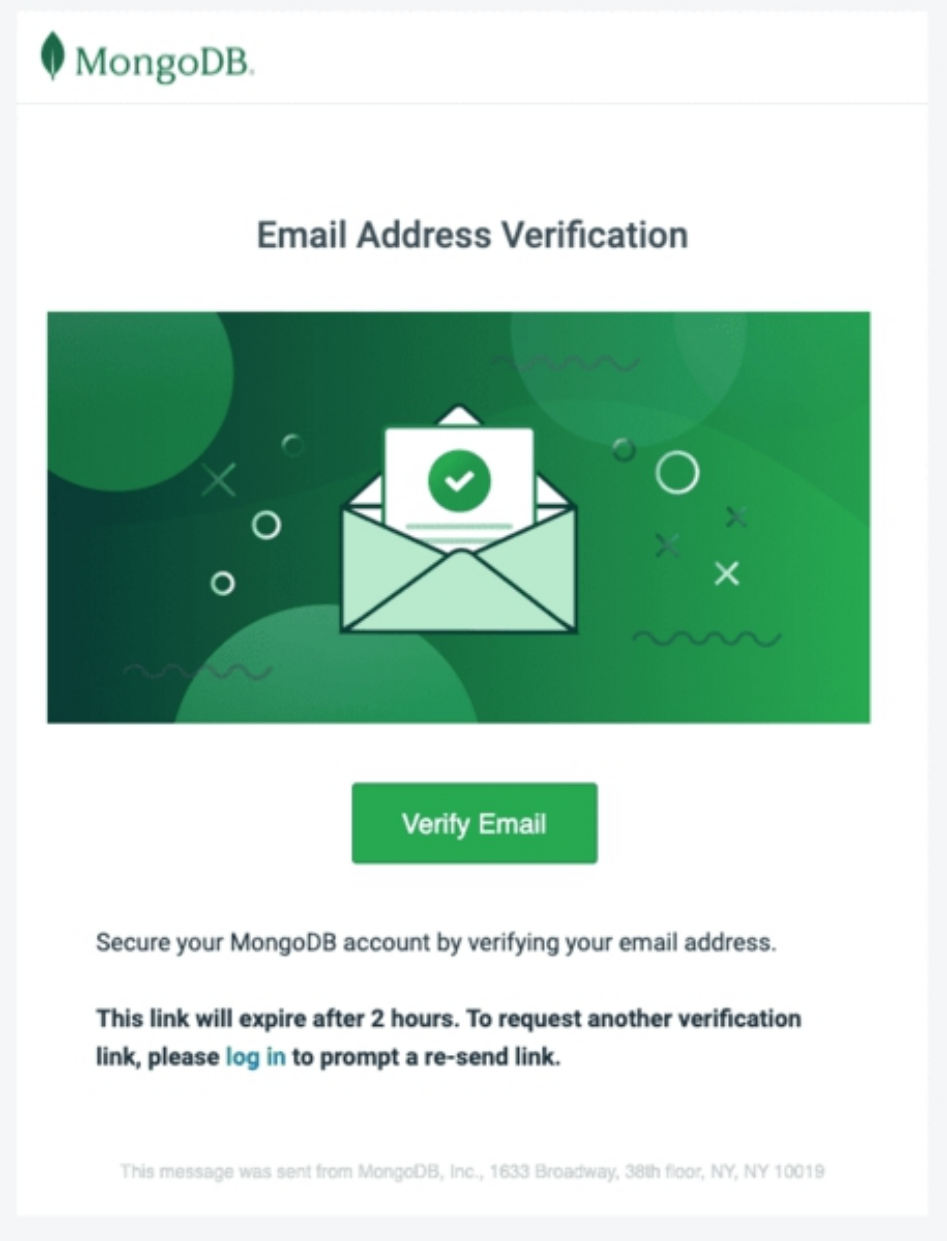


Figure 3. MongoDB Atlas Verify Email

1. After verifying your email, head back to the [Login page](https://account.mongodb.com/account/login) and login using your credentials.

Welcome to Atlas

Upon logging in for the first time, you will be presented with a "Welcome to Atlas!" page.

1. For "What is your goal primary goal?", select "Learn MongoDB".
2. For "How long have you been developing software with MongoDB??", select "Less than a year".
3. Select "JavaScript / Node.js" as your primary language.
4. For "What kind(s) of data will your project use?" and "Will your application include any of the following architectural models?", you can select any answer. Select "Not sure/None" if you want to skip these.

Your form should look like the image below:

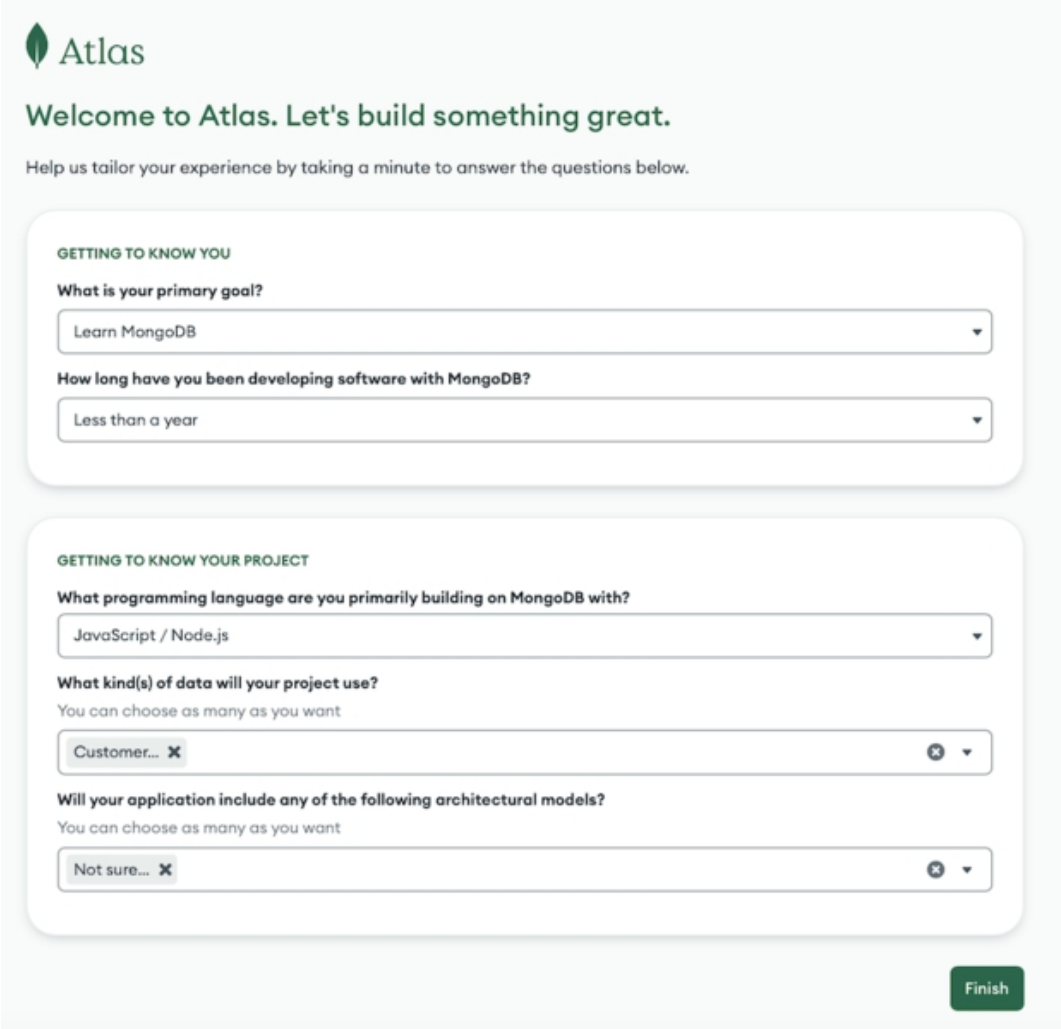


Figure 4. MongoDB Atlas Welcome Form

Create a Cluster

Upon completing the form, you will be directed to the *Deploy your database* page.

If you arrive at the Overview page instead, you can click the big green + *Create* button to go to the *Deploy your database* page:

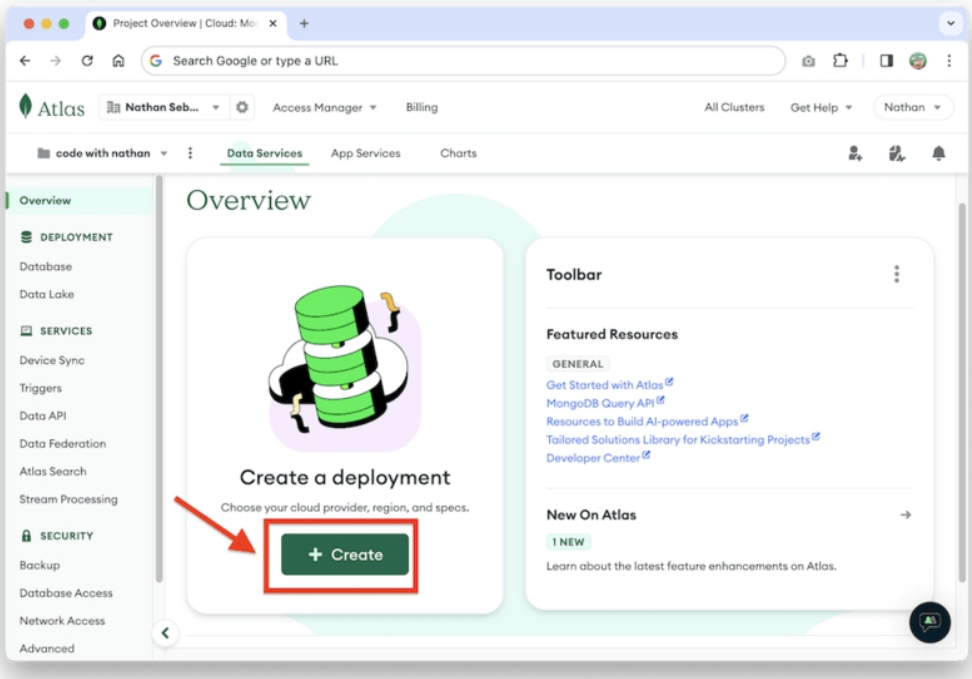


Figure 5. MongoDB Atlas Overview Page

On the *Deploy your database* page, Select the *M0 FREE* plan, which is ideal for testing and development:

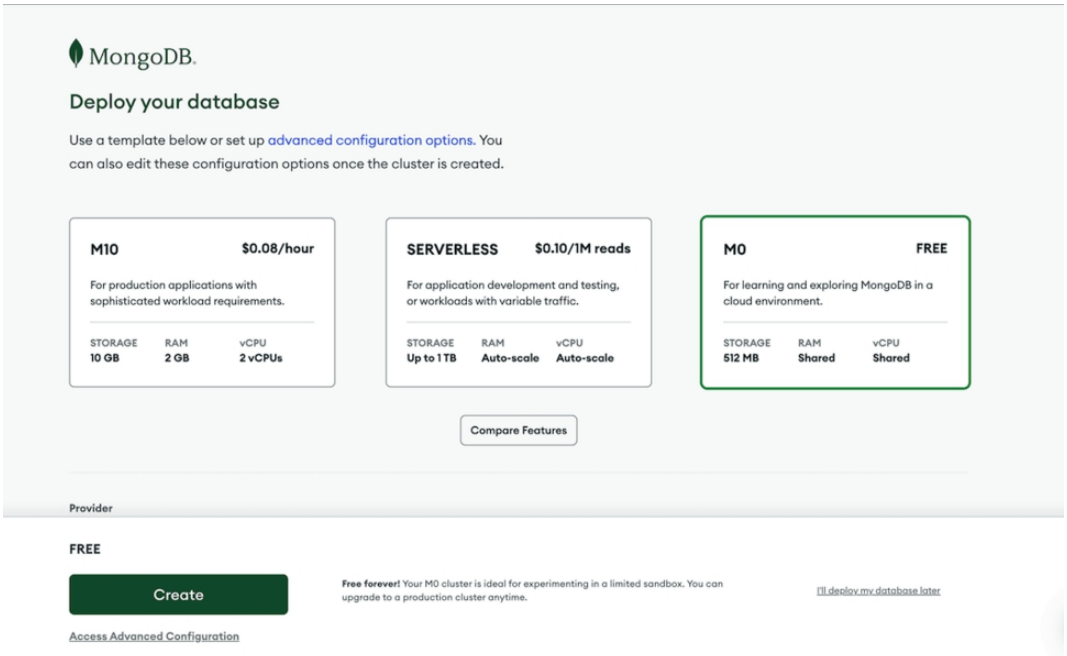


Figure 6. MongoDB Atlas Plans

The *Provider* option is the cloud service provider that will host your database. You're free to select whichever you like.

AWS, Google Cloud, and Azure all offer the same free tier. For the region, select the region that's recommended or is closest to you.

You can accept the default cluster name provided, usually 'Cluster0'. Leave the tag empty as it's optional. After selecting your provider and nearest region, you can click the green "Create" button:

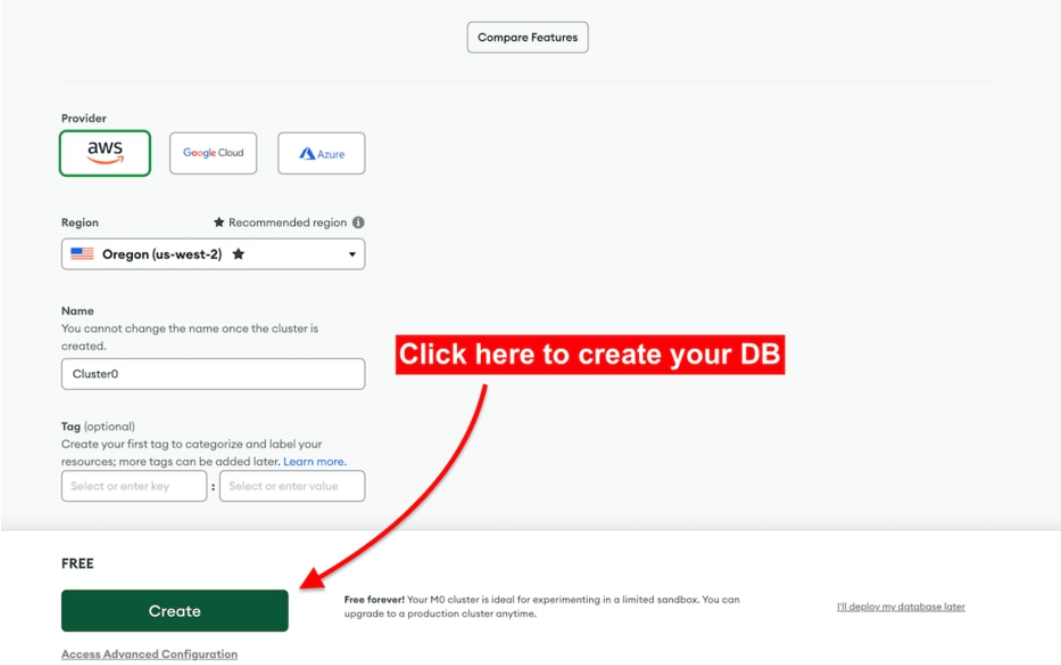


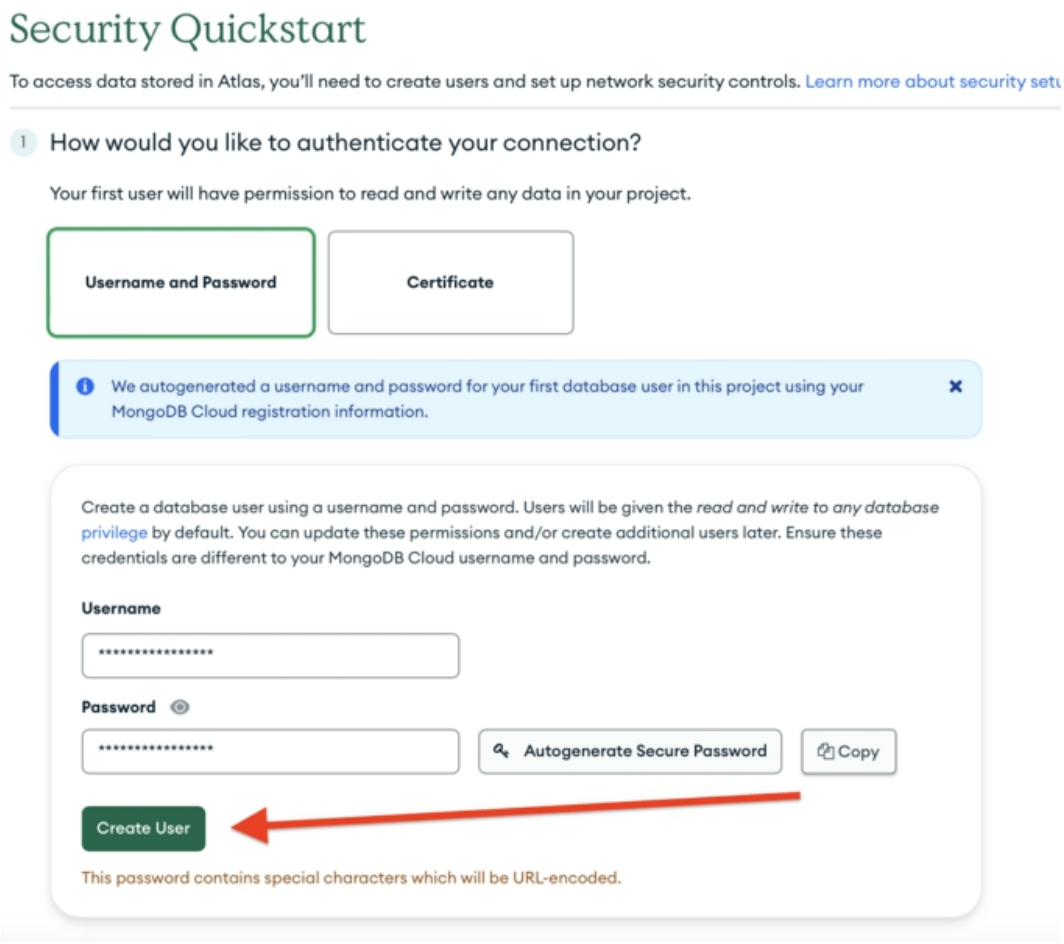
Figure 7. MongoDB Atlas Create Database

Now you need to wait a bit for the cluster to be created by MongoDB. If this is the first time you use MongoDB Atlas, you'll be taken directly to the *Security Quickstart* page after the cluster is created.

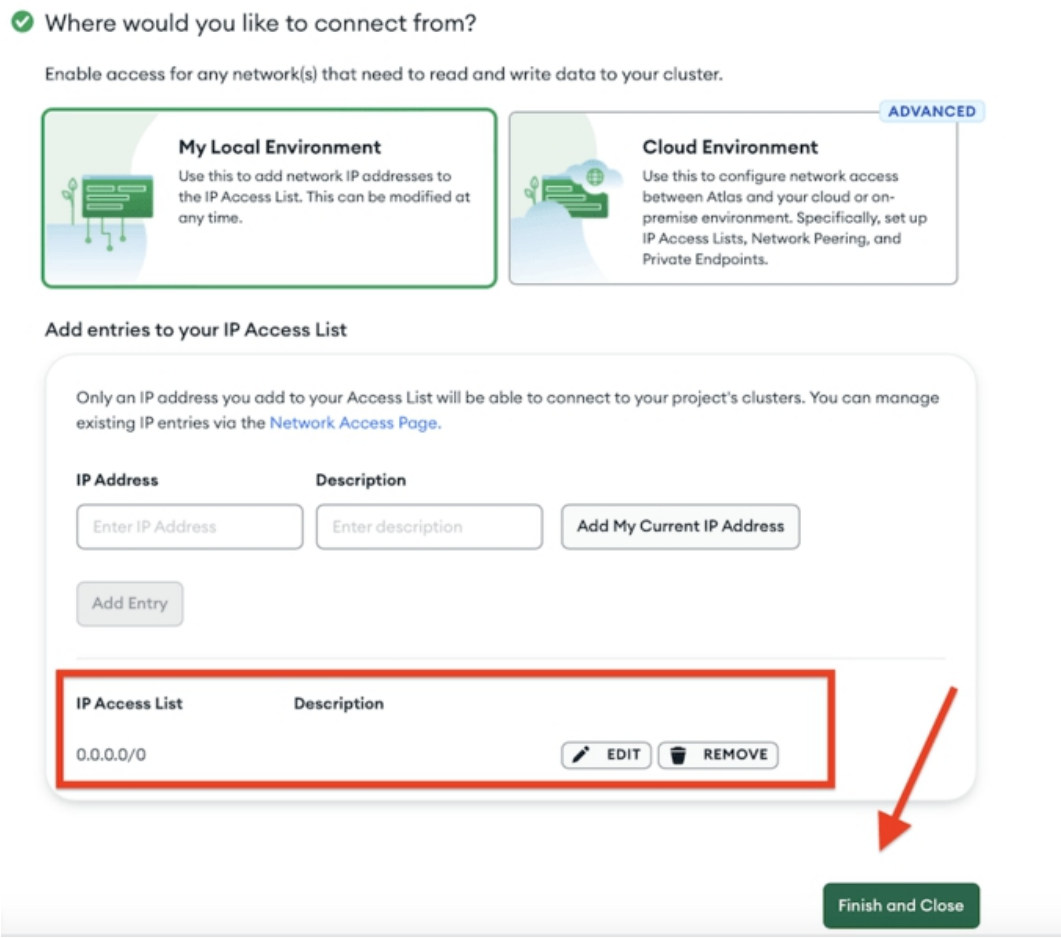
Security Quickstart: Set Authentication and Connection

Here, you'll be asked two questions. First is how to authenticate connections to your database.

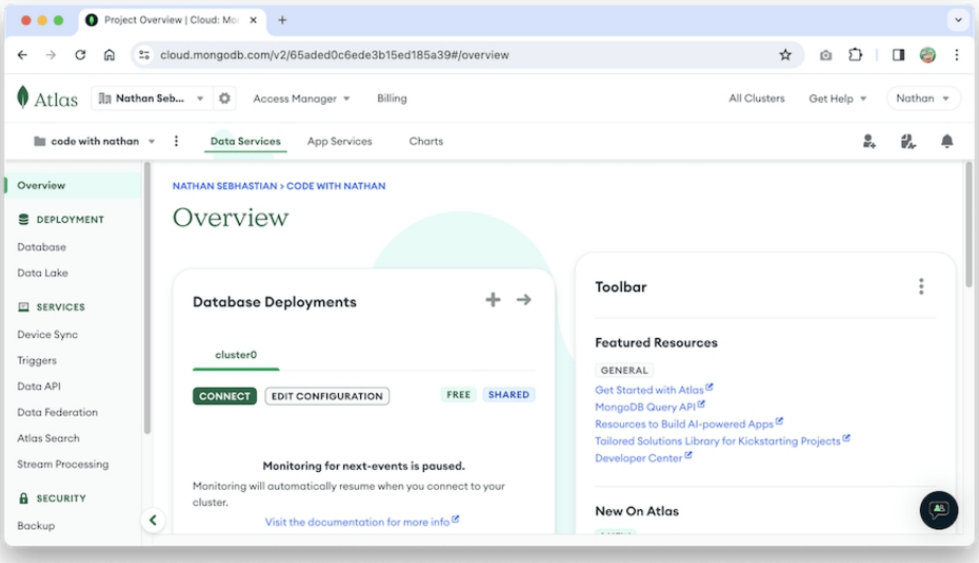
You can select Username and Password, then use the Username and Password that have been generated for you. Click *Create User* as shown below:



Next, you need to enable access to the database from specific IP addresses. Because this is a development database, you can enter '0.0.0.0' as the IP address value to enable access from anywhere:



Now you can click *Finish and Close*. You should be redirected to the *Overview* page, where your active cluster is shown:



The cluster is ready to accept incoming connections. Great work!

Add a New Database User

There may be times when you want to add another user or whitelist an IP address. Because the *Security Quickstart* is gone after you create a cluster, you need to navigate the Atlas menu.

To add a new database user, follow these steps:

1. On the left-hand navigation menu, under Security, select the *Database Access* link.
2. From there, click the *Add New Database User* button just above the Users table. The following image shows the create user modal with the options you'll need to select:

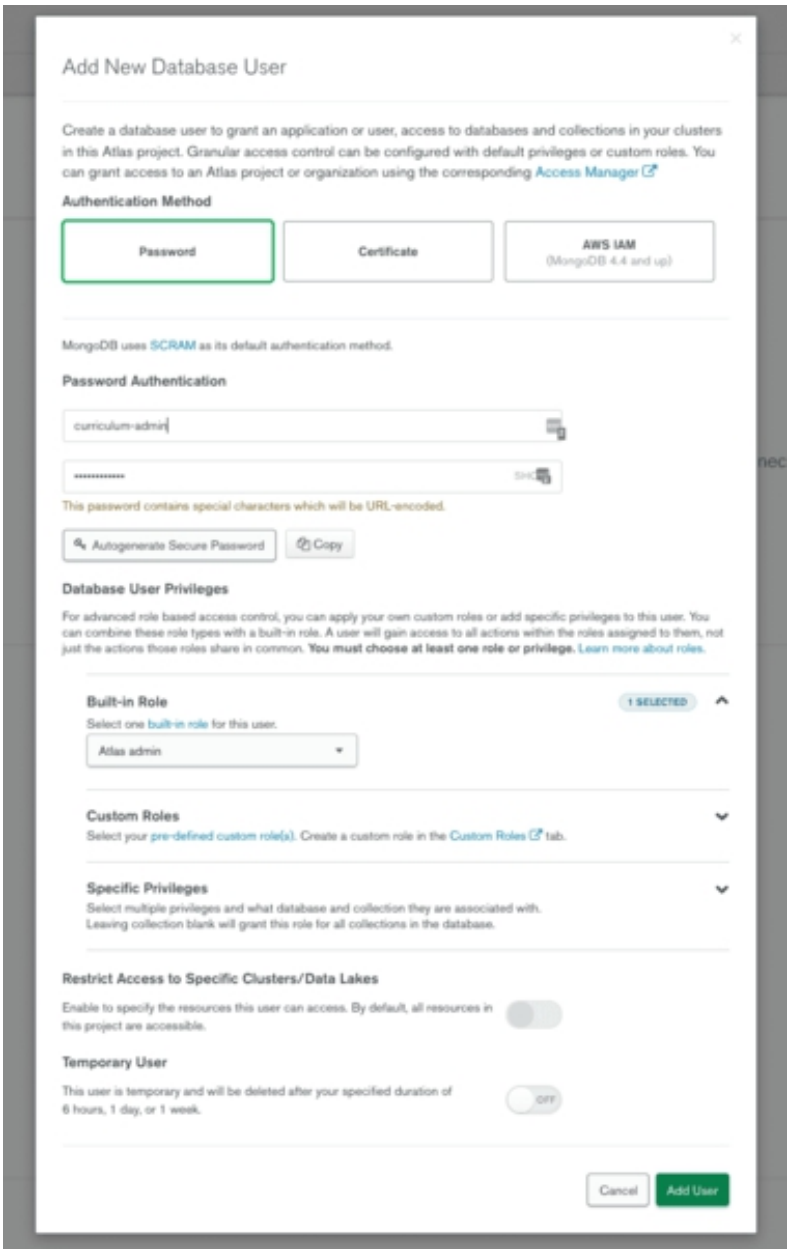


Figure 8. Adding a New User to Database

To fill out the form, follow these steps:

- For *Authentication Method*, choose *Password*.
- Under *Password Authentication*, create a Username and Password that you'll remember.
- Under *Database User Privileges*, select *Atlas admin*.

Leave any remaining options as default.

- IMPORTANT: Do not enable *Temporary User* unless you want to make a new user every so often.

When you're done, click *Add User*.

Allow Your IP Address

To allow a new IP address, you need to select the *Network Access* menu from the *Security* tab on the left side, then add the IP address you want.

On the Network Access page, click the *Add IP Address* green button and a modal will appear:

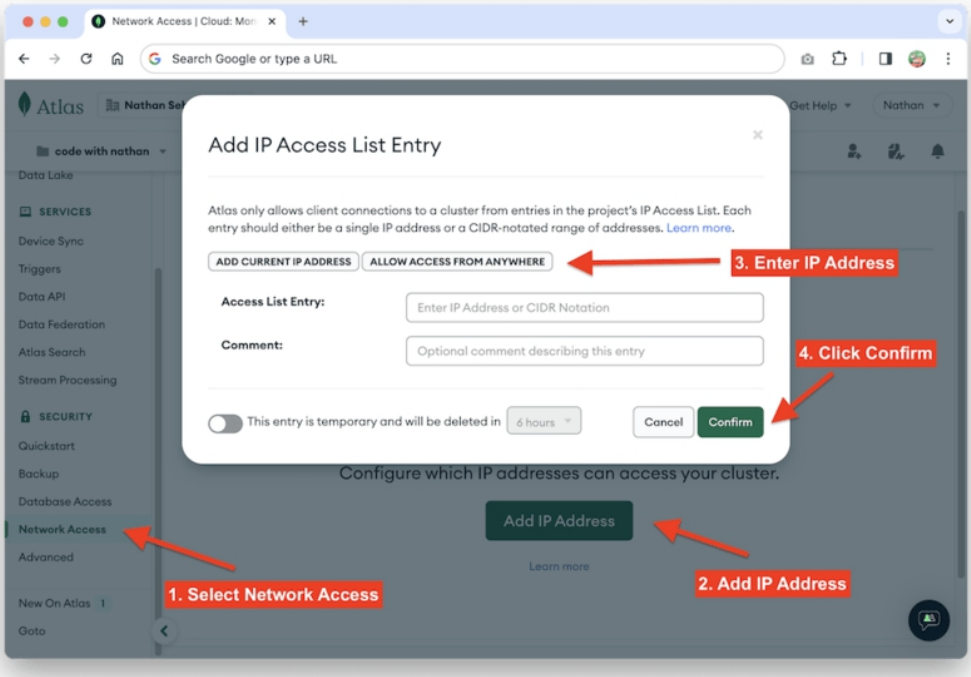


Figure 9. Whitelisting an IP Address

You can add your IP address in the modal shown above. You can also enable connection from anywhere by clicking the *Allow Access From Anywhere* button, or just add your IP address with the *Add Current IP Address* button.

Click *Confirm* when you're done, and that's how you add a new user or IP address in MongoDB Atlas.

With your database cluster ready, it's time to try connecting your Express application to the database.

Summary

Note that there's no code repository for this chapter because we didn't add any.

In this chapter, we've learned what is MongoDB database, the difference between MongoDB and SQL databases, and how to set up a MongoDB Atlas cluster for free.

A MongoDB cluster can have many databases, although you usually only need one database for a single application.

In the next chapter, we're going to see how to connect the MongoDB cluster to our Express application.

CHAPTER 4: INTEGRATING MONGODB TO EXPRESS APPLICATION

In this chapter, we're going to send a connection request from our Express application to the MongoDB cluster we created in the previous chapter.

We're also going to populate our database with some dummy data. Let's get going!

Connecting to MongoDB Cluster

To connect to the MongoDB cluster that we created earlier, we need to install the `mongodb` package using npm:

```
npm install mongodb
```

Next, create a file named `dbConnect.js` in your `libs/` folder and write the following code in it. I will explain the code below:

```
import { MongoClient, ServerApiVersion } from "mongodb";

const { MONGODB_URI, MONGODB_DATABASE } = process.env;
```

```
const client = new MongoClient(MONGODB_URI, {
  serverApi: {
    version: ServerApiVersion.v1,
    strict: true,
    deprecationErrors: true,
  },
});

try {
  // Connect the client to the server
  await client.connect();
  // Send a ping to confirm a successful connection
  await client.db().command({ ping: 1 });
  console.log("Connected to MongoDB!");
} catch (err) {
  console.error(err);
}

export const db = client.db(MONGODB_DATABASE);
```

In this file, we import the `MongoClient` class to set up a connection to the database.

The `ServerApiVersion` object is used to declare the MongoDB stable API we want to use. This is specified to ensure that the connection instance we create has all the methods declared for that API version.

This way, we can upgrade the MongoDB server API version at will and avoid breaking our application.

We use this when creating the connection instance as follows:

```
const client = new MongoClient(MONGODB_URI, {
  serverApi: {
    version: ServerApiVersion.v1,
    strict: true,
    deprecationErrors: true,
  },
});
```

As of this writing, `ServerApiVersion` only has `v1`, but more versions are promised to come.

After creating a new `client` instance, you need to call the `connect()` method to connect to the MongoDB cluster.

Once connected, you send a `ping()` to the cluster to test the connection. Wrap this code in a `try-catch` block to handle any error:

```
try {
  // Connect the client to the server
  await client.connect();
  // Send a ping to confirm a successful connection
  await client.db().command({ ping: 1 });
  console.log("Connected to MongoDB!");
} catch (err) {
  console.error(err);
}
```

Once connected, we select the database we want to use, and export it so that other modules can use it:

```
export const db = client.db(MONGODB_DATABASE);
```

Also, notice that this connection process requires two environment variables named `MONGODB_URI` and `MONGODB_DATABASE` to run properly:

```
const { MONGODB_URI, MONGODB_DATABASE } = process.env;
```

To get this URI, you need to go back to MongoDB Atlas Overview page, and click the *Connect* button on your cluster card:

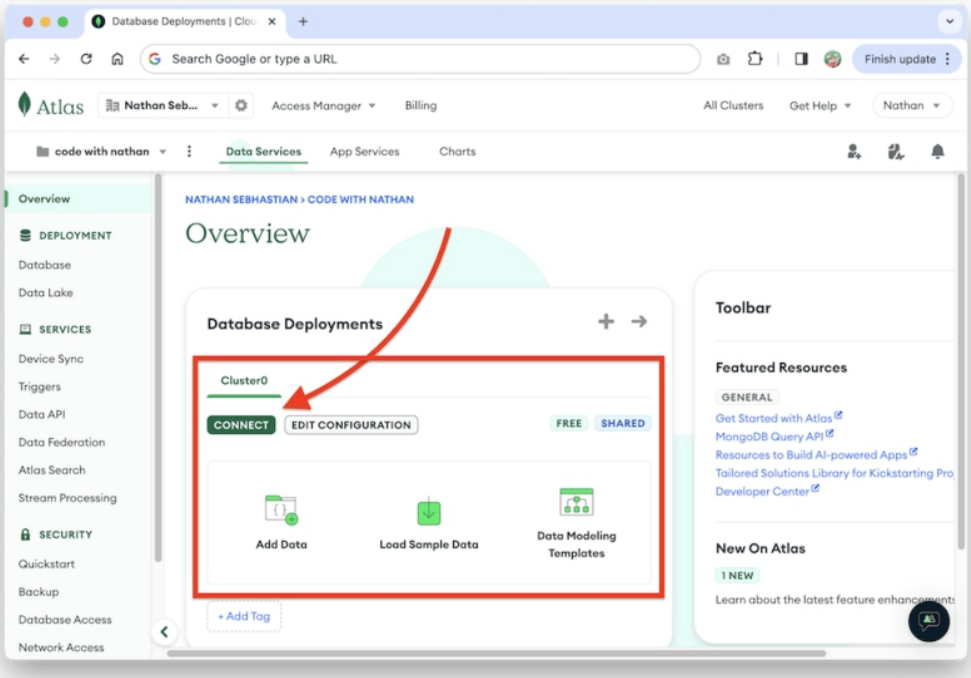
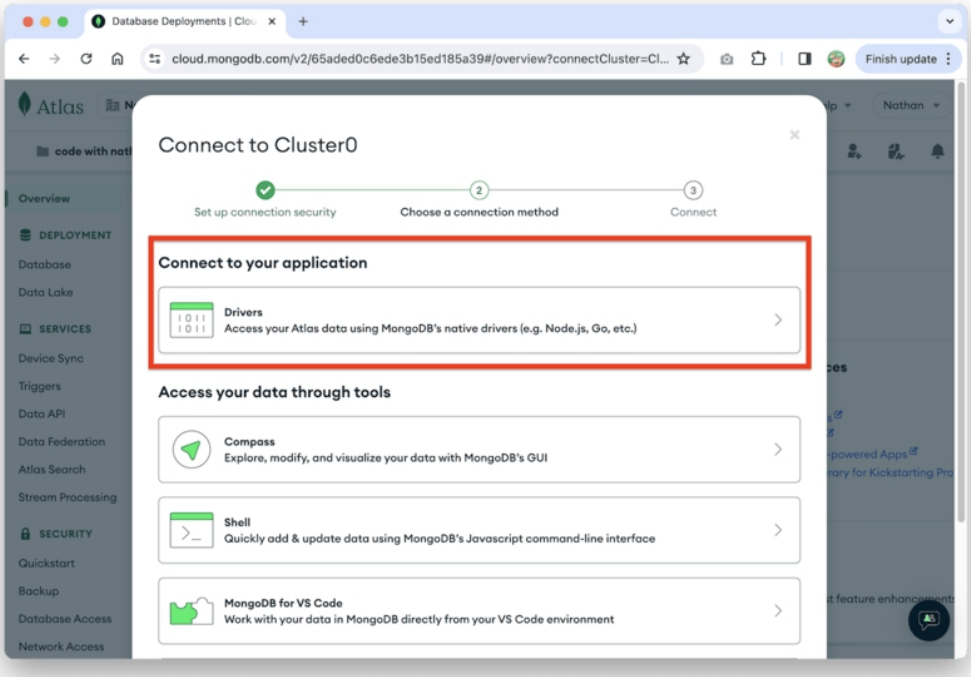
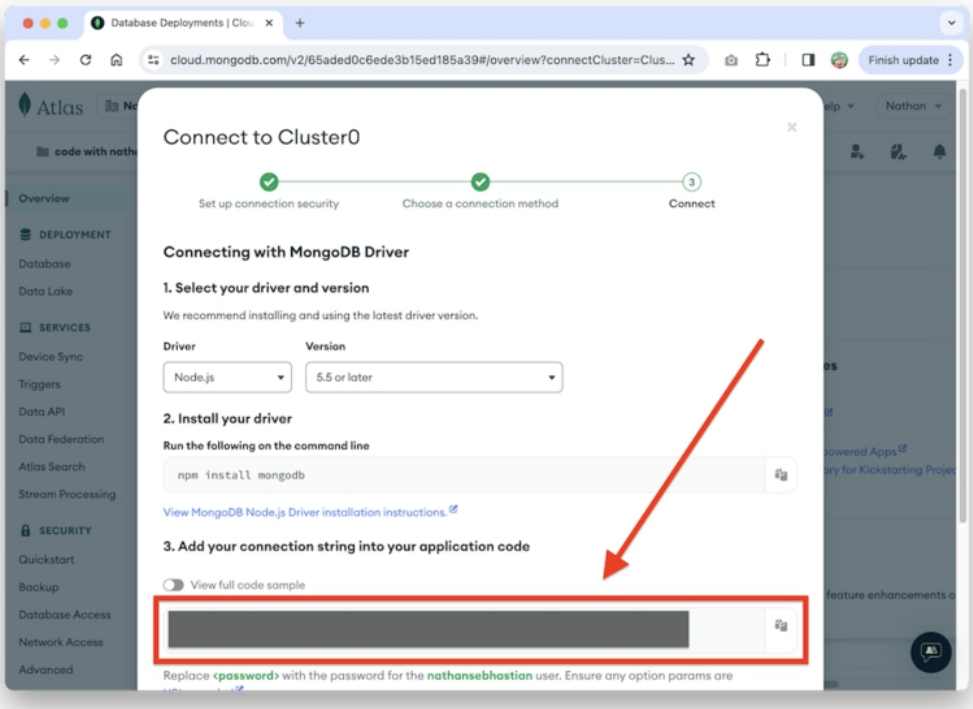


Figure 10. Click the Connect button in MongoDB Cluster

You will see a tab where you can choose the connection method. Choose *Driver* as shown below:



You'll be shown the connection string URI on the next page as shown below:



Copy the connection string, then create a `.env` file in your application root folder with the following content:

```
MONGODB_URI=<Paste your connection string here>

MONGODB_DATABASE=<Define database name>
```

Note that you also need to replace the `<password>` string in the URI with your user's password.

If you forgot the password, you need to create a new password by selecting *Database Access* and editing the password of the user you created earlier.

For the `MONGODB_DATABASE` variable, put the name of the database you want to use for this application. I use 'taskly', but you can use any name you like.

Because we're using environment variables, we need to install the `dotenv` package in our Express project:

```
npm install dotenv
```

To test the MongoDB connection module, we can import the `db` variable from `dbConnect.js` file on our `server.js` file as shown below:

```
import 'dotenv/config';
import { db } from './libs/dbConnect.js';
```

Make sure that the `dotenv/config` import is above the `db` import, or you'll have an error.

Now run the server using `nodemon server.js` command, and you should see the following output on the terminal:

```
Connected to MongoDB!
Server listening on port 8000
```

This means you've created a connection between the Express application and the MongoDB cluster. Nice work!

Seeding Data to MongoDB

Now that you can connect to MongoDB cluster, let's populate the database with some dummy data that you can use in your application.

At the root folder of your project, create a file named `seed.js` and import the modules required to connect to MongoDB:

```
// seed.js file
import 'dotenv/config';

import { db } from './libs/dbConnect.js';
```

First, you need to populate the database with some users data.

You can create an array of objects to define the users data as follows:

```
const users = [
  {
    username: 'nathan121',
    email: 'nathan@mail.com',
    password: '$2b$10$vD5yRWdxLp1j6riuSi/Ozu71x145viXeGC7AHT5R0WcycGalmYTae',
    avatar:
      'https://g.codewithnathan.com/default-user.png',
    createdAt: new Date().toISOString(),
    updatedAt: new Date().toISOString(),
  },
  {
    username: 'jane78',
    email: 'jane@mail.com',
    password: '$2b$10$vD5yRWdxLp1j6riuSi/Ozu71x145viXeGC7AHT5R0WcycGalmYTae',
    avatar:
      'https://g.codewithnathan.com/default-user.png',
    createdAt: new Date().toISOString(),
    updatedAt: new Date().toISOString(),
  },
];
```

Here, the `password` is the word 'admin' hashed using the bcrypt algorithm.

The `createdAt` and `updatedAt` fields keep track of when the data is created and last updated in ISO string format.

The `avatar` is the profile picture for that user, and the `username` and `email` are details needed for this application.

Next, we also need to create some tasks for the users. Create another array of objects named `tasks` as shown below:

```
const tasks = [
  {
    name: 'Read Atomic Habits',
    description: 'Finish reading Atomic Habits by James Clear',
    priority: 'not urgent',
    due: new Date().toISOString(),
    status: 'open',
    createdAt: new Date().toISOString(),
    updatedAt: new Date().toISOString(),
  },
  {
    name: 'Learn MERN Stack',
    description:
      'Learn the MERN stack and build a full-stack application with it',
    priority: 'urgent',
    due: new Date().toISOString(),
    status: 'open',
    createdAt: new Date().toISOString(),
    updatedAt: new Date().toISOString(),
  },
];
```

Each task object stores information about a task: the task name, description, priority, due date (optional), status, and timestamps when the task is created and last updated.

Now that the data is set, let's try to insert them into the database.

Below the `tasks` array, write the following code:

```
try {
  // Seeding Users
  let collection = db.collection('users');
  console.log('[seed]', 'Seeding Users...');
  const result = await collection.insertMany(users);
  console.log(result.insertedIds);
  console.log('[seed]', 'Seeding Users Done');
```

```
  // Seeding Tasks
  tasks[0].owner = result.insertedIds[0];
  tasks[1].owner = result.insertedIds[1];

  collection = db.collection('tasks');
  console.log('[seed]', 'Seeding Tasks...');
  await collection.insertMany(tasks);
  console.log('[seed]', 'Seeding Tasks Done');

  console.log('[seed]', 'All Done');
} catch (error) {
  console.log('[seed]', 'Error: ', error);
}

process.exit();
```

The code above populates the 'users' collection with the `users` array data. The `inserMany()` method returns the `insertedIds` array as part of the response.

You can then use the `insertedIds` data to specify the `owner` for each task in the `tasks` array, then insert them into the 'tasks' collection.

Once done, call the `process.exit()` method to stop Node.js. You can run this seeding process using `node` as shown below:

```
node seed.js
```

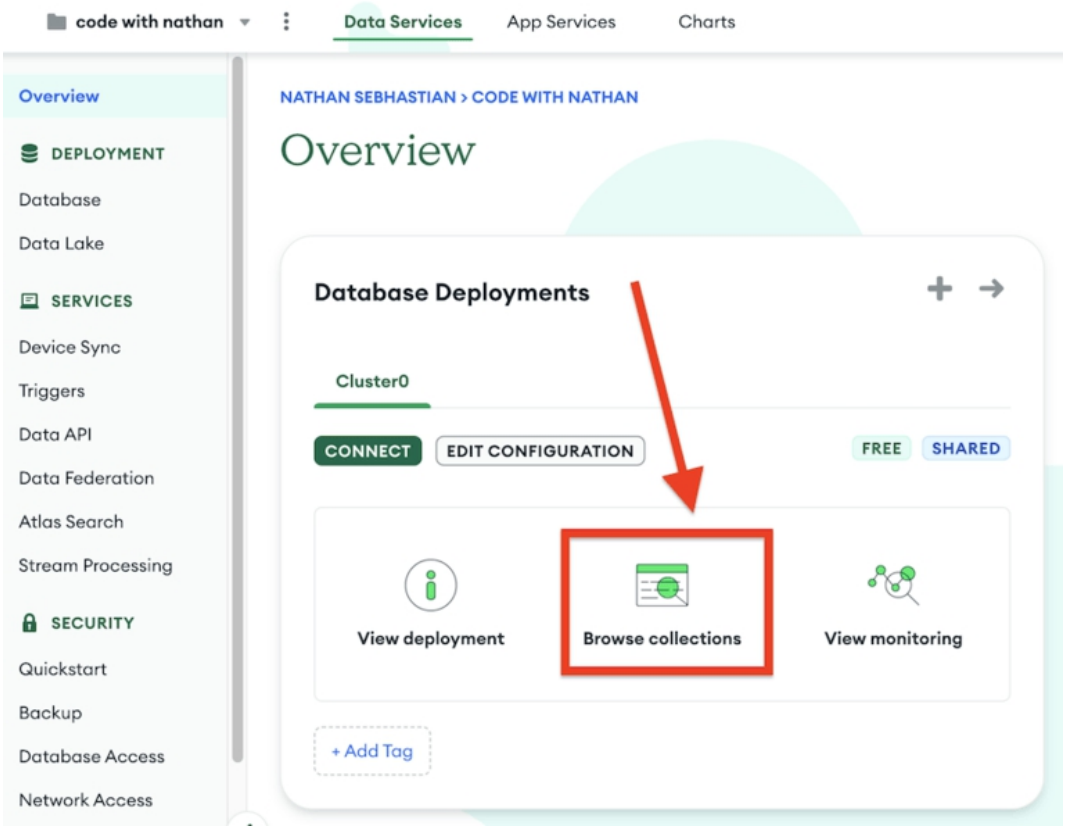
You should see the following output in the terminal:

```
Connected to MongoDB!
[seed] Seeding Users...
{
  '0': new ObjectId('65f0017b804fc42b85675cd4'),
  '1': new ObjectId('65f0017b804fc42b85675cd5')
}
[seed] Seeding Users Done
[seed] Seeding Tasks...
```

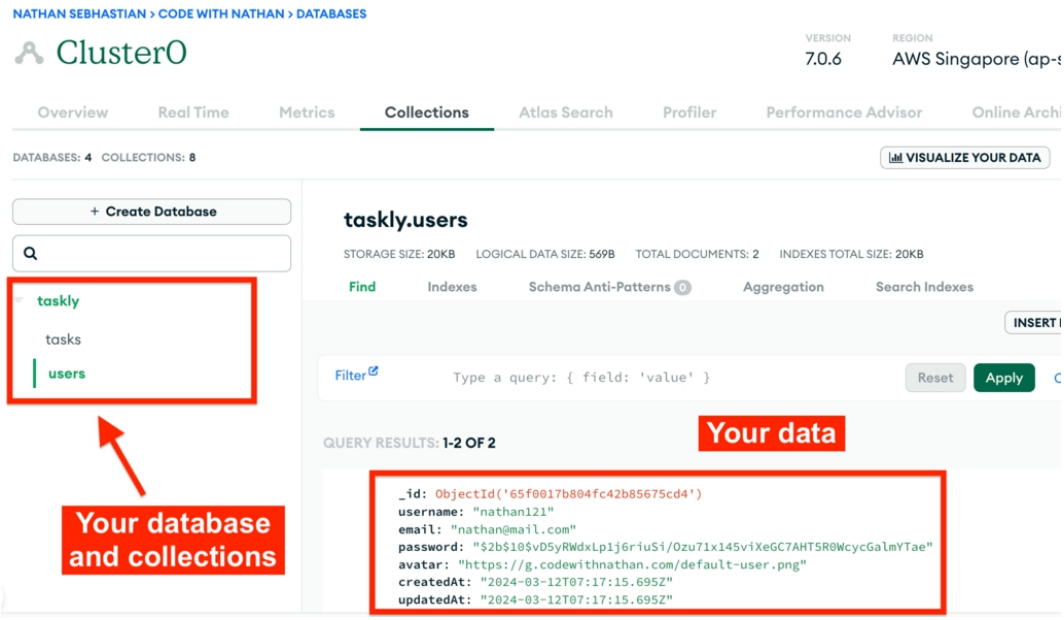
```
[seed] Seeding Tasks Done
[seed] All Done
```

You can see the inserted data from MongoDB cloud as well.

Head over to MongoDB Atlas and click on browse collections on the *Overview* page:



Wait until the page loads, and you should see a database named 'taskly' that contains a 'Users' collection, and in that collection, you'll have a document containing your user data:



Note that MongoDB automatically creates the database and collection when you insert the first document.

Nice work populating the database! In the next chapter, we're going to start writing the Express API to manipulate the user data.

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-4

In this chapter, you've learned how to connect an Express application to a MongoDB Atlas cluster.

You also seed the database with some dummy data that reflects the actual data the application is going to receive.

Seeding the database is a common process when developing a computer application. By using dummy data, you can test the

application properly before deploying it for others to use.

You'll do this often as you develop more web applications in the future.

CHAPTER 5: CREATING USER API ROUTER AND CONTROLLER

Now that we can connect to the database, it's time to create a route so that we can process HTTP requests through Express. Let's handle processing user data first.

The first thing you need to do is to create a route and use it in Express.

Open the `server.js` file, import the user route, and change the `/api` route as follows:

```
import userRouter from './routes/user.route.js';

const PORT = 8000;
const app = express();

app.use('/api/v1/users', userRouter);
```

Here, we change the general `/api` route to a specific `/api/v1/users` route. Requests to the route is handled with the `userRouter` module.

The next step is to create the `userRouter` module. Create a folder named `routes` and then create a new JavaScript file named `user.route.js` as shown below:

```
server
└── routes
    └── user.route.js
```

In that file, write the following code:

```
import express from 'express';

import {
  test,
  getUser,
  updateUser,
  deleteUser,
} from '../controllers/user.controller.js';

const router = express.Router();

router.get('/', test);
router.get('/:id', getUser);
router.patch('/update/:id', updateUser);
router.delete('/delete/:id', deleteUser);

export default router;
```

Here, we use the `express.Router()` function to create a new Router object. With this object, we can define the routes that handle different HTTP request methods sent to Express.

For example, the `router.get('/', test)` code handles any GET requests to the root path '/' and calls the `test` function from the controller.

The `get('/:id', getUser)` router will process GET requests that have the `id` parameter using the `getUser` function.

The `patch('/update/:id', updateUser)` handles the PATCH requests, while `delete('/delete/:id', deleteUser)` handles any DELETE requests.

We need to create the controller file and provide the functions imported in this module next.

Creating the Controller File and Test Function

On the `server/` folder, create a new folder named `controllers/`, then create a new file named `user.controller.js` in it:

```
server
└── controllers
    └── user.controller.js
```

Inside the `user.controller.js` file, import the `db` variable from `dbConnect.js`, then select the `users` collection using the `db.collection()` method.

After that, export a function named `test()` written as follows:

```
import { db } from '../libs/dbConnect.js';

const collection = db.collection('users');

export const test = async (req, res) => {
  let results = await collection.find({}).toArray();
  res.status(200).json(results);
};
```

Here, the `test()` function will handle the HTTP request, so two parameters (`req` and `res`) are defined, just like the functions

defined in `server.js` file.

In this function, we send a query to the MongoDB database using the `find()` method. Because we want to find all users data, an empty object {} is passed as the argument to the method.

We also chain the `toArray()` method next to the `find()` method so that the returned value is converted to an array.

The getUser Function

The next step is to write the `getUser()` function. This function will return the data of a single user back as a response.

First, you need to import the `ObjectId` class from the `mongodb` module:

```
import { ObjectId } from 'mongodb';
```

Then, export a `getUser` function as shown below:

```
export const getUser = async (req, res, next) => {
  try {
    const query = { _id: new ObjectId(req.params.id) };
    const user = await collection.findOne(query);

    if (!user){
      return next({ status: 404, message: 'User not found!' });
    }

    res.status(200).json(user);
  } catch (error) {
    next({ status: 500, error });
  }
};
```

This function will call the `collection.findOne()` method, using the `id` parameter you passed in the URL as the document `_id` value.

The `ObjectId()` function is a constructor that converts the string into an ObjectId object.

When the `user` is found, that user data will be returned as a JSON string. Otherwise, a 404 response will be given when the user is not found.

The `return` statement is added when you call the `next()` function so that Express doesn't execute the code below it.

The process is also wrapped in a `try-catch` block so that any internal error will be handled with a 500 response.

The updateUser Function

The next function we need to create is the `updateUser` function, which uses the `findOneAndUpdate()` method to update the data.

Because you might update the `password` value when updating user data, you need to install the `bcrypt` module to hash the given password. Install the module using npm first:

```
npm install bcrypt
```

Then, add and export the `updateUser()` function as follows:

```
export const updateUser = async (req, res, next) => {
  try {
    if (req.body.password) {
```

```
      req.body.password = await bcrypt.hash(req.body.password, 10);
    }
    const query = { _id: new ObjectId(req.params.id) };
    const data = {
      $set: {
        ...req.body,
        updatedAt: new Date().toISOString(),
      },
    };
    const options = {
      returnDocument: 'after',
    };

    const updatedUser = await collection.findOneAndUpdate(query, data,
options);
    const { password: pass, updatedAt, createdAt, ...rest } = updatedUser;
    res.status(200).json(updatedUser);
  } catch (error) {
    next({ status: 500, error });
  }
};
```

Here, we check if the request `body` object has the `password` property, and hash that password using the `bcrypt.hash()` method if it has.

Next, we define the `query` object and pass the `_id` property like in the `getUser` function.

After that, define the `data` object that contains the new values for the user. To instruct MongoDB to update data, you need to use the `$set` operator and spread the `req.body` object inside it.

The `updatedAt` field is assigned the current datetime in ISO string format.

Next, we set the option object with the `returnDocument` option set to `true`. This way, the method returns the updated user data.

The `updatedUser` are unpacked to protect sensitive data from being returned:

```
const { password: pass, updatedAt, createdAt, ...rest } = user;
```

This ensures that you only send non-sensitive information back to React.

The function process is wrapped in a `try-catch` block to handle any error that might occur.

The deleteUser Function

To delete a user document, you need to call the `deleteOne()` method from the `collection` object.

Here's the code for the `deleteUser()` function:

```
export const deleteUser = async (req, res, next) => {
  try {
    const query = { _id: new ObjectId(req.params.id) };
    await collection.deleteOne(query);
    res.status(200).json({ message: 'User has been deleted!' });
  } catch (error) {
    next({ status: 500, error });
  }
};
```

Here, we define the `query` object for the delete process and then call the `deleteOne()` function.

After deleting the user, we send back a JSON string containing a `message` property.

Adding JSON Middleware

Because our PATCH request requires the new values send in the request `body`, we need to use Express JSON middleware.

In the `server.js` file, add another `app.use()` call after creating the express application:

```
const app = express();

app.use(express.json());
```

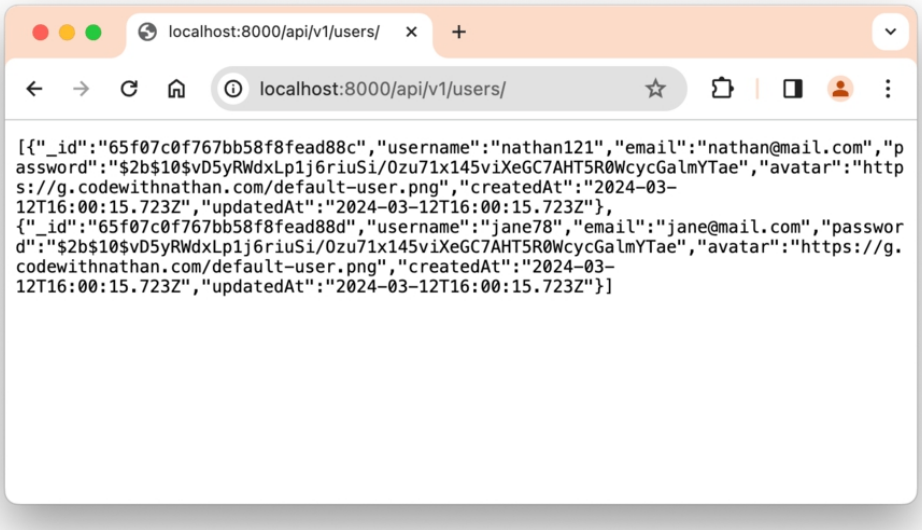
The `express.json()` middleware enables Express to parse incoming JSON data and populate the `req.body` object with that data.

This middleware will be used when we send a POST or PATCH request.

Testing API Routes With Insomnia

Now that we have created routes and functions to manipulate the user data, let's test this API route before moving to the next chapter.

To test GET requests, you can access the URL from the browser as shown below:



But testing POST, PATCH, and DELETE requests using the browser requires you to manually send requests using JavaScript.

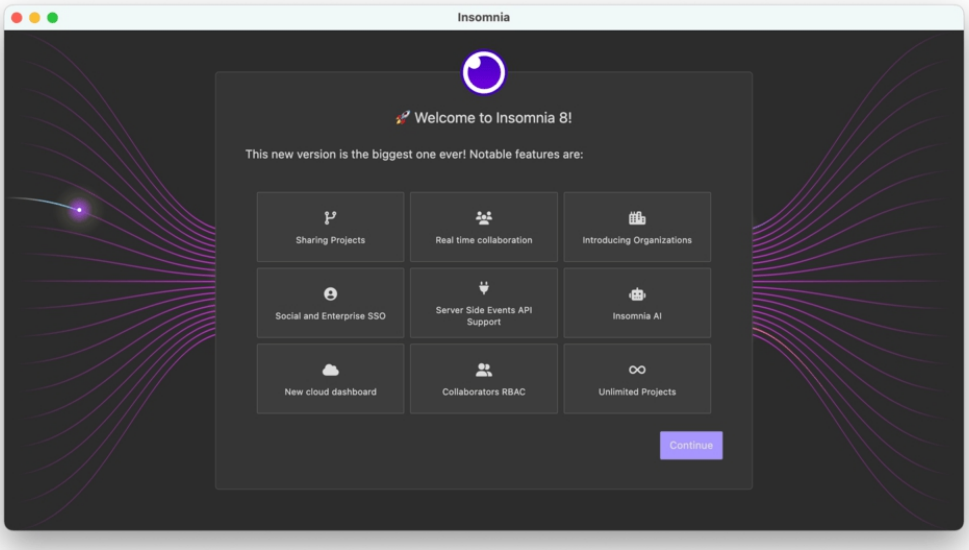
Instead of relying on the browser, let's use a tool called Insomnia.

Insomnia is an open-source API development application that you can use to test API requests.

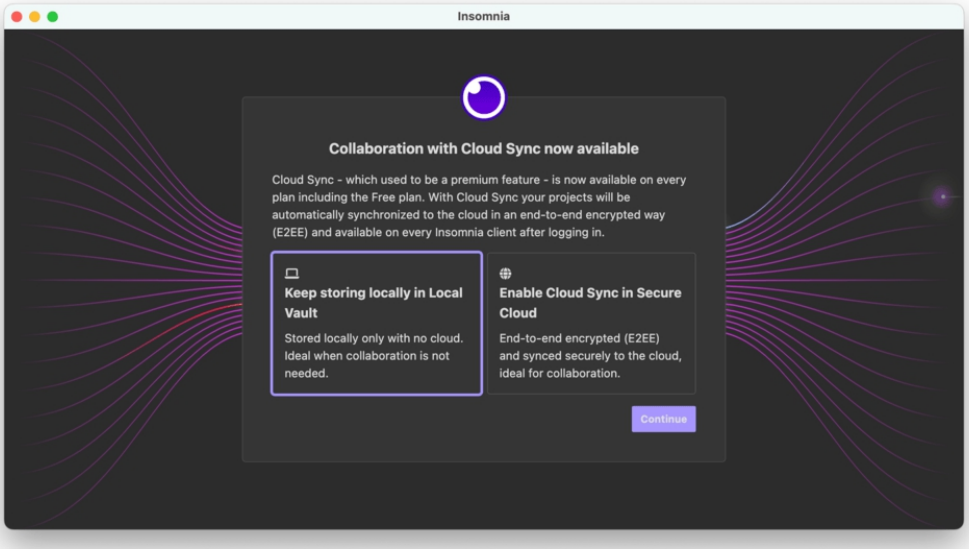
We're not going to explore all features offered by the application, but just enough so that we can easily test our API endpoints.

You can download the application for free at https://insomnia.rest/download

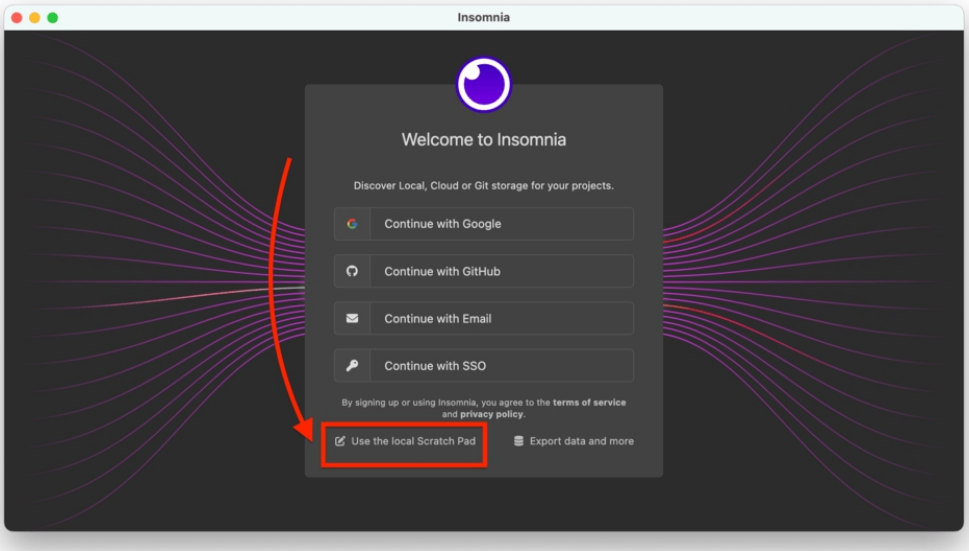
Once the download is complete, install and open the application on your computer. If this is your first time using Insomnia, you'll be greeted with a welcome page:



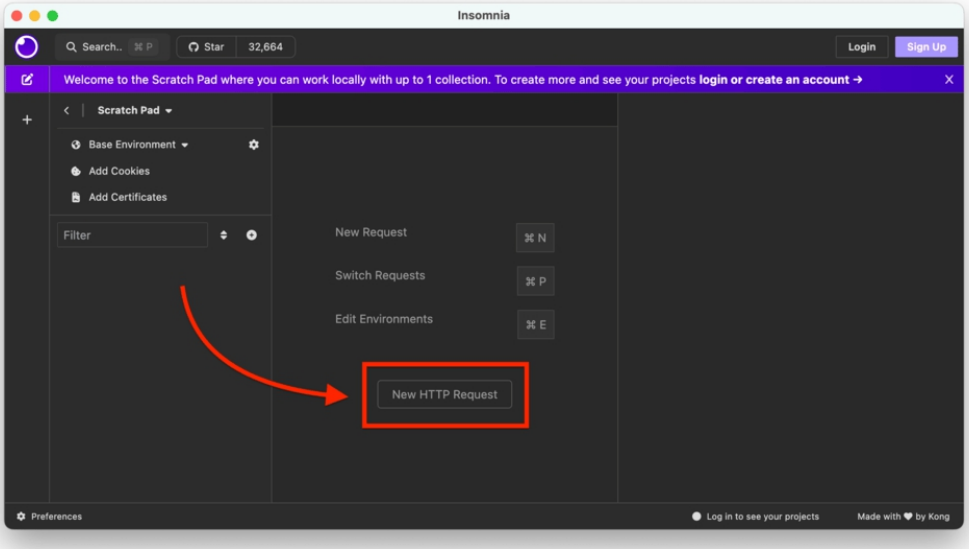
Just click 'Continue', and you'll be asked if you want to enable cloud sync or keep local storage to save your requests:



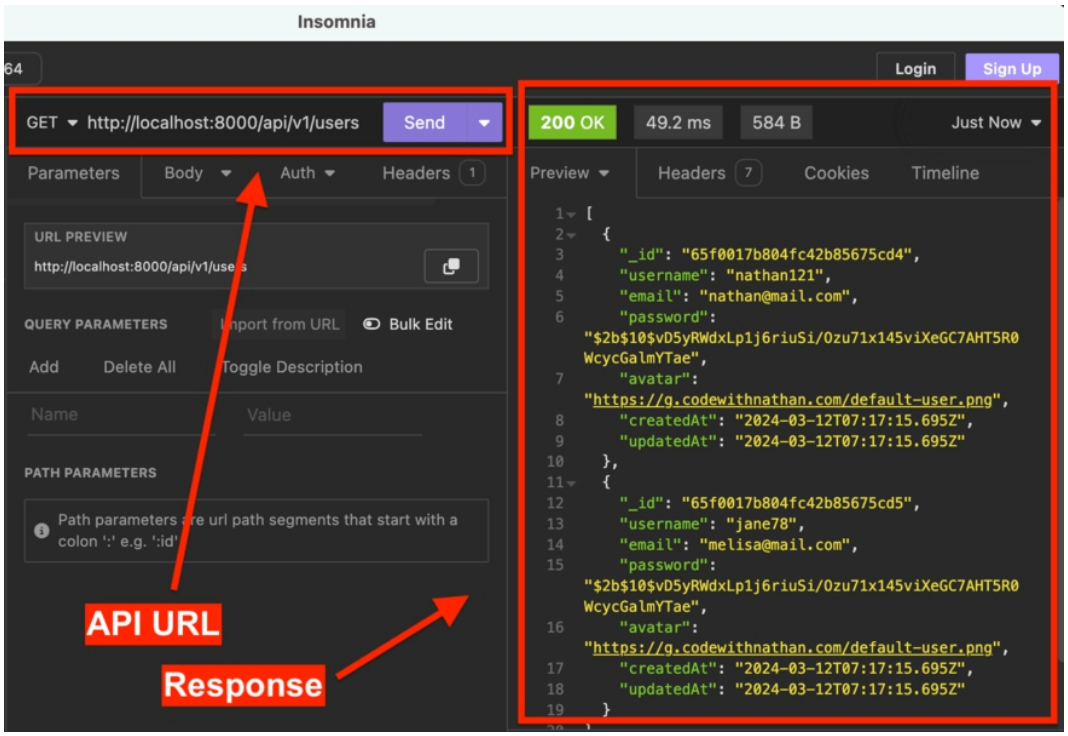
Just use local for now and click 'Continue'. You'll be asked to sign up for an account. Select 'Use the local Scratch Pad' to skip creating an account:



You'll be taken to the main screen of Insomnia. Here, create a new request by clicking on the button below or pressing the shortcut shown:



On the middle screen, input the API URL and select the method. For now, let's test the GET request route at http://localhost:8000/api/v1/users/ as follows:



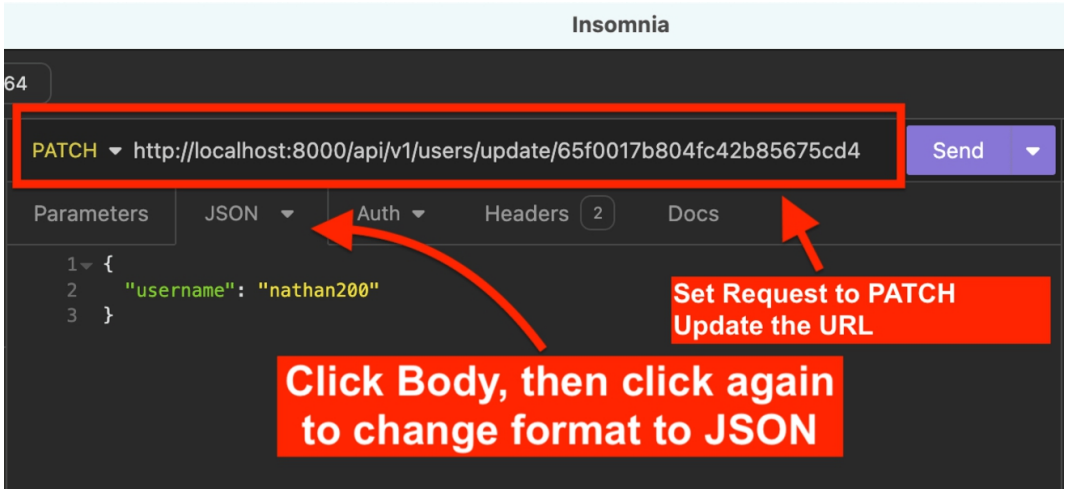
If you see the response as shown above, that means the API is working.

To test getting a single user, you need to pass the user `_id` value in the URL. For example:

```
http://localhost:8000/api/v1/users/65f0017b804fc42b85675cd4
```

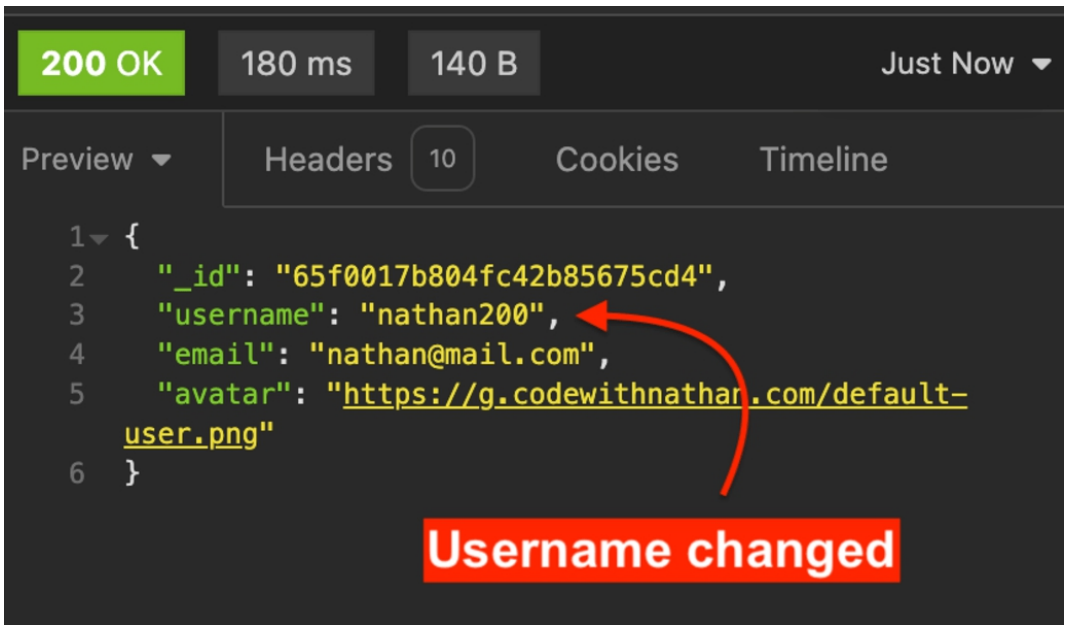
The URL above will get the user 'nathan121' in my database. Your user `_id` would be different because MongoDB generates the `_id` for each document.

To test user update, you need to change the HTTP request method to PATCH and provide the body content in JSON format as shown below:



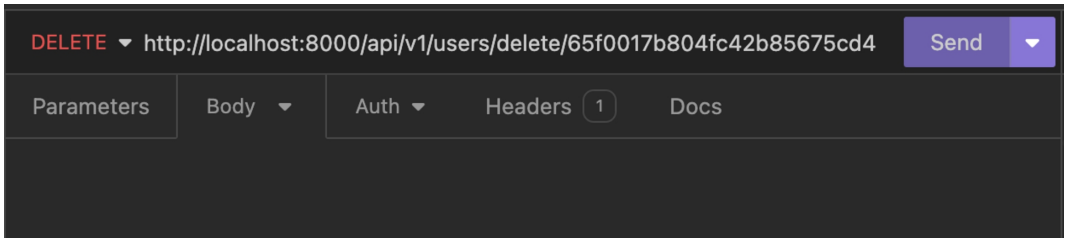
On the request body, add the `username` value because that's the only data we want to change.

Click the 'Send' button and you should see a response as follows:

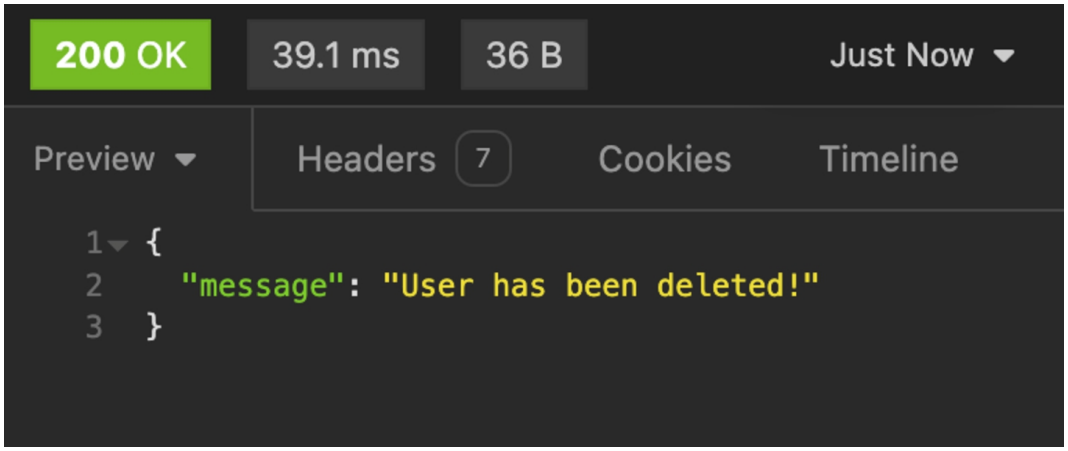


The response shows that the `username` value has been updated successfully.

You can also test the delete user route. Change the HTTP method to DELETE and set Body to No Body:



Send the request, and you will get the following response:



And this means the user route is working. You did a great job!

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-5

In this chapter, you've created the routes and logic used for manipulating the user data, and learn how to test those routes using Insomnia.

In Express, the routes for a specific resource are usually grouped in one file, just like the `user.route.js` created in this

chapter.

A controller file usually holds the logic that handles incoming requests. Add one controller file for each route file you have for a clean separation of the code.

In the next chapter, you're going to add an error handler to Express and do some cleanup.

CHAPTER 6: CREATING ERROR HANDLER IN EXPRESS

In the `user.controller.js` file, we have functions wrapped in a `try-catch` block as follows:

```
export const getUser = async (req, res, next) => {
  try {
    const query = { _id: new ObjectId(req.params.id) };
    const user = await collection.findOne(query);

    if (!user){
      return next({ status: 404, message: 'User not found!' });
    }

    res.status(200).json(user);
  } catch (error) {
    next({ status: 500, error });
  }
};
```

Here, we know that `res.status().json()` is used to send a response back from Express, but we haven't learned what the `next()` function does and why we use it to send an error response.

To explain the `next()` function, we need to learn about Express middleware first. Let's jump in!

Express Middleware Explained

In Express, middlewares are functions that have access to the request, response, and next objects.

These objects are defined as `req`, `res`, and `next` in our functions. Express automatically sends these data when it receives an HTTP request.

An Express application is a series of middleware function calls.

The order of the middlewares is determined by the position of `app.use()` function we defined in our `server.js` file.

This is why you need to place your specific routes above general routes like this:

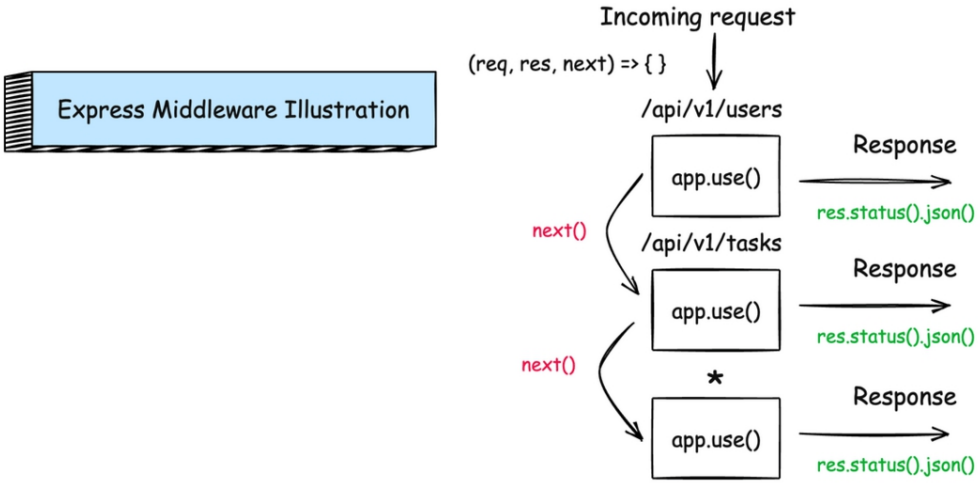
```
app.use('/api/v1/users', userRouter);

app.use('*', (req, res) => {
  res.status(404).json({ message: 'not found' });
});
```

If you switch the route position like this:

```
app.use('*', (req, res) => {
  res.status(404).json({ message: 'not found' });
});

app.use('/api/v1/users', userRouter);
```

Then any request will match the * route, so no request will reach the `/api/v1/users` route.

Here's a visualization of how Express works:



When a request arrives, Express will try to find the middleware that matches the route using a top-down approach.

The `res.status().json()` function will end the running middleware and send back a response.

The `next()` function is used to run the next middleware.

The `(req, res, next)` parameters are passed to the running middleware automatically.

This is why we have the wild card route to handle any URL that doesn't match existing routes. This middleware will be processed last.

How Error Handling Works in Express

To handle errors in Express, you need to write a middleware that accepts four arguments: `err`, `req`, `res`, and `next`, where `err` is the object filled with details of the error.

This middleware needs to be defined last in your middleware list, below the * route.

Let's create an error handler now. In your `libs/` folder, create a new file named `middleware.js` and write the following code:

```
export const errorHandler = (err, req, res, next) => {
  const defaultMessage =
    "We're having technical issues. Please try again later";
  const { status, message, error } = err;
  if (error) {
    console.log(error);
  }
  res.status(status).json({ message: message || defaultMessage });
};
```

Here, the `errorHandler` is a middleware function that has 4 parameters.

The `err` object contains the `status`, `message`, and `error` detail.

To populate the `err` object, you only need to pass an object argument when calling the `next()` function:

```
next({ status: 404, message: 'User not found!' });

next({ status: 500, error });
```

Next, let's use this middleware in your `server.js` file. Place this middleware below the * route:

```
import { errorHandler } from './libs/middleware.js';

// ...

app.use('*', (req, res) => {
  res.status(404).json({ message: 'not found' });
});
```

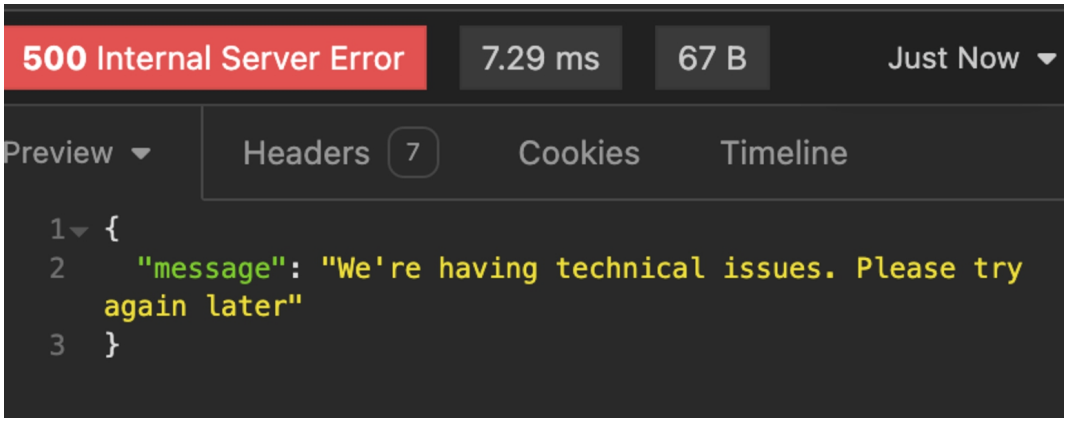
```
app.use(errorHandler);
```

Now the error handler is complete, you can test this middleware by intentionally creating an error.

In the `getUser()` function, remove the `new` operator when creating a new `ObjectId` as follows:

```
const query = { _id: ObjectId(req.params.id) };
```

Now if you try to access the get user API at http://localhost:8000/api/v1/users/:id, you'll get the following response:



On the terminal, you'll see the error stacks logged:

```
TypeError: Class constructor ObjectId cannot be invoked without 'new'
// the rest ...
```

That means the error middleware is running correctly.

Code Cleanup

Before continuing to the next chapter, let's remove unused code from our project to keep it clean.

In the `server.js` file, delete the `db` import statement:

```
import { db } from './libs/dbConnect.js';
```

In the `user.route.js` file, delete the `/` route which calls the `test()` function and the `import` statement:

```
router.get('/', test);
```

Also, delete the test function exported by `user.controller.js` below:

```
export const test = async (req, res) => {
  let results = await collection.find({}).toArray();
  res.status(200).json(results);
};
```

The code above is only used for demonstration purposes. We no longer need it.

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-6

In this chapter, you've learned how Express handles incoming HTTP request through a series of middleware functions, and implement an error handler middleware.

The backend part is starting to take a solid shape, so let's move to the frontend section and start a React project in the next chapter.

CHAPTER 7: GETTING STARTED WITH REACT

The frontend part of our web application will be created using React, so we need to generate a new React application and configure it to connect to our Express backend.

In this chapter, you're going to learn a bit about React and how to create a React application using Vite. Let's dive in.

Introduction to React

React is a very popular JavaScript frontend library that's very popular for its simplicity and fast performance.

React was initially developed by Facebook as a solution to frontend problems they are facing:

- DOM manipulation is an expensive operation and should be minimized
- No library specialized in handling frontend development at the time (there is Angular, but it's an ENTIRE framework.)

- Using a lot of vanilla JavaScript turns a web application into a mess, hard to maintain.

Let me show you an example. Suppose you have a web page where you change the visibility of a `<div>` element by clicking a button. Here's how you do it using vanilla JavaScript:

```
<body>
  <div id="myDiv">
    Hello World!
  </div>
  <button onclick="toggleDiv()">Toggle Div</button>
  <script>
    function toggleDiv() {
      var div = document.querySelector('#myDiv');
      if (div.style.display === 'none') {
        div.style.display = 'block';
      } else {
        div.style.display = 'none';
      }
    }
  </script>
</body>
```

Here, the `toggleDiv()` function is assigned to the `onclick` attribute of the button.

The code inside `toggleDiv()` manipulates the DOM directly by selecting the div using `document.querySelect()` and changing the `display` style based on the current `display` value.

Using React, here's how you can rewrite the same page:

```
import { useState } from 'react';

function App() {
  const [visibility, setVisibility] = useState(true);

  const toggleDiv = () => {
```

```
    setVisibility(!visibility);
  };

  return (
    <div>
      {visibility && (
        <div id='myDiv'>
          <p>Hello World!</p>
        </div>
      )}
      <button onClick={toggleDiv}>Toggle Div</button>
    </div>
  );
}

export default App;
```

Here, we're using the `useState` hook to manage the visibility of the `<div>` element.

The state is initially set to `true` so that the `<div>` is shown on the screen.

When the button is clicked, the `toggleDiv()` function toggles the state value. React then manipulates the DOM and refresh the browser for you.

When using React, you don't manipulate the DOM directly. Instead, you describe how the UI should look like and let React handle the rest.

If you need a solid introduction to React, check out my React book at https://codewithnathan.com/beginning-react

Creating React Application

To create a React Application, open the terminal on the 'taskly' folder you've created earlier and run the following command:

```
npm create vite@5.1.0 client -- --template react
```

The command above will generate a minimal React Application using Vite.

Vite (pronounced 'veet') is a build tool that you can use to bootstrap a new React project.

You should see npm asking to install a new package (create-vite) as shown below. Proceed by typing 'y' and pressing Enter:

```
Need to install the following packages:
  create-vite@5.1.0
Ok to proceed? (y) y
```

Then Vite will create a new React project named 'client' as follows:

```
Scaffolding project in /Users/nsebhastian/dev/taskly/client...

Done. Now run:

  cd client
  npm install
  npm run dev
```

When you're done, follow the next steps you see in the above output.

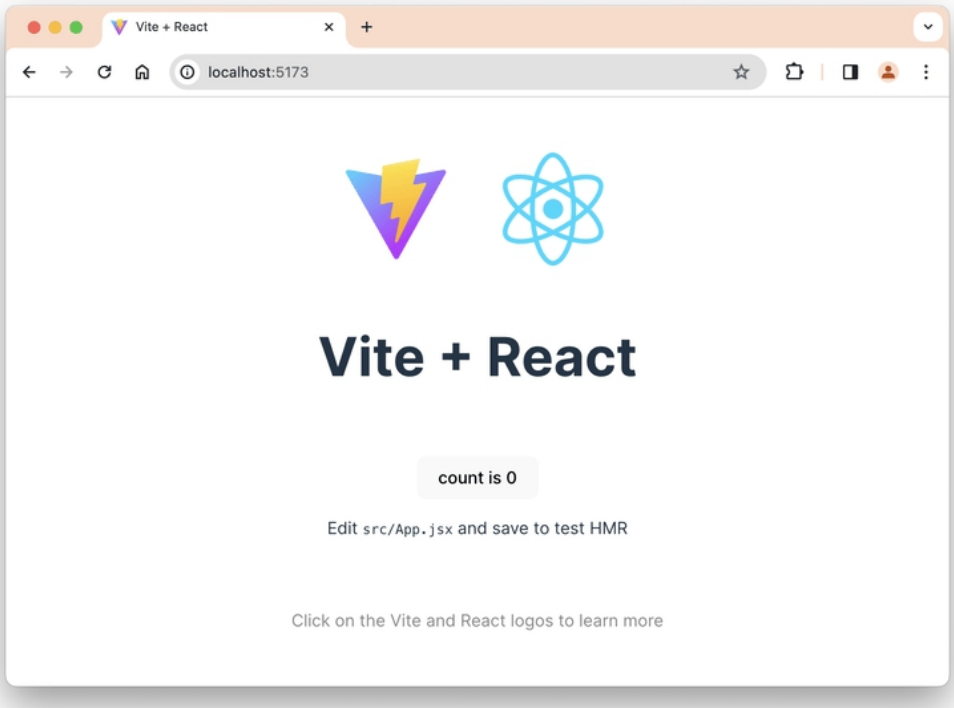
Use the `cd` command to change the working directory to the `client/` folder we've just created, then run `npm install` to

download and install packages required by the application.

Then, we need to run the `npm run dev` command to start our application:

```
$ npm run dev

> client@0.0.0 dev
> vite


  VITE v5.0.10  ready in 509 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

Now you can view the running application from the browser, at the designated localhost address:

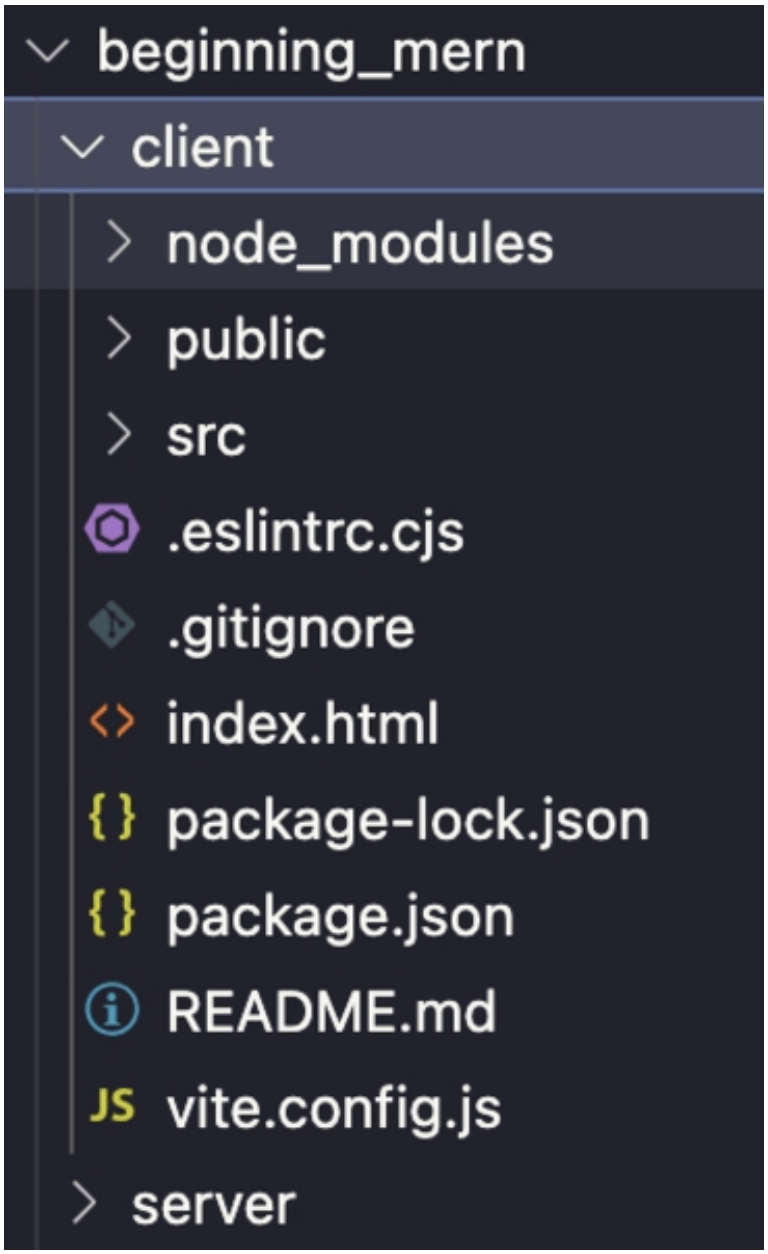


This means you have successfully created your first React app. Congratulations!

Explaining the Source Code

Now that you've successfully run a React application, let's take a look at the source code generated by Vite to understand how things work.

Open the 'client' folder inside VSCode, and you should see several folders and files that make up the React application as follows:



The `vite.config.js` is a configuration file that instructs Vite on how to run the application. Because we have a React application, you'll see the React plugin imported inside:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
```

```
  plugins: [react()],
})
```

When you run the `npm run dev` command, Vite will look into this file to know how to run the program.

The `package.json` file stores the information about the project, including the packages required to run the project without any issues. The `package-lock.json` file keeps track of the installed package versions.

The `.eslintrc.cjs` file contains ESLint rules. ESLint is a code analysis tool that can identify problematic code in your project without needing to run the project. It will report any errors and warnings in VSCode.

The `index.html` file is a static HTML document that's going to be used when running the React application, and the `README.md` file contains an introduction to the project.

You don't need to modify any of these files. Instead, let's go to the `src/` folder where the React application code is written.

```
src
├── App.css
├── App.jsx
├── assets
│   └── react.svg
├── index.css
└── main.jsx
```

First, the `App.css` file contains the CSS rules applied to the `App.jsx` file, which is the main React application code.

The `assets/` folder contains the assets required for this project. In this case, it's the React icon, which you had seen in the browser.

The `index.css` file is the root CSS file applied globally to the application, and the `main.jsx` file is the root file that access the `index.html` file to render the React application.

Here's the content of `main.jsx` file:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

Here, you can see that the ReactDOM library creates a root at the `<div>` element that contains the `root` ID, and then renders the whole React application to that element.

You can open the `App.jsx` file to view the React code:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
```

```
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

In this file, a single component named `App` is defined.

The Vite and React logos are rendered with a link to the respective library, and there's a counter button that will increment the counter by 1 when you click on it.

Here's where you can test showing and hiding a `<div>` element by clicking a button.

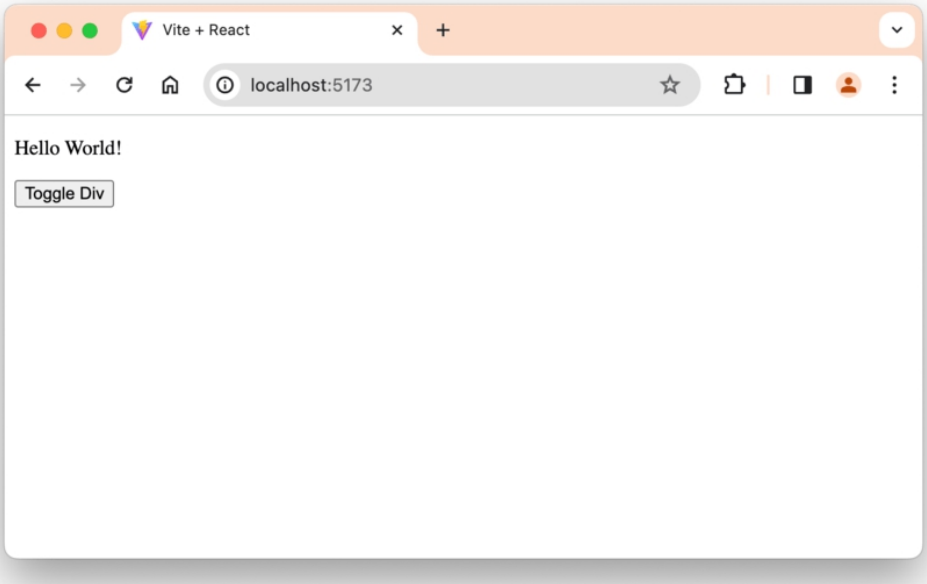
Replace the code in `App.jsx` with the following:

```
import { useState } from 'react';

function App() {
  const [visibility, setVisibility] = useState(true);

  const toggleDiv = () => {
    setVisibility(!visibility);
```

```
  };

  return (
    <div>
      {visibility && (
        <div id='myDiv'>
          <p>Hello World!</p>
        </div>
      )}
      <button onClick={toggleDiv}>Toggle Div</button>
    </div>
  );
}

export default App;
```

Next, delete the `index.css`, `app.css`, `public/vite.svg`, and the `assets/` folder. You also need to delete the `import './index.css'` statement in your `main.jsx` file.

If you open the browser again, you should see the div and button rendered as follows:



Well done! Now we're ready to develop the frontend project. Let's start our first lesson in the next chapter.

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-7

In this chapter, we've explored the origins of React and how it compares to vanilla JavaScript. We've also learned how to generate a minimal React application using Vite.

Finally, we went through the source code generated by Vite to know how the application works. If you encounter any issues, you can email me at nathan@codewithnathan.com and I will do my best to help you.

CHAPTER 8: SUPERCHARGE REACT WITH SUPPORTING LIBRARIES

To create an interactive web application with an awesome user experience, you need to install supporting React libraries.

We’re going to use the following libraries in our project:

- React Router for routing and creating multi-page application
- Chakra UI for building user interfaces
- React Hook Form for building forms
- React Hot Toast for showing toast notifications

There are alternatives for each of these libraries, but after years of developing with React, I found them to be the easiest to work with.

You can also build these functionalities yourself, but it's going to take days or months to write the code for these libraries.

All these libraries also have good documentation, so whenever you need help, you can go to their websites.

Adding React Router

React Router is a popular library for creating and managing routes in a React application.

Routing is the process of pointing a specific browser URL to a specific React component you want to show for that URL.

It's like routing in Express, where a specific URL points to a certain task, only this time you show different user interfaces.

To start using React Router, you need to install the library using npm:

```
npm install react-router-dom
```

Once the library is installed, create a folder named `pages/` inside the `src/` folder, then create three files named `Home.jsx`, `SignIn.jsx` and `SignUp.jsx` as follows:

```
src
├── App.jsx
├── main.jsx
└── pages
    ├── Home.jsx
    ├── SignIn.jsx
    └── SignUp.jsx
```

On these three files, write a simple component that returns a <h1> element as follows:

```
// Home.jsx
export default function Home() {
    return <h1>Home</h1>
}

// SignIn.jsx
export default function SignIn() {
    return <h1>Sign In</h1>
}

// SignUp.jsx
export default function SignUp() {
    return <h1>Sign Up</h1>
}
```

Next, open the `App.jsx` file and replace the code inside the file with the one below:

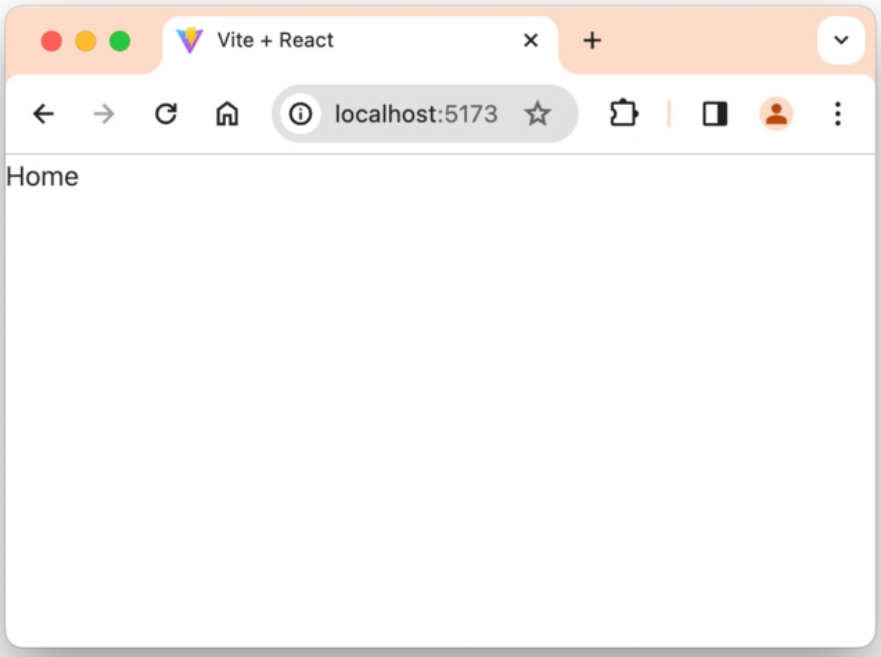
```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

export default function App() {
  return (
      <BrowserRouter>
        <Routes>
          <Route path='/' element={<Home />} />
          <Route path='/signin' element={<SignIn />} />
          <Route path='/signup' element={<SignUp />} />
        </Routes>
      </BrowserRouter>
  );
}
```

Here, the `BrowserRouter`, `Routes`, and `Route` components are used to initialize React Router.

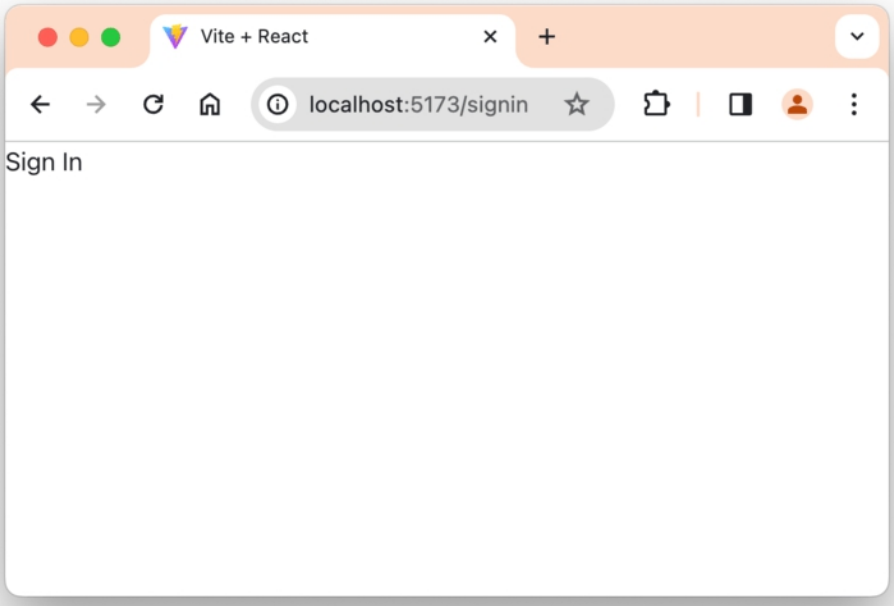
The `BrowserRouter` component handles managing the browser's address bar (the URL you see at the top).

The `Routes` component is used to group `Route` components together, and the `Route` component is where you match the URL `path` with the `element` to render.

Save the changes, then open your browser. You should see the Home page rendered as follows:



You can navigate to `/signin` or `/signup` by changing the URL manually. See the `<SignIn>` or `<SignUp>` component rendered as follows:



And that's how React Router works in essence. You're going to use it more in the next chapter.

But for now, let's install another library you need to use, which is Chakra UI.

Adding Chakra UI

Chakra UI is a library of pre-made React components that you can use to make user interfaces.

The components provided by Chakra UI can be used as building blocks for your React application. To start using the library, install the package first:

```
npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion
```

Next, open the `App.jsx` file import the `<ChakraProvider>` component from `chakra-ui` as follows:

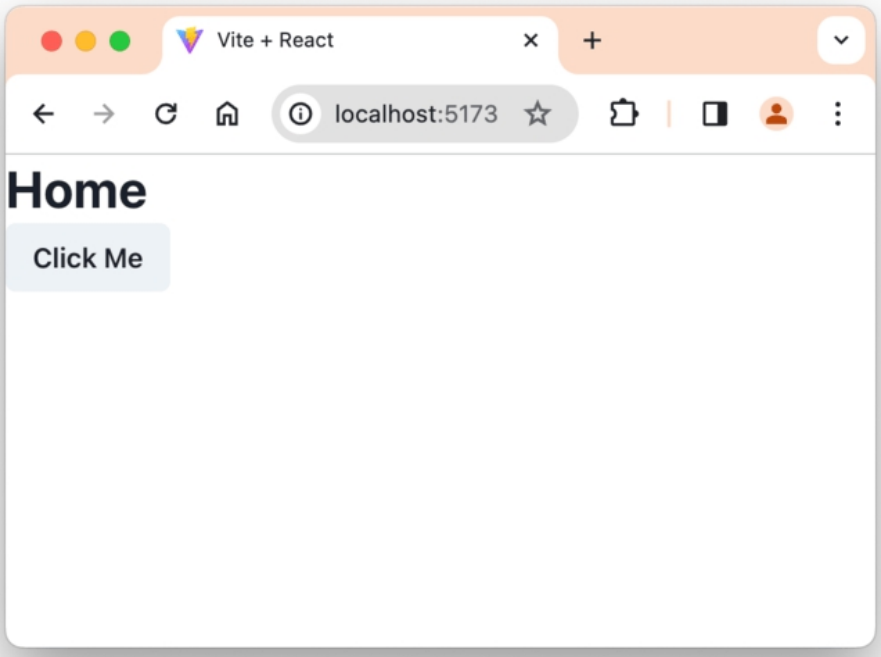
```
import { ChakraProvider } from '@chakra-ui/react'

export default function App() {
  return (
    <ChakraProvider>
      <BrowserRouter>
        <Routes>
          <Route path='/' element={<Home />} />
          <Route path='/signin' element={<SignIn />} />
          <Route path='/signup' element={<SignUp />} />
        </Routes>
      </BrowserRouter>
    </ChakraProvider>
  );
}
```

Now any component below `<ChakraProvider>` will be able to use Chakra UI's components.

Let's test the library. Open the `Home.jsx` file, then import and use the `Box`, `Heading`, and `Button` components from Chakra UI as follows:

```
import { Box, Heading, Button } from '@chakra-ui/react';

export default function Home() {
  return (
    <Box>
      <Heading>Home</Heading>
      <Button>Click Me</Button>
    </Box>
  );
}
```

You should see the following output on the browser:

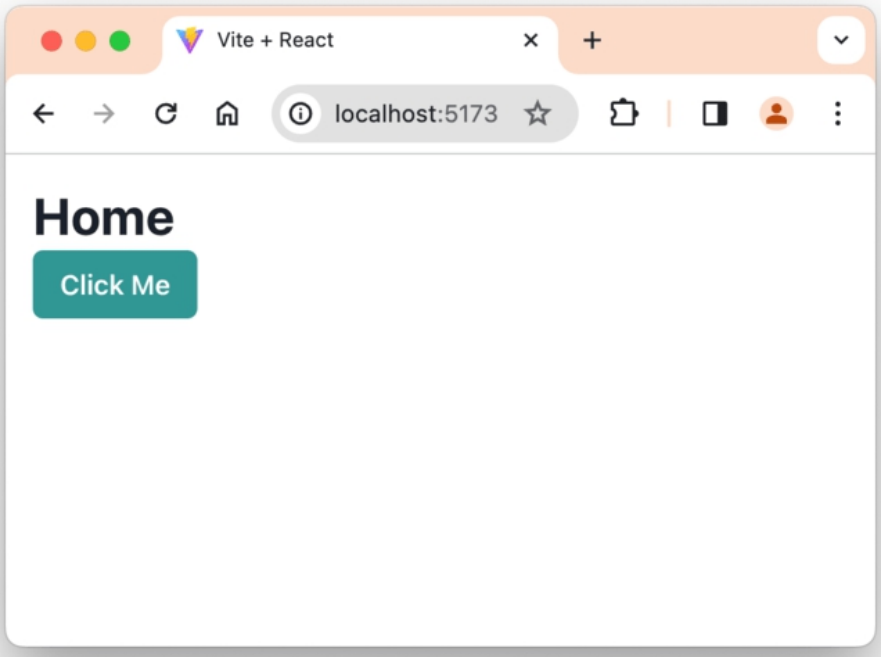


To customize the components, you can pass CSS style as props.

The code below shows how to add padding around <Box> and change the <Button> color:

```
export default function Home() {
  return (
    <Box p='4'>
      <Heading>Home</Heading>
      <Button colorScheme='teal'>Click Me</Button>
    </Box>
  );
}
```

The components above will be updated according to the given props:



To see the full list of components supported by Chakra UI, you can check out the documentation at https://chakra-ui.com/docs/components

To see all available style props, check out https://chakra-ui.com/docs/styled-system/style-props

We will explore Chakra UI more later. For now, let's install another helpful library, which is React Hook Form.

Adding React Hook Form

React Hook Form is a form builder library that aims to reduce the pain of creating forms with React. You can use this library to handle three cumbersome parts of building a form:

1. Getting values in and out of form state
2. Validation and error messages
3. Handling form submission

To get started with React Hook Form, you need to install the library using npm:

```
npm install react-hook-form
```

Let's test this library by creating a form on the Sign Up page next.

Creating the Sign Up Form

Open the `SignUp.jsx` file and import the `useForm` hook, which contains all the functionalities available in React Hook Form:

```
// SignUp.jsx
import { useForm } from 'react-hook-form';
```

Then, call the `useForm()` hook from inside the component and unpack the functionalities needed to build a form as follows:

```
export default function SignUp() {
  const {
    handleSubmit,
    register,
    formState: { errors, isSubmitting },
  } = useForm();
}
```

Here, we take the `handleSubmit()` function to run the submission process. The `register()` function is used to register an input to

the library.

The `formState` contains information about the form state, such as storing `errors` from failed validation and whether the form is currently submitting.

There are more variables and functions in this library, but this would be enough for the sign up form.

Now you need to import components from Chakra UI and React Router as follows:

```
import { Link } from 'react-router-dom';
import {
  FormControl,
  Input,
  Button,
  Text,
  Box,
  Flex,
  Heading,
  Stack,
  FormErrorMessage
} from '@chakra-ui/react';
```

The `Link` component from React Router is used to change the URL of the browser.

Next, put down the layout of the Sign Up page as follows:

```
// Inside SignUp Component:
return (
  <Box p='3' maxW='lg' mx='auto'>
    <Heading
      as='h1'
      textAlign='center'
      fontSize='3xl'
      fontWeight='semibold'
      my='7'
    >
```

```
      Create an Account
    </Heading>
    <form>
      {/* form details... */}
    </form>
    <Flex gap='2' mt='5'>
      <Text>Have an account?</Text>
      <Link to={'/signin'}>
        <Text as='span' color='blue.400'>
          Sign in
        </Text>
      </Link>
    </Flex>
  </Box>
);
```

Here, we use the `Box` component to determine the max width and margin for the element. Then we have a `Heading` and `form` element.

Below the form, we have a `Flex` component that contains a text and link to the Sign In page.

Inside the `form` element, add the `username` input as shown below:

```
<Stack gap='4'>
  <FormControl isInvalid={errors.username}>
    <Input
      id='username'
      type='text'
      placeholder='username'
      {...register('username', { required: 'Username is required' })}
    />
    <FormErrorMessage>
      {errors.username && errors.username.message}
    </FormErrorMessage>
  </FormControl>
</Stack>
```

The `Stack` component is used to group elements together. The space between each element is specified by the `gap` style prop.

To create an input form with Chakra UI, you need to use the `FormControl` element for controlling the input. The `isInvalid` prop will show a red border around the input when it's `true`.

Next, the `Input` component is used to create a text input. This component is registered in React Hook Form by calling the `register()` function.

The `register()` function has two parameters: the input name (`String`) and the register options (`Object`)

Because the username must not be empty, the `required` option is passed.

Below the `Input`, the `FormErrorMessage` is used to display any error message.

The sign up form will have three inputs, so add these below the username:

```
<FormControl isInvalid={errors.email}>
  <Input
    id='email'
    type='email'
    placeholder='email'
    {...register('email', { required: 'Email is required' })}
  />
  <FormErrorMessage>
    {errors.email && errors.email.message}
  </FormErrorMessage>
</FormControl>
<FormControl isInvalid={errors.password}>
  <Input
    id='password'
    type='password'
    placeholder='password'
    {...register('password', { required: 'Password is required' })}
  />
```

```
  <FormErrorMessage>
    {errors.password && errors.password.message}
  </FormErrorMessage>
</FormControl>
<Button
  type='submit'
  isLoading={isSubmitting}
  colorScheme='teal'
  textTransform='uppercase'
>
  Sign Up
</Button>
```

On the code above, two more text inputs are added for email and password. After that, a `Button` is added for submitting the form.

For the last step, you need to create a function that will process the form submission. Write the `doSubmit()` function just above the `return` statement as follows:

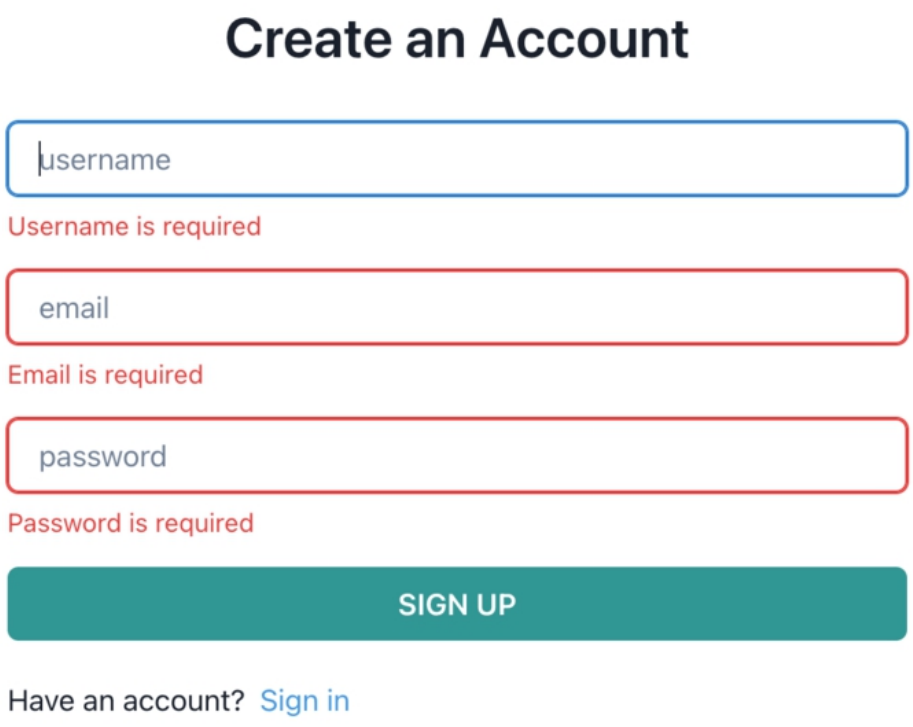
```
const doSubmit = async values => {
  alert('Sign Up Successful. You are now logged in');
};
```

Finally, add the `onSubmit` attribute to the `form` element, which calls the `handleSubmit()` function from React Hook Form, and pass the `doSubmit` function as its argument:

```
<form onSubmit={handleSubmit(doSubmit)}>
</form>
```

Alright! Now you can test the finished form on the browser.

If you click the Submit button without entering any value, you will see error messages like this:



Fill in the inputs and click the Submit button again. You'll see an alert shown on the page.

Adding React Hot Toast

React Hot Toast is a library used to create a toast notification with minimal code. You can see the notification demo at https://react-hot-toast.com

This toast notification can be used to show the response of the form submission process, such as successfully registering a new account or creating a new task.

Install the library using npm as usual:

```
npm install react-hot-toast
```

In the `App.jsx` file, import the `Toaster` component and place it inside `BrowserRouter` as follows:

```
import { Toaster } from 'react-hot-toast';

// ...

<BrowserRouter>
  <Toaster position='bottom-right' />
  <Routes>
    <Route path='/' element={<Home />} />
    <Route path='/signin' element={<SignIn />} />
    <Route path='/signup' element={<SignUp />} />
  </Routes>
</BrowserRouter>
```

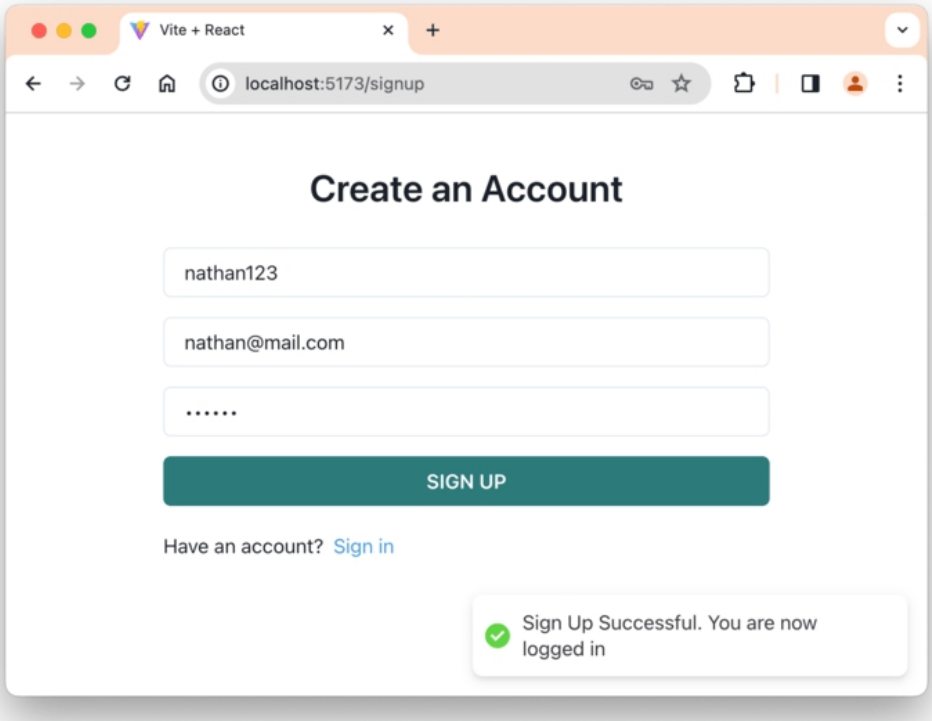
The `Toaster` component will show the toast notification at the designated `position`.

The next step is to call the `toast()` function anytime you want to show a toast.

Back on the `SignUp.jsx` file, replace the `alert()` function with `toast.success()` when the form is submitted:

```
import toast from 'react-hot-toast';

// ...

const doSubmit = async values => {
  toast.success('Sign Up Successful. You are now logged in');
};
```

Now when you submit the form, you will see the toast notification as follows:



All the libraries required to develop a React application are now installed. Let's take a break before we continue developing the frontend part.

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-8

The rich ecosystem of libraries and tools that has been created around React is one of the main benefits of using this library.

Many developers around the world have created packages that enable certain functionalities in React.

In this chapter, we've installed and tested just 4 of the many libraries available for free. We will install a few more in the coming chapters.

CHAPTER 9: AUTHENTICATION WITH JSON WEB TOKEN

With the frontend part taking shape, your next task is to implement the sign up process.

In web applications, a sign up process involves authenticating the user by creating a JSON Web Token, or JWT for short.

Once Express successfully authenticates the user, it will generate a JWT containing some information about the user (could be username or id value) and then encrypt that information using a secret key.

The JWT is sent back to the client and stored in the browser as a cookie. Anytime you send a request to a route that requires authentication, Express must check on the cookie included on that request.

The request only continues when JWT is verified. Otherwise, Express must not allow that request.

Let's jump in and see how to implement authentication to your MERN application.

Sending a Network Request

In the previous chapter, you've created a sign up form for people to register an account.

Let's continue developing that form by sending a network request to the Express server.

First, create a `.env` file inside the `client/` folder and add the base URL to the server as follows:

```
VITE_API_BASE_URL=http://localhost:8000/api/v1
```

This variable will be the base URL whenever you send a request to the server.

Next, create a file named `util.js` inside the `client/src/` folder, and import the environment variable as follows:

```
export const API_BASE_URL = import.meta.env.VITE_API_BASE_URL;
```

The `import.meta.env` is used by Vite to expose environment variables. Only variables prefixed with `VITE_` are exposed.

Okay, now you can continue developing the sign up form.

Import the `API_BASE_URL` constant, then use the `fetch()` function to send a POST request inside the `doSubmit()` function:

```
import { API_BASE_URL } from '../util.js';

// Inside SignUp component...

const doSubmit = async values => {
  try {
```

```
    const res = await fetch(`${API_BASE_URL}/auth/signup`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(values),
    });
    const data = await res.json();
    if (res.status === 200) {
      toast.success('Sign Up Successful. You are now logged in');
    } else {
      toast.error(data.message);
    }
  } catch (error) {
    toast.error('Something went wrong');
  }
};
```

Here, a POST request is sent using `fetch()` to the http://localhost:8000/api/v1/auth/signup URL. The form `values` are assigned to the `body` of the request as a JSON string.

When the request is successful, show a success notification. Otherwise, show an error notification.

Now you need to add an `auth/signup` route in Express.

Express Sign Up Route

Before creating the sign up route, let's install the library used for authentication in Express.

On the `server/` folder, run the following install command:

```
npm install cookie-parser jsonwebtoken
```

The `cookie-parser` library is used to parse cookies received by Express, while `jsonwebtoken` is used to create a JWT that Express will send back on successful authentication.

Now open the `server.js` file and add the `/auth` route as follows:

```
// Import the router
import authRouter from './routes/auth.route.js';

// Create the route
app.use('/api/v1/auth', authRouter);
```

On the `server/routes/` folder, create a new file named `auth.route.js` and write the following code:

```
import express from 'express';
import {
  signup,
} from '../controllers/auth.controller.js';

const router = express.Router();

router.post('/signup', signup);

export default router;
```

Here, the `/auth/signup` route is defined. Let's create the `auth.controller.js` file next and import the libraries required for this controller:

```
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';
import { db } from '../libs/dbConnect.js';
```

Next, select the 'users' collection from the `db` object:

```
const collection = db.collection('users');
```

Then, export the `signup()` function as shown below:

```
export const signup = async (req, res, next) => {
  try {
    const { username, email, password } = req.body;
    const query = {
      $or: [{ email }, { username }],
    };
    const existingUser = await collection.findOne(query);
    if (existingUser) {
      return next({
        status: 422,
        message: 'Email or Username is already registered.',
      });
    }
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = {
      username,
      email,
      password: hashedPassword,
      avatar: 'https://g.codewithnathan.com/default-user.png',
      createdAt: new Date().toISOString(),
      updatedAt: new Date().toISOString(),
    };

  } catch (error) {
    next({ status: 500, error });
  }
}
```

In the code above, `username`, `password`, and `email` values are unpacked from the `req.body` object.

The `email` and `username` values are used to find an existing user with the `findOne()` method.

If there's an existing user, a 422 error is sent back. Otherwise, the `password` is hashed and the `user` data is organized for saving in MongoDB.

Next, run the `collection.insertOne()` method as follows:

```
const { insertedId } = await collection.insertOne(user);
const token = jwt.sign({ id: insertedId }, process.env.AUTH_SECRET);
user._id = insertedId;
const { password: pass, updatedAt, createdAt, ...rest } = user;
res
  .cookie('taskly_token', token, { httpOnly: true })
  .status(200)
  .json(rest);
```

The `insertOne()` method will attempt to save the `user` data in MongoDB 'users' collection.

When successful, the method returns the `insertedId` variable which contains the new user `_id` value.

After that, a JWT `token` is created by calling the `jwt.sign()` method and passing the `id` object as the data stored on the token.

There's also an `AUTH_SECRET` variable that `jwt` will use as the signing key. We're going to add the key after this.

The `insertedId` is added to the `user` object, and then we unpack values of `user` to separate `password`, `createdAt`, and `updatedAt` data from the rest:

```
const { password: pass, updatedAt, createdAt, ...rest } = user;
```

The `rest` object now contains data that's relevant and non-sensitive for the frontend.

After that, Exporess sends a response to the client. The `res.cookie()` method sends a cookie back to the client, while `json()` sends JSON data.

Adding AUTH_SECRET Environment Variable

The `jwt.sign()` method requires an `AUTH_SECRET` environment variable, so let's add it.

The `AUTH_SECRET` is an alphanumeric string used only in Express to sign the JWT. You can generate one by using the `openssl` command.

Run the following command from the terminal:

```
openssl rand -base64 32
```

This will generate a string that you can use as the value of `AUTH_SECRET` and put it in the `.env` file.

If your terminal doesn't have `openssl` installed, you can use the one I provide below:

```
AUTH_SECRET=9X8f/hXA6eHQ7W9aHjtzdKJRkMvVSYFG0KRyBVJU1rU=
```

That will take care of the JWT signing key.

Enabling Cookie Parser Middleware

To enable the installed `cookie-parser` middleware, you need to use this middleware in your `server.js` file:

```
import cookieParser from 'cookie-parser';
// ...
app.use(express.json());
app.use(cookieParser());
```

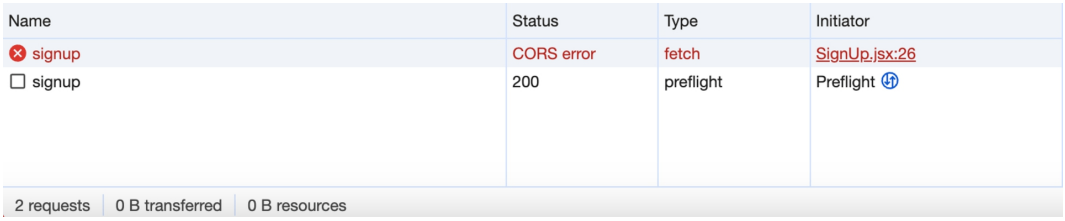
That's all you need to do. Express will store incoming cookies under the `req.cookies` object.

Enabling CORS Option

Now that the `/auth/signup` route is complete, you can test the route from the client.

Fill in the form and press the submit button. An error toast saying 'Something went wrong' is shown.

If you open the browser's console, you'll see an error as follows:



CORS stands for Cross-Origin Resource Sharing, and it's a server configuration that controls whether we can send a request to a different origin or not.

An example of a cross-origin request is when a website at `domain-a.com` sends a request to `domain-b.com`

Currently, we have the React application served at `localhost:5173` and Express served at `localhost:8000`, so any `fetch()` request is effectively a cross-origin request.

Express doesn't allow a CORS request by default, so you need to enable it using the `cors` middleware.

Install the middleware using npm:

```
npm install cors
```

Then import and use the middleware in the `server.js` file as follows:

```
import cors from 'cors';

app.use(
  cors({
    origin: process.env.CLIENT_URL,
    credentials: true,
  })
);
```

Here, the `origin` option is used to define the origin allowed to access Express API routes.

The `credentials` option enables Express to accept cookies that you'll send from React later.

Enter Vite local address for the `CLIENT_URL` as follows:

```
CLIENT_URL=http://localhost:5173
```

Now you should be able to register a new user successfully. Nice work!

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-9

In this chapter, you've learned how to code the sign up process in Express.

You receive the form data from React, check if the email is already registered, encrypt the password, and save it to MongoDB.

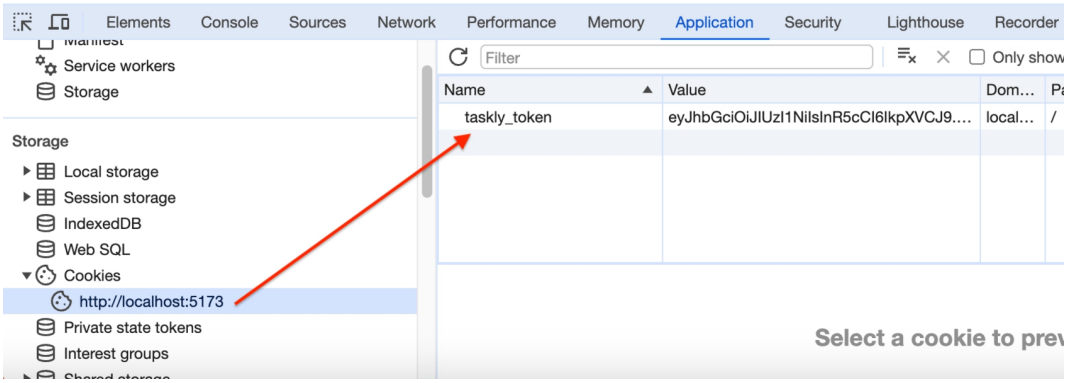
After that, a JWT is signed using the `AUTH_SECRET` key and sent back as a cookie to the browser.

In the next chapter, you'll see how to save the user data in local storage and synchronize that value with React state.

CHAPTER 10: SAVE USER DATA WITH CONTEXT API

The sign up process that we've implemented in the previous chapter sends a cookie and a JSON string back to the browser.

The cookie will automatically be saved in the browser, and you can check it by opening the developer console on the 'Application' tab:



You'll see how to include this cookie when sending a request to Express later.

For now, let's see how to save the JSON response in React using the Context API and local storage.

The Context API

In React, components manage their own state. This means the state data within a component is confined to that specific component.

To let other components use the state data, you need to pass it as a prop. The following example shows how to pass data from Parent Component state to Child Component:

```
export default function Parent(){
  const [user, setUser] = useState();

  return (
    <div>
      <Child user={user} />
    </div>
  )
}
```

But there are times when you need to share state or data between components that are not directly related in terms of parent-child relationships.

React Context is the solution provided by React for this case. It allows you to create a global state that can be accessed by any component that requires the data.

Let's see how to create a Context for the `user` data received from Express.

Create a new folder named `context` and add a `UserContext.jsx` file inside it. Write the following code:

```
import { createContext, useContext, useState } from 'react';
```

```
const UserContext = createContext();
```

The `createContext()` function creates a context object that holds the context value.

Next, create the provider for this context, named as `userProvider` as follows:

```
const UserProvider = props => {
  const [user, setUser] = useState(null);

  const updateUser = user => {
    setUser(user);
  };

  const value = {
    user,
    updateUser,
  };

  return (
    <UserContext.Provider value={value}>{props.children}
</UserContext.Provider>
  );
};
```

In `UserProvider()`, the `user` state is initialized as `null`, and the `updateUser()` function is defined as a way to update the `user` value.

The `UserContext.Provider` component allows any component that sits below it to access the context value. You're going to import this component in `App.jsx` later.

The `value` prop in the `Provider` component is the values that will be accessible by any component that consumes the context.

Below the `UserProvider()` function, write a `useUser()` hook as follows:

```
const useUser = () => {
  const context = useContext(UserContext);
  if (context === undefined) {
    throw new Error('useUser must be used within a UserProvider');
  }
  return context;
};
```

This hook will call the `useContext()` hook and return the `context` value to the caller. Any component that wants to consume the context only needs to call this hook.

Now you need to export both `UserProvider` and `useUser` from the file:

```
export { UserProvider, useUser };
```

For the next step, add the `UserProvider` component in `App.jsx` file as follows:

```
import { UserProvider } from './context/UserContext';

export default function App() {
  return (
    <UserProvider>
      <ChakraProvider>
        <BrowserRouter>
          {/* ... */}
        </BrowserRouter>
      </ChakraProvider>
    </UserProvider>
  );
}
```

Now any component inside `UserProvider` can access the context.

Protecting Routes in React Router

When developing a web application, you'll need to protect routes from unauthorized access.

React Router doesn't provide a way to protect routes, so you need to write one yourself.

Don't worry, though! Protecting a route is easy. You only need to create a React component that checks on the Context API and see if user data exists.

First, create a new folder named `components/` and add the `PrivateRoute.jsx` file in that folder:

```
import { Outlet, Navigate } from 'react-router-dom';
import { useUser } from '../context/UserContext';

export default function PrivateRoute() {
  const { user } = useUser();
  return user ? <Outlet /> : <Navigate to='/signin' />;
}
```

This component checks whether the `user` state in `UserContext` has a truthy value.

If the `user` exists, the `Outlet` component will render the child component. Otherwise, use the `Navigate` component to render the Sign In page.

To test the route protection, let's create another page named `Profile.jsx` in the `pages/` folder:

```
export default function Profile(){
  return <h1>Profile Page</h1>
```

```
}
```

Now add this page to `App.jsx` file and wrap it inside `PrivateRoute` as follows:

```
import Profile from './pages/Profile';

import PrivateRoute from './components/PrivateRoute';

// Inside App() component:
<Routes>
  <Route path='/' element={<Home />} />
  <Route path='/signin' element={<SignIn />} />
  <Route path='/signup' element={<SignUp />} />

  <Route element={<PrivateRoute />}>
    <Route path='/profile' element={<Profile />} />
  </Route>
</Routes>
```

Here, you can see that the `/profile` route was placed as the child of the `PrivateRoute`.

Anytime you access the `/profile` URL, the `PrivateRoute()` function will be executed.

If you try to access the Profile page now, you will always be redirected to the Sign In page because the `user` context is `null`.

To enable access, call the `useUser()` hook from `SignUp.jsx` and update the user on successful registration:

```
import { useUser } from '../context/UserContext.jsx';

export default function SignUp() {
  const { updateUser } = useUser();

  //... Inside doSubmit function:
  if (res.status === 200) {
```

```
    toast.success('Sign Up Successful. You are now logged in');
    updateUser(data);
  } else {
    toast.error(data.message);
  }
}
```

Now when you register a new user, the JSON data received from Express will be set as the value of the `user` context.

By the way, let's also navigate to the Profile page automatically on success. You need to use the `useNavigate` hook:

```
import { Link, useNavigate } from 'react-router-dom';

export default function SignUp() {
  const navigate = useNavigate();

  //... Inside doSubmit function:
  if (res.status === 200) {
    toast.success('Sign Up Successful. You are now logged in');
    updateUser(data);
    useNavigate('/profile')
  } else {
    toast.error(data.message);
  }
}
```

The sign up process is now perfect. You did a great job here!

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-10

In this chapter, you've learned how to save authenticated user data in React Context API.

The Context API can then be used to protect routes from unauthenticated users by calling the `PrivateRoute()` function.

CHAPTER 11: ADDING A NAVIGATION BAR

With the sign up process completed in both React and Express, let's continue by adding a navigation bar before adding the sign in and sign out process.

Creating the Navigation Bar

The Navigation Bar will show the user's profile picture and show links that change depending on the value of the `user` context.

First, create a `NavBar.jsx` file inside the `components/` folder and import the required modules:

```
import { Link as RouterLink, useNavigate } from 'react-router-dom';
import toast from 'react-hot-toast';
import { useUser } from '../context/UserContext';
import { API_BASE_URL } from '../util.js';
import {
  Flex,
  Box,
  Spacer,
  Link,
  Menu,
  MenuButton,
  MenuList,
```

```
  MenuItem,
  Image,
} from '@chakra-ui/react';
```

Since the `Link` components from both React Router and Chakra UI are imported, we need to give an alias to one of them to avoid any error.

The `Link` component from React Router is renamed as `RouterLink` here.

Next, create the `NavBar()` component as shown below:

```
export default function NavBar() {
  const { user, updateUser } = useUser();
  const navigate = useNavigate();

  const handleSignOut = async () => {
    try {
      const res = await fetch(`${API_BASE_URL}/auth/signout`, {
        credentials: 'include',
      });
      const message = await res.json();
      toast.success(message);
      updateUser(null);
      navigate('/');
    } catch (error) {
      toast.error(error);
    }
  };
}
```

A `handleSignOut()` function is created in the component to handle sign out process.

Notice that there's a `credentials: 'include'` header added to the request. This is used to include the cookie saved in the browser on the request.

We're going to implement the sign out route in Express later. For now, let's continue by adding the `return` statement:

```
return (
  <Box as='nav' bg='red.50'>
    <Flex
      minWidth='max-content'
      alignItems='center'
      p='12px'
      maxW='7xl'
      m='0 auto'
    >
      <Box p='2'>
        <Link as={RouterLink} fontSize='lg' fontWeight='bold' to='/'>
          Taskly
        </Link>
      </Box>
      <Spacer />
      <Box>
        {user ? (
          <Menu>
            <MenuButton>
              <Image
                boxSize='40px'
                borderRadius='full'
                src={user.avatar}
                alt={user.username}
              />
            </MenuButton>
            <MenuList>
              <MenuItem as={RouterLink} to='/profile'>
                Profile
              </MenuItem>
              <MenuItem as={RouterLink} to='/tasks'>
                Tasks
              </MenuItem>
              <MenuItem onClick={handleSignOut}>Sign Out</MenuItem>
            </MenuList>
          </Menu>
        ) : (
          <Link as={RouterLink} to='/signin'>
            Sign In
          </Link>
        )}
      </Box>
    </Flex>
```

```
  </Box>
);
```

When the profile picture is clicked, a menu to navigate to other pages will be shown.

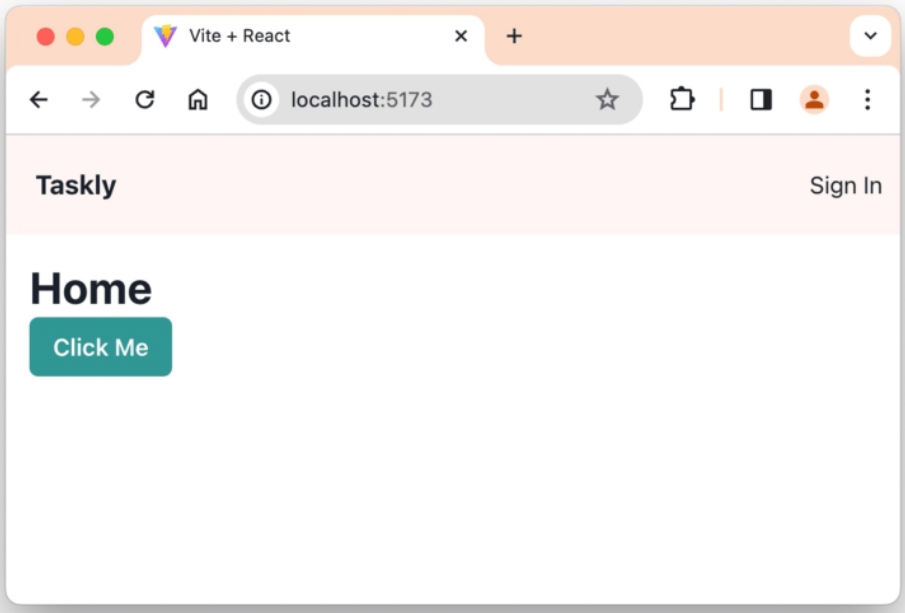
When the user is not authenticated, a link to the Sign In page is shown.

Rendering the Navigation Bar

Import the `Navbar` component in `App.jsx` file to render it:

```
import NavBar from './components/NavBar';

export default function App() {
  return (
    <UserProvider>
      <ChakraProvider>
        <BrowserRouter>
          <NavBar />
          {/* ... */}
        </BrowserRouter>
      </ChakraProvider>
    </UserProvider>
  );
}
```

Now you should see the navigation bar rendered as follows:



Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-11

In this chapter, we've added a navigation bar that renders a different interface depending on the value of the context API.

This navigation bar also has the sign out functionality, which we need to implement in the next chapter.

CHAPTER 12: IMPLEMENT SIGN IN AND SIGN OUT PAGES

With the navigation bar in place, we can start implementing the sign in and sign out functionalities.

We'll start with implementing the routes in Express, and then use those routes in React.

Creating the /signup and /signout Routes in Express

Create the `/signup` and `/signout` routes in `auth.route.js` as follows:

```
import {
  signup,
  signin,
  signOut,
} from '../controllers/auth.controller.js';

const router = express.Router();

router.post('/signup', signup);
router.post('/signin', signin);
router.get('/signout', signOut);
```

Then, open the `auth.controller.js` file and create the functions for these routes.

The `signin()` middleware will take the data in `req.body` and use the `email` to find a valid user:

```
export const signin = async (req, res, next) => {
  const { email, password } = req.body;
  try {
    const validUser = await collection.findOne({ email });
    if (!validUser) {
      return next({ status: 404, message: 'User not found!' });
    }
    const validPassword = await bcrypt.compare(password, validUser.password);
    if (!validPassword) {
      return next({ status: 401, message: 'Wrong password!' });
    }
    const token = jwt.sign({ id: validUser._id }, process.env.AUTH_SECRET);
    const { password: pass, updatedAt, createdAt, ...rest } = validUser;
    res
      .cookie('taskly_token', token, { httpOnly: true })
      .status(200)
      .json(rest);
  } catch (error) {
    next({ status: 500, error });
  }
};
```

When a `validUser` exists, the process continuew to validate the password using `bcrypt.compare()` method.

When the `password` is valid, a JWT is signed and sent as a response, similar to the sign up process.

The next step is to add the `signOut()` function. This function only needs to clear the cookie as follows:

```
export const signOut = async (req, res, next) => {
  try {
```

```
    res.clearCookie('taskly_token');
    res.status(200).json({ message: 'Sign out successful' });
  } catch (error) {
    next({ status: 500 });
  }
};
```

The `res.clearCookie()` function will send an instruction to the client (browser) to clear the stored cookie.

Implementing the SignIn Page in React

The sign out process was already implemented in the navigation bar before, so we only need to implement the Sign In page now.

Open the `client/pages/SignIn.jsx` file, and import the modules:

```
import { useForm } from 'react-hook-form';
import { Link, useNavigate } from 'react-router-dom';
import {
    FormControl,
    Input,
    Button,
    Text,
    Box,
    Flex,
    Heading,
    Stack,
    FormErrorMessage,
} from '@chakra-ui/react';
import toast from 'react-hot-toast';
import { API_BASE_URL } from '../util.js';
import { useUser } from '../context/UserContext.jsx';
```

These modules are similar to the one you use on the `SignUp.jsx` file.

Inside the `SignIn()` function, initialize the hooks:

```
export default function SignIn() {
  const navigate = useNavigate();
  const { updateUser } = useUser();

  const {
    handleSubmit,
    register,
    formState: { errors, isSubmitting },
  } = useForm();
}
```

Then write the `doSubmit()` function as shown below:

```
const doSubmit = async values => {
  try {
    const res = await fetch(`${API_BASE_URL}/auth/signin`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      credentials: 'include',
      body: JSON.stringify(values),
    });
    const data = await res.json();
    if (res.status === 200) {
      toast.success('Sign In Successful');
      updateUser(data);
      navigate('/profile');
    } else {
      toast.error(data.message);
    }
  } catch (error) {
    console.log(error);
    toast.error('Something went wrong');
  }
};
```

The `doSubmit()` function performs a POST request to the `/auth/signin` route, sending the form `values` on the request body.

If Express responds with `200` HTTP code, a success toast notification will be shown, the user context data will be updated, and the user will be directed to the Profile page.

Now you need to write the `return` statement and create the form as follows:

```
return (
  <Box p='3' maxW='lg' mx='auto'>
    <Heading
      as='h1'
      textAlign='center'
      fontSize='3xl'
      fontWeight='semibold'
      my='7'
    >
      Enter Your Credentials
    </Heading>
    <form onSubmit={handleSubmit(doSubmit)}>
      <Stack gap='4'>
        <FormControl isInvalid={errors.email}>
          <Input
            id='email'
            type='email'
            placeholder='email'
            {...register('email', { required: 'Email is required' })}
          />
          <FormErrorMessage>
            {errors.email && errors.email.message}
          </FormErrorMessage>
        </FormControl>
        <FormControl isInvalid={errors.password}>
          <Input
            id='password'
            type='password'
            placeholder='password'
            {...register('password', { required: 'Password is required' })}
          />
          <FormErrorMessage>
            {errors.password && errors.password.message}
          </FormErrorMessage>
        </FormControl>
        <Button
          type='submit'
```

```
            isLoading={isSubmitting}
            colorScheme='teal'
            textTransform='uppercase'
          >
            Sign In
          </Button>
        </Stack>
      </form>
      <Flex gap='2' mt='5'>
        <Text>Dont have an account?</Text>
        <Link to={'/signup'}>
          <Text as='span' color='blue.400'>
            Sign up
          </Text>
        </Link>
      </Flex>
    </Box>
  );
```

This form is very similar to the sign up form, only this one has two text inputs instead of three.

And that's it. Now you can try to sign in using existing user.

Persisting User Data in Local Storage

If you try to sign in and then refresh the page, you'll see that the user is signed out of Taskly.

This is because the user context is temporary. Back in the `UserContext.jsx` file, you can see that the default value of the `user` state is `null`:

```
const [user, setUser] = useState(null);
```

Instead of always initializing with `null`, the `user` state should be saved on the browser.

When we leave and return to the application, the `user` state can then be retrieved from the browser

The local storage is a persistent storage that's included in the browser. By using JavaScript, you can store data as key/value pair in the local storage.

Let's create a hook that syncs the `user` state on the local storage.

In the `util.js` file, create and export a `useLocalStorage` hook as shown below:

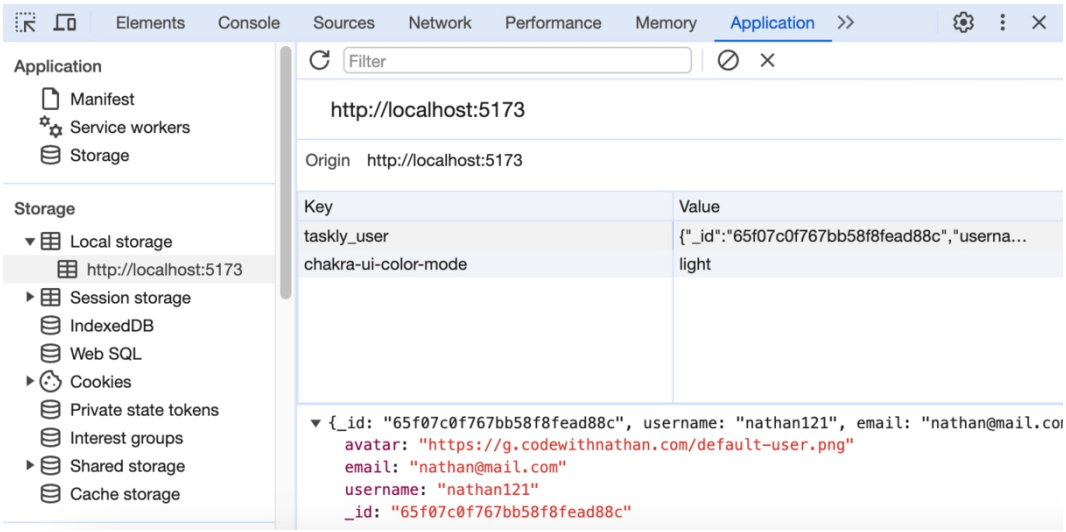
```
import { useState, useEffect } from "react";

export const useLocalStorage = (key, defaultValue) => {
  const [value, setValue] = useState(() => {
    const currentValue = localStorage.getItem(key);

    if (currentValue) {
      return JSON.parse(currentValue)
    }
    else {
      return defaultValue;
    }
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [value]);

  return [value, setValue];
};
```

This function will take two arguments, the local storage `key` and the `defaultValue` used when the `key` doesn't exist in local storage.

Now you can use this hook when initializing the `user` state:

```
import { createContext, useContext } from 'react';
import { useLocalStorage } from '../util.js';

const UserProvider = props => {
  const [user, setUser] = useLocalStorage('taskly_user', null);

  //...
};
```

If you try to sign in again, you'll see the local storage data now changed on the browser console:



When you sign out, the local storage will be updated to `null` and you'll be directed to the Home page.

Finishing the Home Page

Before we wrap up this chapter, let's update the Home page to make it look good.

Here, you only need to show some text, a link, and an image as follows:

```
import { Link as RouterLink } from 'react-router-dom';
import productiveSvg from '/productive.svg';
import {
  Box,
  Heading,
  Text,
  Link,
  Flex,
  Stack,
  Image,
} from '@chakra-ui/react';

export default function Home() {
  return (
    <Flex direction={{base: 'column', md: 'row'}} gap={6} p={14} maxW='6-xl'
mx='auto'>
      <Stack flex='1' alignSelf='center' >
        <Heading
          as='h1'
          fontSize={{ base: '4xl', lg: '6xl' }}
          fontWeight='bold'
          color='gray.700'
        >
          Make your {' '}
          <Text as='span' color='gray.500'>
            perfect
          </Text>
          <br />
          day
        </Heading>
        <Box color='gray.500' pt='8'>
          Taskly will help you manage your day and to-do list.
          <br />
          We have a wide range of features to help you get things done.
        </Box>
        <Link
          as={RouterLink}
          to={'/profile'}
          fontWeight='bold'
          color='blue.400'
        >
          Let's get started...
        </Link>
      </Stack>
      <Box flex='1'>
        <Image src={productiveSvg} maxWidth='400px' alt='Productive
Illustration' />
```

```
      </Box>
    </Flex>
  );
}
```

You can get the `productive.svg` file at https://g.codewithnathan.com/productive.svg and place it inside the `public/` folder.

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-12

In this chapter, you've implemented the sign in and sign out routes in Express, and then completed the sign in form in React.

You've also persisted the authenticated user data by syncing the Context API value and the local storage.

So far so good!

CHAPTER 13: CREATING USER PROFILE PAGE

In the Profile page we created earlier, we're going to show the user data. They can change the username, email, password, and profile picture.

In React, we've secured private routes by checking for the user data from the user context API, which is synchronized with the local storage.

In Express, we need to secure the user routes as well by verifying the JWT in each request. This can be done by creating a middleware.

We're also going to add links to create a new task, show tasks, and delete the user account.

Most of the code in this chapter has already been used previously, so it will seem familiar to you.

Express verifyToken() Middleware

To secure the API routes we've created in Express, we need to check on the JWT cookie whenever we want to process a

request.

On the `libs/middleware.js` file, add a new middleware named `verifyToken()` as follows:

```
import jwt from 'jsonwebtoken';

export const verifyToken = (req, res, next) => {
  const token = req.cookies.taskly_token;

  if (!token) return next({ status: 401, message: 'Unauthorized' });

  jwt.verify(token, process.env.AUTH_SECRET, (err, user) => {
    if (err) return next({ status: 403, message: 'Forbidden' });
    req.user = user;
    next();
  });
};
```

Since we've added the `cookieParser()` middleware before, any cookies sent to Express will be included in the `req.cookies` object.

The `verifyToken()` middleware simply checks on the content of the `taskly_token` object parsed by `cookieParser()`.

If the `token` doesn't exist, we reject the request by sending back an 'Unauthorized' response:

```
if (!token) return next({ status: 401, message: 'Unauthorized' });
```

If the cookie exists, call the `jwt.verify()` function to verify the token and decrypt the content.

When the verification process fails, return a response that says 'Forbidden'. Otherwise, attach the `user` object decrypted from

the JWT to the `req.user` object.

The next middleware that handles the response can use this object to perform further validation.

Adding verifyToken() to User Routes

You need to use the `verifyToken()` middleware in `user.route.js` file as follows:

```
import { verifyToken } from '../libs/middleware.js';

const router = express.Router();

router.get('/:id', verifyToken, getUser);
router.patch('/update/:id', verifyToken, updateUser);
router.delete('/delete/:id', verifyToken, deleteUser);
```

The middleware will protect these routes from requests that doesn't have JWT included.

Next, you need to open the `user.controller.js` file and check if the `user.id` equals the `params.id` in `updateUser()` and `deleteUser()` functions:

```
// updateUser()
export const updateUser = async (req, res, next) => {
  if (req.user.id !== req.params.id) {
    return next({
      status: 401,
      message: 'You can only update your own account',
    });
  }
  try {
    // ...
  }
};

// deleteUser()
```

```
export const deleteUser = async (req, res, next) => {
  if (req.user.id !== req.params.id) {
    return next({
      status: 401,
      message: 'You can only delete your own account',
    });
  }
  try {
    // ...
  }
};
```

These `if` statements will ensure that the users are manipulating their own accounts.

The Express changes are completed. Let's move to the frontend and make changes to the React application.

Adding New Pages

Before updating the Profile page, let's add the pages needed in our React application. We need pages for:

- creating a task
- updating a task
- showing all tasks
- showing a single task

Some of these routes will be connected to the Profile page as well, so create them in `src/pages/` folder as follows:

```
src/pages
├── CreateTask.jsx  <<<
├── Home.jsx
├── Profile.jsx
├── SignIn.jsx
├── SignUp.jsx
├── SingleTask.jsx  <<<
├── Tasks.jsx       <<<
└── UpdateTask.jsx  <<<
```

For the content, just add a `<h1>` element for each page:

```
// CreateTask.jsx
export default function CreateTask() {
  return <h1>Create Task</h1>;
}

// SingleTask.jsx
export default function SingleTask() {
  return <h1>Single Task</h1>;
}

// Tasks.jsx
export default function Tasks() {
  return <h1>Tasks</h1>;
}

// UpdateTask.jsx
export default function UpdateTask() {
  return <h1>Update Task</h1>;
}
```

We'll work more on these pages later on. Add these pages to React Router in `App.jsx` file:

```
import CreateTask from './pages/CreateTask';
import UpdateTask from './pages/UpdateTask';
import Tasks from './pages/Tasks';
import SingleTask from './pages/SingleTask';

// Inside App() function:

<Route element={<PrivateRoute />}>
  <Route path='/profile' element={<Profile />} />
  <Route path='/create-task' element={<CreateTask />} />
  <Route path='/update-task/:taskId' element={<UpdateTask />} />
  <Route path='/tasks' element={<Tasks />} />
  <Route path='/tasks/:taskId' element={<SingleTask />} />
</Route>
```

Now the new pages can be accessed from the URLs defined above.

Updating the Profile Page

Now it's time to show the user data on the Profile Page. Open the `Profiles.jsx` file and import the libraries we're going to use:

```
import { useForm } from 'react-hook-form';
import { Link as RouterLink, useNavigate } from 'react-router-dom';
import { useUser } from '../context/UserContext.jsx';
import { API_BASE_URL } from '../util.js';
import toast from 'react-hot-toast';
import {
  Box,
  Heading,
  Center,
  Image,
  Input,
  Stack,
  FormControl,
  Button,
  Link,
  Flex,
  Text,
  FormErrorMessage,
} from '@chakra-ui/react';
```

These imports are familiar since we've used them when creating the sign up and sign in forms.

The next step is to initialize values in the `Profile` component. Notice there's a `defaultValues` property defined when calling the `useForm()` hook:

```
export default function Profile() {
  const navigate = useNavigate();
  const { user, updateUser } = useUser();

  const {
    register,
    handleSubmit,
    resetField,
    formState: { errors, isSubmitting },
  } = useForm({
    defaultValues: {
      avatar: user.avatar,
      username: user.username,
      email: user.email,
    },
  });
}
```

The `defaultValues` property will populate the form with default values for the form fields.

Below the `useForm()` call, you need to create a function that does the form submission process:

```
const doSubmit = async values => {
  try {
    const res = await fetch(`${API_BASE_URL}/users/update/${user._id}`, {
      method: 'PATCH',
      credentials: 'include',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(values),
```

```
    });
    const data = await res.json();
    if (res.status === 200) {
      resetField('password');
      updateUser(data);
      toast.success('Profile Updated');
    } else {
      toast.error(data.message);
    }
  } catch (error) {
    toast.error('Profile Update Error: ', error);
  }
};
```

In the `doSubmit()` function above, we send a PATCH request to the `/users/update/` Express route.

When the request is successful, we reset the `password` field from the form, update the user context data, and show a success toast:

```
resetField('password');
updateUser(data);
toast.success('Profile Updated');
```

The next step is to write a function that handles the user deletion request:

```
const handleDeleteUser = async () => {
  try {
    const res = await fetch(`${API_BASE_URL}/users/delete/${user._id}`, {
      method: 'DELETE',
      credentials: 'include',
    });
    const data = await res.json();
    if (res.status === 200) {
      toast.success(data.message);
      updateUser(null);
      navigate('/');
    } else {
      toast.error(data.message);
```

```
    }
  } catch (error) {
    toast.error('Delete Error: ', error);
  }
};
```

After the user is deleted, the user context data is set to `null`, and the user is directed to the Home page.

The last function we need to add is for handling the sign out process:

```
const handleSignOut = async () => {
  try {
    const res = await fetch(`${API_BASE_URL}/auth/signout`, {
      credentials: 'include',
    });
    const data = await res.json();
    toast.success(data.message);
    updateUser(null);
    navigate('/');
  } catch (error) {
    toast.error(error);
  }
};
```

With the functions ready, it's time to write the `return` statement and render the form. Add the `<form>` wrapper as follows:

```
return (
  <Box p='3' maxW='lg' mx='auto'>
    <Heading
      as='h1'
      fontSize='3xl'
      fontWeight='semibold'
      textAlign='center'
      my='7'
    >
      Your Profile
    </Heading>
    <form onSubmit={handleSubmit(doSubmit)}>
    </form>
```

```
  </Box>
);
```

Inside the `<form>` element, add the form inputs needed to update the user data.

For now, the user avatar will just be shown. We're going to add a way to update the user avatar in the next chapter:

```
<Stack gap='4'>
  <Center>
    <Image
      alt='profile'
      rounded='full'
      h='24'
      w='24'
      objectFit='cover'
      cursor='pointer'
      mt='2'
      src={user.avatar}
    />
  </Center>
  <FormControl isInvalid={errors.username}>
    <Input
      id='username'
      type='text'
      placeholder='username'
      {...register('username', { required: 'Username is required' })}
    />
    <FormErrorMessage>
      {errors.username && errors.username.message}
    </FormErrorMessage>
  </FormControl>
  <FormControl isInvalid={errors.email}>
    <Input
      id='email'
      type='email'
      placeholder='email'
      {...register('email', { required: 'Email is required' })}
    />
    <FormErrorMessage>
      {errors.email && errors.email.message}
    </FormErrorMessage>
  </FormControl>
```

```
  <FormControl isInvalid={errors.password}>
    <Input
      id='password'
      type='password'
      placeholder='New password'
      {...register('password', { required: 'Password is required' })}
    />
    <FormErrorMessage>
      {errors.password && errors.password.message}
    </FormErrorMessage>
  </FormControl>
  <Button
    type='submit'
    isLoading={isSubmitting}
    colorScheme='teal'
    textTransform='uppercase'
  >
    Update Profile
  </Button>
</Stack>
```

With the form completed, add another `Stack` component below the `<form>` element as follows:

```
<Stack gap='4' mt='5'>
  <Link
    as={RouterLink}
    to='/create-task'
    p='2'
    bg='green.500'
    rounded='lg'
    textTransform='uppercase'
    textAlign='center'
    textColor='white'
    fontWeight='semibold'
    _hover={{ bg: 'green.600' }}
  >
    Create New Task
  </Link>
  <Flex justify='space-between'>
    <Text
      as='span'
      color='red.600'
      cursor='pointer'
      onClick={handleDeleteUser}
```

```
      >
        Delete Account
      </Text>
      <Text
        as='span'
        color='red.600'
        cursor='pointer'
        onClick={handleSignOut}
      >
        Sign Out
      </Text>
    </Flex>
    <Text textAlign='center'>
      <Link
        as={RouterLink}
        to='/tasks'
        color='teal'
        _hover={{ textDecor: 'none' }}
      >
        Show Tasks
      </Link>
    </Text>
  </Stack>
```

These buttons and links allow the users to perform additional actions. They can create a task, see all tasks, delete their own account, or sign out.

You can try updating user profile data using the form, but the user avatar can't be changed yet.

Creating a Delete Confirmation Component

If you click on the *Delete Account* link, the application will immediately delete the account without any confirmation.

This is dangerous in case a user mistakenly clicks on the button. Let's add an alert box that asks if the user really wants to delete his or her account.

On the `components/` folder, create a new file named `DeleteConfirmation.jsx` and write the following code in it:

```
import {
  Button,
  AlertDialog,
  AlertDialogBody,
  AlertDialogFooter,
  AlertDialogHeader,
  AlertDialogContent,
  AlertDialogOverlay,
} from '@chakra-ui/react';

export default function DeleteConfirmation({ alertTitle, handleClick, isOpen,
onClose }) {
  return (
    <AlertDialog isOpen={isOpen} onClose={onClose}>
      <AlertDialogOverlay>
        <AlertDialogContent>
          <AlertDialogHeader fontSize='lg' fontWeight='bold'>
            {alertTitle}
          </AlertDialogHeader>

          <AlertDialogBody>
            Are you sure? You can't undo this action.
          </AlertDialogBody>

          <AlertDialogFooter>
            <Button onClick={onClose}>Cancel</Button>
            <Button colorScheme='red' onClick={handleClick} ml={3}>
              Delete
            </Button>
          </AlertDialogFooter>
        </AlertDialogContent>
      </AlertDialogOverlay>
    </AlertDialog>
  );
}
```

This component accepts 4 arguments:

- `alertTitle` - The title of the alert box (string)

- `handleClick` - The process to run when confirmed (function)
- `isOpen` - The process to open the alert dialog (function)
- `alertTitle` - The process to close the alert dialog (function)

In your `Profile.jsx` file, import the component along with the `useDisclosure` hook from Chakra UI as follows:

```
import { useDisclosure } from '@chakra-ui/react';
import DeleteConfirmation from '../components/DeleteConfirmation.jsx';
```

Inside the `Profile` component, initialize the `useDisclosure` hook as follows:

```
const { isOpen, onOpen, onClose } = useDisclosure();
```

On the `Text` component where you show the *Delete Account* link, change the `onClick` prop to reference `onOpen` function as follows:

```
<Text
  as='span'
  color='red.600'
  cursor='pointer'
  onClick={onOpen}
>
  Delete Account
</Text>
```

The last step is to add the `DeleteConfirmation` component inside the `Box` component, and pass the props as shown below:

```
return (
  <Box p='3' maxW='lg' mx='auto'>
    <DeleteConfirmation
```

```
        alertTitle='Delete Account'
        handleClick={handleDeleteUser}
        isOpen={isOpen}
        onClose={onClose}
      />
      {/* the rest... */}
    </Box>
  )
```

Now when you click on the *Delete Account* link, an alert box will be shown before running the delete process.

The account will be deleted only when the user confirms the decision.

Summary

The code added in this chapter can be found at https://g.codewithnathan.com/mern-13

In this chapter, you've implemented the `verifyToken()` middleware in Express to guard specific routes from unauthorized access.

You've also finished the Profile page which enables the users to update their profile data.

CHAPTER 14: INTEGRATING CLOUDINARY FOR IMAGE UPLOAD

The process to change the user avatar image is as follows:

- React accepts an image uploaded by the user
- On form submission, React will send the image to Express using a special upload route
- Express will accept the image and send back a URL of the uploaded image
- React would then send a request to update the user profile, including the new avatar URL

To handle this process, you need a place where you can store the uploaded images.

You can store images on the server where Express runs, but it's better to use an optimized cloud platform instead.

Many cloud platforms can be used for the purpose of storing images, such as Google Cloud Platform, Amazon Web Services,

and Cloudinary.

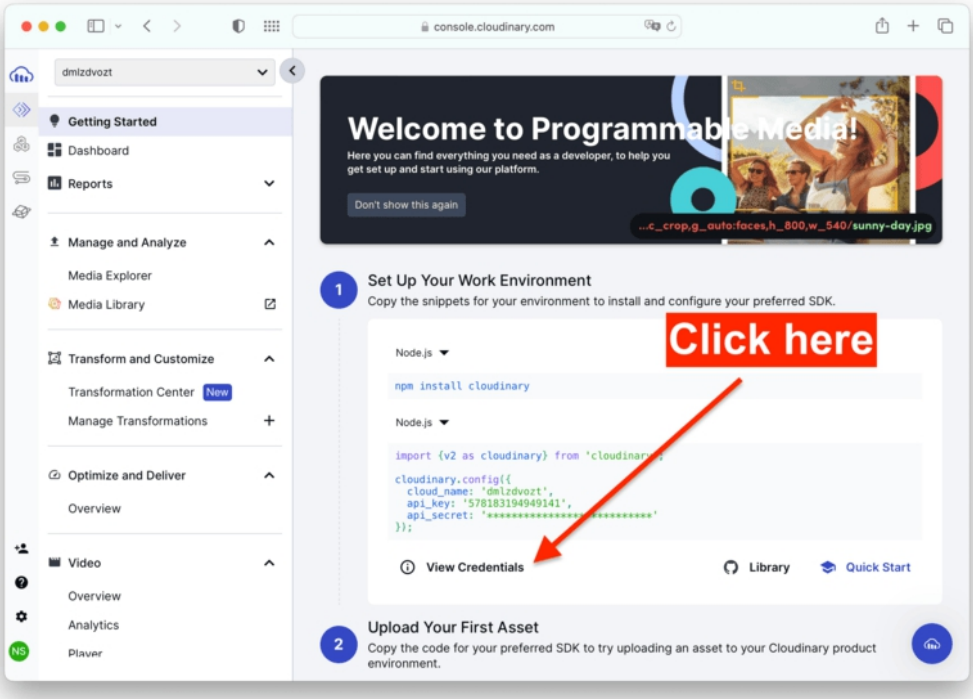
The one we will use for this application will be Cloudinary, a file uploader service that you can use to upload images and videos.

Cloudinary is easy to integrate with Express, as I will show you in this chapter.

Cloudinary Setup

To use Cloudinary, create a free account at https://cloudinary.com, and Cloudinary will ask some questions.

You can skip the questions and be directed to the Getting Started page as shown below. You can follow the guide shown on the right side of the screen:



There's no need to configure anything in Cloudinary. All you need to do is get the credentials by clicking on the *View Credentials* button, then copy these credentials to `server/.env` file as follows:

```
CLOUDINARY_CLOUD_NAME=xxxxxxxxxx
CLOUDINARY_API_KEY=xxxxxxxxxx
CLOUDINARY_API_SECRET=xxxxxxxxxx
```

Next, install the Cloudinary library using npm as follows:

```
npm install cloudinary
```

Now that Cloudinary is installed, you can start creating the route for image upload in Express.

Express Image Upload API

To accept images in Express, you need to use the `express-fileupload` middleware. First, install the package using npm:

```
npm install express-fileupload
```

Then use the middleware in `server.js` file as follows:

```
import fileUpload from 'express-fileupload';

app.use(express.json());
app.use(cookieParser());
app.use(fileUpload());
```

To complete the image upload API, you need to create the controller and route for Cloudinary.

Inside the `controllers/` folder, create the `cloudinary.controller.js` file and write the code below:

```
import { v2 as cloudinary } from 'cloudinary';

const { CLOUDINARY_CLOUD_NAME, CLOUDINARY_API_KEY, CLOUDINARY_API_SECRET } =
  process.env;

cloudinary.config({
  cloud_name: CLOUDINARY_CLOUD_NAME,
  api_key: CLOUDINARY_API_KEY,
  api_secret: CLOUDINARY_API_SECRET,
});
```

Here, you import version 2 of the Cloudinary library, and then initialize the library using the credentials you get in the previous section.

Next, you need to create a function that uploads an image to Cloudinary as follows:

```
const cldUpload = async imagePath => {
  const options = {
    use_filename: true,
    unique_filename: false,
    overwrite: true,
  };
  try {
    const result = await cloudinary.uploader.upload(imagePath, options);
    return result.secure_url;
  } catch (error) {
    console.error(error);
  }
};
```

The `cldUpload()` function above uploads an image to the cloudinary server and returns the secure URL (`secure_url`) variable.

Below this function, you need to write another function to accept the upload request:

```
export const addImage = async (req, res, next) => {
  try {
    const { data, mimetype } = req.files.image;
    const base64String = Buffer.from(data).toString('base64');
    const withPrefix = `data:${mimetype};base64,${base64String}`;
    const imageUrl = await cldUpload(withPrefix);
    return res.status(200).json({ status: 'ok', imageUrl });
  } catch (error) {
    next({ status: 500, error });
  }
};
```

The `addImage()` function will be the middleware that process the image upload requests from React.

Next, create a new route named `cloudinary.route.js` and call the `addImage` middleware here:

```
import express from 'express';

import { addImage } from '../controllers/cloudinary.controller.js';

import { verifyToken } from '../libs/middleware.js';

const router = express.Router();

router.post('/upload', verifyToken, addImage);

export default router;
```

There's only one route you need to define for image upload.

Now add the route in `server.js` as shown below:

```
import cldRouter from './routes/cloudinary.route.js';
// ...
app.use('/api/v1/users', userRouter);
```

```
app.use('/api/v1/auth', authRouter);
app.use('/api/v1/image', cldRouter);
```

Now the API route for image upload is ready. The next step is to update the Profile page and enable users to upload a new avatar image.

React Dropzone for Image Upload

To enable image upload in React, we're going to create a component where the user can drag and drop an image into the avatar rendered on the Profile page.

The usual click and browse a file from the computer can still be done, but the drag and drop functionality makes the application more interactive.

To implement the image upload, you need to use the React Dropzone library:

```
npm install react-dropzone
```

This library provides a simple hook that you can use to create a drag and drop zone in your React application.

On the `components/` folder, create a new component called `AvatarUploader.jsx` and import the following modules:

```
import { useCallback } from 'react';
import { useDropzone } from 'react-dropzone';
import { Center, Input, Image, Tooltip } from '@chakra-ui/react';
```

The `useCallback` hook is used to cache a function. This reduces the need to recreate the function on every re-render, improving

React performance.

The `useDropzone` hook is used to create a dropzone where the user can drop a file. The rest are Chakra UI components used for the dropzone.

Next, create the `AvatarUploader` component as follows:

```
export function AvatarUploader({ imageUrl, onFieldChange, setFiles }) {
  const convertFileToUrl = file => URL.createObjectURL(file);

  const onDrop = useCallback(acceptedFiles => {
    setFiles(acceptedFiles);
    onFieldChange(convertFileToUrl(acceptedFiles[0]));
  }, []);

  const { getRootProps, getInputProps } = useDropzone({
    onDrop,
    accept: {
      'image/jpeg': [],
      'image/png': [],
    },
  });
}
```

The component accepts 3 arguments:

- `imageUrl` - The user avatar's URL
- `onFieldChange` - The function to run when a new image is added
- `setFiles` - The function to set the image file in React as a state

These props allow the component to communicate with the form on the Profile Page. It will be clear when you use this

component later.

The `convertFileToUrl` function converts an image file into a URL string that represents the image.

The `onDrop` function will be executed when a file is dropped to the drop zone. This function is wrapped in `useCallback` for optimization purposes.

The `useDropzone` hook is used to define the drop zone rules. Here, we define that the zone accepts only images in `jpeg` or `png` format, and it needs to run the `onDrop` function when a new file is received.

The hook returns two functions:

- The `getRootProps` function converts a component into a drop zone
- The `getInputProps` connects a file input element to React Dropzone

Now you need to render the uploader component as shown below:

```
return (
  <Center {...getRootProps()}>
    <Input {...getInputProps()} id='avatar' cursor='pointer' />
    <Tooltip label='Change your avatar'>
      <Image
        alt='profile'
        rounded='full'
        h='24'
        w='24'
        objectFit='cover'
        cursor='pointer'
```

```
          mt='2'
          src={imageUrl}
        />
      </Tooltip>
    </Center>
  );
```

In the `return` statement, the `Center` component creates a `<div>` aligned at the center. This component will be the drag and drop zone.

The `Input` component is a normal `<input>` element that you can click to browse and upload a file. The `getInputProps()` function connects this component to React Dropzone.

The `Tooltip` component shows a helper text when the user hovers over the image, and the `Image` component renders the avatar.

Integrating AvatarUploader to Profile Page

Alright, it's time to import the `AvatarUploader` component to our Profile page.

You need to import `useState` from React and `Controller` from React Hook Form as well:

```
import { useState } from 'react';
import { useForm, Controller } from 'react-hook-form';
import { AvatarUploader } from '../components/AvatarUploader.jsx';
```

The `Controller` component allows you to control a custom input, such as the `AvatarUploader` component.

Next, initialize a state to keep the file uploaded by the user as follows:

```
const [files, setFiles] = useState(false);
```

When an image is uploaded, this state will hold that image file.

You also need to get the `control` object from useForm. This object will let `Controller` know which form controls the component:

```
const {
    control,
    // ...
  } = useForm({
    defaultValues: {
      avatar: user.avatar,
      username: user.username,
      email: user.email,
    },
  });
```

After that, replace the `Image` component inside the `<form>` with the one shown below:

```
<form onSubmit={handleSubmit(doSubmit)}>
  <Stack gap='4'>
    <Controller
      name='avatar'
      control={control}
      rules={{ required: true }}
      render={({ field }) => (
        <AvatarUploader
          onFieldChange={field.onChange}
          imageUrl={field.value}
          setFiles={setFiles}
        />
      )}
    />
```

```
    </Stack>
  </form>
```

The `Controller` component allows you to render a custom field using its `render` prop. Here, we render the `AvatarUploader` component and set the props appropriately.

The `onFieldChange` prop will convert the image to a URL string, which is set as the `value` of the field.

The `setFiles` prop will set the new file as the value of the `files` state.

Now you need to create a function that sends the image to Express using the `/image/upload` route:

```
const handleFileUpload = async files => {
  const formData = new FormData();
  formData.append('image', files[0]);
  try {
    const res = await fetch(`${API_BASE_URL}/image/upload`, {
      method: 'POST',
      credentials: 'include',
      body: formData,
    });
    const response = await res.json();
    return response.imageUrl;
  } catch (error) {
    console.log(error);
    Throw(error);
  }
};
```

This function creates a new `FormData` object, and the image will be appended to this form.

You then send a POST request to Express, which uploads the image to Cloudinary and returns the `imageUrl`.

As the last step, you need to call this function from `doSubmit()` as follows:

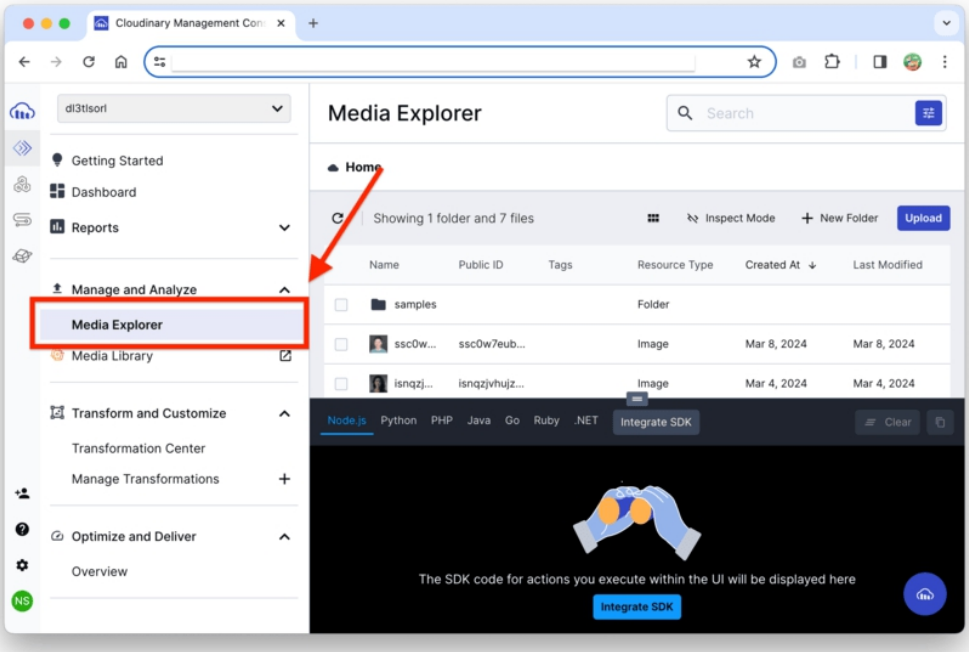
```
const doSubmit = async values => {
  try {
    if (files.length > 0) {
      const newUrl = await handleFileUpload(files);
      if (newUrl) {
        values.avatar = newUrl;
      }
    }
    // ...
  }
}
```

Once the file has been uploaded, you set the value of `values.avatar` as Cloudinary image URL. The user update function can then proceed as before.

Now you can test the uploader. Upload a new image, then submit the form.

You'll see the `avatar` URL has the res.cloudinary.com domain.

You can see this image on Cloudinary console under the *Media Explorer* menu as shown below:



You've done a great job integrating the upload image feature. Congratulations!

Summary

The code added in this chapter is available at https://g.codewithnathan.com/mern-14

As a web developer, you will frequently work with cloud platforms to develop your application features.

Cloudinary is just one of many cloud platforms, so I hope the experience of using it helps you see that it's not that difficult to integrate cloud platforms.

It's always a good idea to make yourself familiar with popular cloud platforms.

CHAPTER 15: SHOWING TASKS TO USERS

With the profile page completed, it's time to add functionalities to show the tasks stored in the database.

We'll start with creating the task routes in Express, and then use React to fetch and show available tasks to the user.

Let's jump in.

Creating Task Controller in Express

Just like with user data, you need to create controllers and routes to manipulate the task data.

Inside the `controllers/` folder, create a new file named `task.controller.js`, import the required modules, and select the collection used in this file:

```
import { db } from '../libs/dbConnect.js';
import { ObjectId } from 'mongodb';

const collection = db.collection('tasks');
```

Next, write the `getTasksByUser()` function that will get all tasks created by a specific user:

```
export const getTasksByUser = async (req, res, next) => {
  try {
    const query = { owner: new ObjectId(req.params.id) };

    const tasks = await collection.find(query).toArray();

    res.status(200).json({ tasks });
  } catch (error) {
    next({ status: 500, error });
  }
};
```

Then, create a function to retrieve a single task as follows:

```
export const getTask = async (req, res, next) => {
  try {
    const query = { _id: new ObjectId(req.params.id) };
    const task = await collection.findOne(query);

    if (!task){
      return next({ status: 404, message: 'Task not found!' });
    }

    res.status(200).json(task);
  } catch (error) {
    next({ status: 500, error });
  }
};
```

Then, create a function to create a new task as shown below:

```
export const createTask = async (req, res, next) => {
  try {
    const newTask = req.body;
    newTask.owner = new ObjectId(req.user.id);
    newTask.createdAt = new Date().toISOString();
    newTask.updatedAt = new Date().toISOString();
    const task = await collection.insertOne(newTask);
    return res.status(200).json(task);
  } catch (error) {
```

```
    next({ status: 500, error });
  }
};
```

After that, add a function to update an existing task from the collection:

```
export const updateTask = async (req, res, next) => {
  try {
    const query = { _id: new ObjectId(req.params.id) };
    const data = {
      $set: {
        ...req.body,
        owner: new ObjectId(req.body.owner),
        updatedAt: new Date().toISOString(),
      },
    };
    const options = {
      returnDocument: 'after',
    };
    const updatedTask = await collection.findOneAndUpdate(query, data,
options);
    res.status(200).json(updatedTask);
  } catch (error) {
    next({ status: 500, error });
  }
};
```

The last function below is used to delete a task:

```
export const deleteTask = async (req, res, next) => {
  try {
    const query = { _id: new ObjectId(req.params.id) };
    await collection.deleteOne(query);
    res.status(200).json('Task has been deleted!');
  } catch (error) {
    next({ status: 500, error });
  }
};
```

Now that the controller is finished, let's create the route next.

Creating Express Task Routes

Create a `task.route.js` file inside the `routes/` folder and write the following code in it:

```
import express from 'express';

import {
  getTask,
  getTasksByUser,
  createTask,
  updateTask,
  deleteTask,
} from '../controllers/task.controller.js';

import { verifyToken } from '../libs/middleware.js';

const router = express.Router();

router.get('/:id', verifyToken, getTask);
router.get('/user/:id', verifyToken, getTasksByUser);
router.post('/create', verifyToken, createTask);
router.patch('/:id', verifyToken, updateTask);
router.delete('/:id', verifyToken, deleteTask);

export default router;
```

The routes are similar to the user routes, so I won't explain it again.

Let's add this route to `server.js` file:

```
import taskRouter from './routes/task.route.js';

// ...

app.use('/api/v1/tasks', taskRouter);
```

Now our task routes are ready to use. It's time to update the frontend.

Showing All Tasks in React

The tasks stored in the database will be shown on the `/tasks` route, so let's open the `pages/Tasks.jsx` file and start importing the required modules:

```
import { useState, useEffect } from 'react';
import { useUser } from '../context/UserContext';
import { API_BASE_URL } from '../util';
import { Link } from 'react-router-dom';
import {
  Badge,
  Box,
  Button,
  Flex,
  Heading,
  Select,
  Table,
  Thead,
  Tbody,
  Tr,
  Th,
  Td,
  TableContainer,
} from '@chakra-ui/react';
```

Next, create the `Tasks` component, initialize the states, and create a `useEffect` hook to get the user tasks:

```
export default function Tasks() {
  const { user } = useUser();
  const [tasks, setTasks] = useState([]);

  useEffect(() => {
    const fetchTasks = async () => {
      const res = await fetch(`${API_BASE_URL}/tasks/user/${user._id}`, {
        credentials: 'include',
      });
      const { tasks } = await res.json();
      setTasks(tasks);
    };
    fetchTasks();
```

```
  }, []);
}
```

The `useEffect` hook above will run only once to retrieve the tasks data. Once the data is retrieved, call the `setTasks()` function to update the `tasks` state value.

After that, you can show the tasks data by rendering a table as shown below:

```
return (
  <Box p='5' maxW='3lg' mx='auto'>
    <Heading
      as='h1'
      fontSize='3xl'
      fontWeight='semibold'
      textAlign='center'
      my='7'
    >
      Tasks to do
    </Heading>
    <TableContainer>
      <Table px='3' border='2px solid' borderColor='gray.100'>
        <Thead backgroundColor='gray.100'>
          <Tr>
            <Th>Task</Th>
            <Th>Priority</Th>
            <Th>Status</Th>
            <Th>Due Date</Th>
          </Tr>
        </Thead>
        <Tbody>
          {tasks.map(task => (
            <Tr key={task._id}>
              <Td>
                <Link to={`/tasks/${task._id}`}>{task.name}</Link>
              </Td>
              <Td>
                <Badge
                  colorScheme={task.priority === 'urgent' ? 'red' : 'gray'}
                >
                  {task.priority}
                </Badge>
```

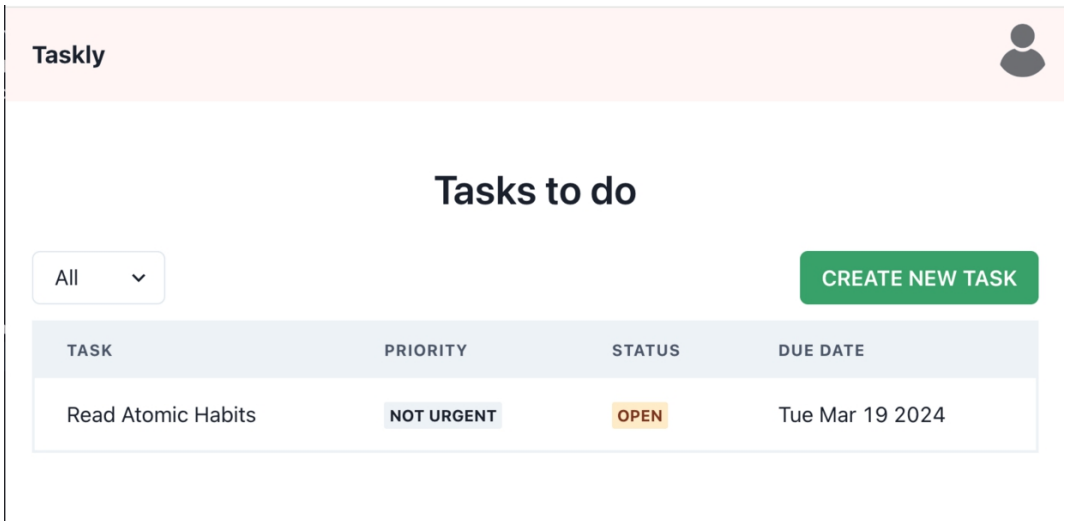
```
              </Td>
              <Td>
                <Badge
                  colorScheme={task.status === 'open' ? 'orange' : 'green'}
                >
                  {task.status}
                </Badge>
              </Td>
              <Td>{task.due ? new Date(task.due).toDateString() : ''}</Td>
            </Tr>
          ))}
        </Tbody>
      </Table>
    </TableContainer>
  </Box>
);
```

You can see the table rendered on the page already, but it looks a bit empty.

Above the `TableContainer` component, let's add a dropdown for filtering the tasks by status and a button to create a new task:

```
<Flex justify='space-between' mb='3'>
  <Box w='100px'>
    <Select placeholder='All'>
      <option value='open'>Open</option>
      <option value='done'>Done</option>
    </Select>
  </Box>
  <Button
    colorScheme='green'
    textTransform='uppercase'
    fontWeight='semibold'
  >
    <Link to='/create-task'>Create New Task</Link>
  </Button>
</Flex>
```

Now the page looks better. Here's what you should see on the tasks page:



If you try to refresh the page many times, you would see the page blink a few milliseconds before showing the tasks data.

This is because the page has already been shown before React finished fetching data.

When the component is rendered on the screen, the table is still empty, and when the data is fetched, React populates the table.

To offer a better user experience, you can show a skeleton component instead of an empty table.

Adding Skeleton Component for Tasks Page

A skeleton component is simply a loading indicator that intuitively communicates that the page is still loading to the user.

Chakra UI already has skeleton components that you can use, so let's create one for the Tasks page.

Inside the `src/` folder, create a new folder named `_skeletons`, then create a file named `TasksSkeleton.jsx`:

```
src
└── _skeletons
    └── TasksSkeleton.jsx
```

On the `TasksSkeleton.jsx` file, create a skeleton component as follows:

```
import {
  Skeleton,
  Stack
} from '@chakra-ui/react';

export default function TasksSkeleton() {
  return (
    <Stack p='5' maxW='3lg' mx='auto' gap='4'>
      <Skeleton height='20px' my='7' />
      <Skeleton height='20px' />
      <Skeleton height='100px' />
    </Stack>
  );
}
```

Back on the `Tasks.jsx` file, import the skeleton, then show the skeleton by adding a `return` statement as follows:

```
return <TasksSkeleton />;
return (
  // ...
)
```

We will show the skeleton only when the `tasks` state is not ready, so add an `if` statement in front of the `return <TasksSkeleton />` statement:

```
if(!tasks){
  return <TasksSkeleton />;
}
```

When initializing the `tasks` state, you need to remove the empty array that's set as the initial value:

```
const [tasks, setTasks] = useState();
```

The empty array was set as the default value of the `tasks` state to prevent an error when calling the `map()` method in the `return` statement.

Now that we have a skeleton returned when the `tasks` state is undefined, we no longer need to set an empty array as the default value.

If you refresh the page again, you'll see the skeleton shown for a moment before the `tasks` state is filled. Nice work!

Installing React Icons Library

In the next section, we're going to work on the single task page, which uses an icon to improve the user experience.

React Icons is an icon library that contains popular icons from many frameworks. You can install the library using npm:

```
npm install react-icons
```

To use an icon, you only need to import it from the library, as you'll see in the next section.

Finishing the Single Task Page

When you click on the task name, we will be directed to the single task page, so let's work on that page now.

The single task page will show the task detail and three links: to delete a task, edit a task, and show all tasks.

This page will also have a skeleton component that will be shown before the task data has been loaded.

First, you need to create a `_skeletons/SingleTaskSkeleton.jsx` file and write the following code in it:

```
import {
  Box,
  Stack,
  Skeleton,
} from '@chakra-ui/react';

export default function SingleTaskSkeleton() {
  return (
    <Box p='3' maxW='lg' mx='auto'>
      <Stack gap='4'>
        <Skeleton height='20px' my='10' />
        <Skeleton height='20px'/>
        <Skeleton height='20px'/>
        <Skeleton height='20px'/>
      </Stack>
    </Box>
  );
}
```

Next, open the `pages/SingleTask.jsx` file and import the modules that will be used in this component:

```
import { Link as RouterLink, useNavigate, useParams } from 'react-router-
dom';
import { useState, useEffect } from 'react';
import { API_BASE_URL } from '../util';
```

```
import {
  Badge,
  Box,
  Flex,
  Heading,
  Stack,
  Text,
  Link,
  Card,
  CardBody,
} from '@chakra-ui/react';
import SingleTaskSkeleton from '../_skeletons/SingleTaskSkeleton';
import { BsChevronLeft } from 'react-icons/bs';
```

The `useParams` hook from React Router allows you to get URL parameters.

The `BsChevronLeft` is an icon component from `react-icons`.

Now you need to initialize the states and hooks inside the `SingleTask` component:

```
export default function SingleTask() {
  const [task, setTask] = useState();
  const { taskId } = useParams();
  const navigate = useNavigate();
}
```

The `useParams` hook returns an object containing the current URL parameters. The name of the route parameter is used as the name of the property (we use `/tasks/:taskId` when defining the route)

Next, add a `useEffect` hook that fetches the single task data:

```
useEffect(() => {
  const fetchTask = async () => {
    const res = await fetch(`${API_BASE_URL}/tasks/${taskId}`, {
      credentials: 'include',
```

```
    });
    const data = await res.json();
    setTask(data);
  };
  fetchTask();
}, []);
```

Then, write an `if` statement to check whether you should return the skeleton component as follows:

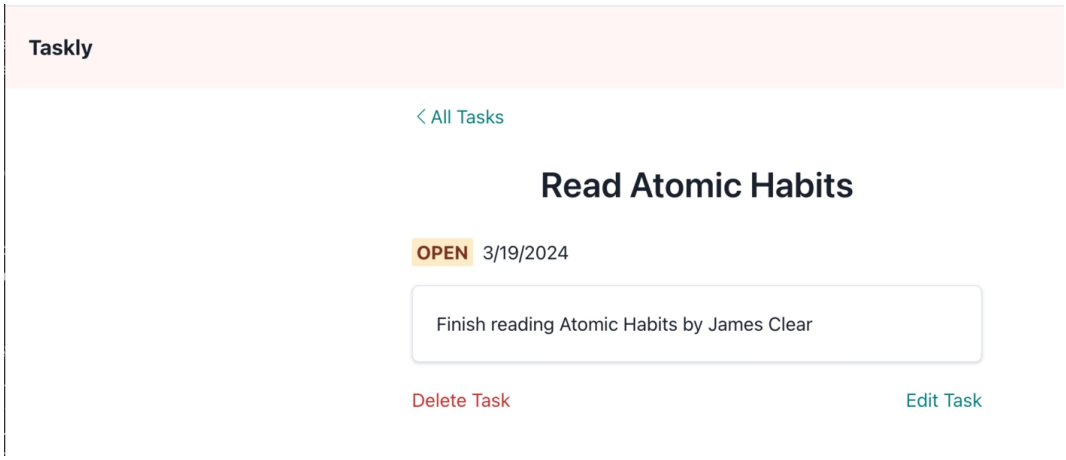
```
if (!task) {
  return <SingleTaskSkeleton />;
}
```

Below the `if` statement, write the `return` statement to render the task data:

```
return (
  <Box p='3' maxW='lg' mx='auto'>
    <Link
      as={RouterLink}
      to={`/tasks`}
      color='teal'
      _hover={{ textDecor: 'none' }}
      display='flex'
      alignItems='center'
    >
      <BsChevronLeft /> All Tasks
    </Link>
    <Heading fontSize='3xl' fontWeight='semibold' textAlign='center' my='7'>
      {task.name}
    </Heading>
    <Stack direction='row'>
      <Badge
        fontSize='md'
        colorScheme={task.status === 'open' ? 'orange' : 'green'}
      >
        {task.status}
      </Badge>
      {task.due && <Text>{new Date(task.due).toLocaleDateString()}</Text>}
    </Stack>
    <Card mt='4' border='1px solid' borderColor='gray.200'>
      <CardBody>
```

```
          <Text>{task.description}</Text>
        </CardBody>
      </Card>
      <Flex justify='space-between' mt='5'>
        <Text as='span' color='red.600'>
          Delete Task
        </Text>
        <Link
          as={RouterLink}
          to={`/update-task/${task._id}`}
          color='teal'
          _hover={{ textDecor: 'none' }}
        >
          Edit Task
        </Link>
      </Flex>
    </Box>
  );
```

Okay, now you can see the single task page rendered as follows:



There's a *Delete Task* text that we haven't finished yet. We'll do it in the next chapter.

Summary

The code added in this chapter is available at https://g.codewithnathan.com/mern-15

In this chapter, you've added an Express route that will be used for creating, reading, updating, and deleting task data.

The pattern of creating a controller, a route, and then integrating it into the main Express application is what you will use when developing the backend system of an application.

The controller stores the functions that will perform the business logic, while the route defines the URL and the function.

Once the backend is ready, continue by building the frontend that will use those routes.

This way, you have an efficient process to code a full-stack web application.

CHAPTER 16: DELETE, CREATE, AND UPDATE TASKS

Now that we can show the tasks to the users, it's time to add delete, create, and update task functionalities to the frontend.

The code patterns you will see in this chapter have already been explained in the previous chapters. It will be modified to suit the task data specifications.

You're also going to use a date input so that users can enter the deadline of a task. Let's do this.

Deleting a Task

To delete a task, you need to update the *Delete Task* text added on the single task page.

First, import the `DeleteConfirmation` module and `useDisclosure` into the page:

```
import { useDisclosure } from '@chakra-ui/react';
import DeleteConfirmation from '../components/DeleteConfirmation';
```

Next, initialize the `useDisclosure` hook inside the `SingleTask` component:

```
export default function SingleTask() {
  const { isOpen, onOpen, onClose } = useDisclosure();
  // ...
}
```

After that, create a `handleDeleteTask()` function that sends a DELETE request to Express:

```
const handleDeleteTask = async () => {
  const res = await fetch(`${API_BASE_URL}/tasks/${taskId}`, {
    method: 'DELETE',
    credentials: 'include',
  });
  const data = await res.json();
  if (res.status === 200) {
    toast.success(data.message);
    navigate('/tasks');
  } else {
    toast.error(data.message);
  }
};
```

When the task is deleted, the user will be directed to the All Tasks page.

Now you need to add the `DeleteConfirmation` component inside the `return` statement. You can put it as the last component inside the `Box` component:

```
return(
  <Box>
    {/* ... other code */}
    <DeleteConfirmation
      alertTitle='Delete Task'
      handleClick={handleDeleteTask}
      isOpen={isOpen}
```

```
      onClose={onClose}
    />
  </Box>
);
```

The last step is to update the `Text` component that has the Delete Task text:

```
<Text as='span' color='red.600' cursor='pointer' onClick={onOpen}>
  Delete Task
</Text>
```

The `cursor` prop will change the cursor to a pointer icon. When you click on the text, a delete confirmation dialog will be shown.

And now you can delete the task from the database.

Creating the Task Form

The next task is to add a way to create and update a task. To do so, you need to create a task form that users can use.

This task form can be used for both creating and updating a task, depending on the currently active route.

Because there's a due date in the task data specifications, you need to create an input date (also called datepicker) in this form.

Instead of creating a datepicker from scratch, let's install the React Datepicker library:

```
npm install react-datepicker
```

This library is easy to use and can be integrated into React Hook Form. I'll show you how to use it below.

On the `components/` folder, create a new file named `TaskForm.jsx` and import the required modules below:

```
import DatePicker from 'react-datepicker';
import 'react-datepicker/dist/react-datepicker.css';
import { useForm, Controller } from 'react-hook-form';
import { SERVER_BASE_URL } from '../util';
import { toast } from 'react-hot-toast';
import { useNavigate } from 'react-router-dom';
import {
  Stack,
  Flex,
  FormControl,
  FormErrorMessage,
  Input,
  Textarea,
  Select,
  Button,
} from '@chakra-ui/react';
```

To use the React Datepicker library, you need to import the component from `react-datepicker`, and then import the CSS file used for this component.

The rest of the imports are already used before.

As always, the next step is to define the component and initialize the hooks we're going to use:

```
export default function TaskForm({ type, task }) {
  const {
    handleSubmit,
    register,
    control,
    formState: { errors, isSubmitting },
  } = useForm({
    defaultValues:
```

```
      type === 'update'
        ? {
            ...task,
            due: task.due ? new Date(task.due) : '',
          }
        : {},
  });

  const navigate = useNavigate();
}
```

In the code above, you see that the `TaskForm` component accepts the `type` and `task` props.

The `type` will be used to determine whether the form is for creating or updating a task. The `task` prop contains the task data you want to update.

When initializing the form, the `defaultValues` passed to the `useForm` hook will be different depending on the `type` prop.

When the `type` is 'update', then the `task` prop data will be used as the default value. Otherwise, the `defaultValue` is empty.

The next step is to write the `doSubmit()` function. Here, you need to check the `type` prop value once again:

```
const doSubmit = async values => {
  if (type === 'create') {
    const res = await fetch(`${API_BASE_URL}/tasks/create`, {
      method: 'POST',
      credentials: 'include',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(values),
    });
    const response = await res.json();
    if (res.status === 200) {
```

```
      toast.success(`New Task Created: ${values.name}`);
      navigate(`/tasks/${response.insertedId}`);
    } else {
      toast.error(response.message);
    }
  }
};
```

When the `type` is 'create', send a POST request to save the task as a new task.

Now when the `type` is 'update', send a PATCH request to update an existing task as shown below:

```
if (type === 'update') {
  delete values._id;
  const res = await fetch(`${SERVER_BASE_URL}/tasks/${task._id}`, {
    method: 'PATCH',
    credentials: 'include',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(values),
  });
  const response = await res.json();
  if (res.status === 200) {
    toast.success(`Task Updated: ${values.name}`);
    navigate(`/tasks/${task._id}`);
  } else {
    toast.error(response.message);
  }
}
```

Here, the form values are sent as a PATCH request except the `_id` value.

This is because you can't include the `_id` of the task when calling MongoDB update function.

The id will be included in the API URL route instead, as shown in the `fetch()` function.

When the request is fulfilled, the user will be redirected to the single task page.

Next, write the `return` statement to render the form.

The form will be shown as two columns on medium to large screens, or a single column on small screens.

Begin with creating the layout as follows:

```
return (
  <form onSubmit={handleSubmit(doSubmit)}>
    <Stack direction={{ base: 'column', md: 'row' }} gap='4'>
      <Flex direction='column' flex='1' gap='4'>
      </Flex>
      <Flex direction='column' flex='1' gap='4'>
      </Flex>
    </Stack>
  </form>
);
```

The `Stack` component will use the `direction` prop to determine the direction of the component.

When on medium to large screens, the direction is row, so the two `Flex` components will be shown side by side.

On small screens, the `Flex` components will be stacked from top to bottom.

Next, write the inputs for the first `Flex` component, which contains the task name and description:

```
<Stack direction={{ base: 'column', md: 'row' }} gap='4'>
  <Flex direction='column' flex='1' gap='4'>
    <FormControl isInvalid={errors.name}>
      <Input
        id='name'
        type='text'
        placeholder='Task Name'
        {...register('name', { required: 'Task Name is required' })}
      />
      <FormErrorMessage>
        {errors.name && errors.name.message}
      </FormErrorMessage>
    </FormControl>
    <FormControl isInvalid={errors.description}>
      <Textarea
        id='description'
        type='text'
        placeholder='Description'
        rows={4}
        {...register('description', {
          required: 'Description is required',
        })}
      />
      <FormErrorMessage>
        {errors.description && errors.description.message}
      </FormErrorMessage>
    </FormControl>
  </Flex>
  <Flex direction='column' flex='1' gap='4'>
  </Flex>
</Stack>
```

For the second `Flex` component, you need to add the priority, status, due date, and submit button.

Add the priority and status inputs first:

```
<Stack direction={{ base: 'column', md: 'row' }} gap='4'>
  <Flex direction='column' flex='1' gap='4'>
    {/* ... */}
  </Flex>
  <Flex direction='column' flex='1' gap='4'>
    <FormControl isInvalid={errors.priority}>
      <Select
```

```
        placeholder='Priority'
        {...register('priority', { required: 'Priority is required' })}
      >
        <option value='urgent'>Urgent</option>
        <option value='not urgent'>Not Urgent</option>
      </Select>
      <FormErrorMessage>
        {errors.priority && errors.priority.message}
      </FormErrorMessage>
    </FormControl>
    <FormControl isInvalid={errors.status}>
      <Select
        placeholder='Status'
        {...register('status', { required: 'Status is required' })}
      >
        <option value='open'>Open</option>
        <option value='done'>Done</option>
      </Select>
      <FormErrorMessage>
        {errors.status && errors.status.message}
      </FormErrorMessage>
    </FormControl>
  </Flex>
</Stack>
```

For the due date, you need to use the `Controller` component from React Hook Form and integrate it into React Datepicker.

Here's how you do it:

```
<FormControl>
  <DatePickerStyles />
  <Controller
    control={control}
    name='due'
    render={({ field }) => (
      <Input
        as={DatePicker}
        id='due'
        {...field}
        selected={field.value}
        showTimeSelect
        timeInputLabel='Time:'
        dateFormat='MM/dd/yyyy h:mm aa'
        placeholderText='Due Date (Optional)'
```

```
      />
    )}
  />
</FormControl>
```

Just like when integrating the `UploadAvatar` component, you use the `Controller` component and the `render` prop to define the input that will be rendered.

To use Chakra UI style for the `DatePicker` component, we use the `Input` component and render it as the `DatePicker` component using the `as` prop.

Now the form only needs a button for submission. Add it below the `FormControl` for the datepicker:

```
<Button
  type='submit'
  isLoading={isSubmitting}
  colorScheme='teal'
  textTransform='uppercase'
>
  Submit
</Button>
```

The form is now ready for use. You only need to import this form on the create task and update task pages.

Completing Create Task Page

On the `CreateTask.jsx` file, import the `TaskForm` and render it as follows:

```
import TaskForm from '../components/TaskForm';
import { Box, Heading } from '@chakra-ui/react';

export default function CreateTask() {
```

```
  return (
    <Box p='3' maxW='4xl' mx='auto'>
      <Heading
        as='h1'
        fontSize='3xl'
        fontWeight='semibold'
        textAlign='center'
        my='7'
      >
        Create a New Task
      </Heading>
      <TaskForm type='create' />
    </Box>
  );
}
```

Now if you visit the Create Task page, you should be able to create and save a new task.

Completing Update Task Page

To update a task, you need to fetch the task data from Express and pass it to the `TaskForm`.

Because we have a fetch, that means we need to create a skeleton for the Update Task page.

On the `_skeletons/` folder, create a new file named `UpdateTaskSkeleton.jsx` and write the code below:

```
import { Box, Stack, Skeleton, Flex } from '@chakra-ui/react';

export default function UpdateTaskSkeleton() {
  return (
    <Box p='3' maxW='4xl' mx='auto'>
      <Flex direction='row' gap='4' mt='7'>
        <Box width='50%'>
          <Stack gap='4'>
            <Skeleton height='20px' />
            <Skeleton height='20px' />
```

```
            <Skeleton height='20px' />
            <Skeleton height='20px' />
          </Stack>
        </Box>
        <Box width='50%'>
          <Stack gap='4'>
            <Skeleton height='20px' />
            <Skeleton height='20px' />
            <Skeleton height='20px' />
            <Skeleton height='20px' />
          </Stack>
        </Box>
      </Flex>
    </Box>
  );
}
```

Then, open the `UpdateTask.jsx` file and import both the `TaskForm` and the skeleton component:

```
import { API_BASE_URL } from '../util';
import { useState, useEffect } from 'react';
import { useParams } from 'react-router-dom';
import { Box, Heading } from '@chakra-ui/react';
import TaskForm from '../components/TaskForm';
import UpdateTaskSkeleton from '../_skeletons/UpdateTaskSkeleton';

export default function UpdateTask() {
  const [task, setTask] = useState();
  const { taskId } = useParams();

  useEffect(() => {
    const fetchTask = async () => {
      const res = await fetch(`${API_BASE_URL}/tasks/${taskId}`, {
        credentials: 'include',
      });
      const data = await res.json();
      setTask(data);
    };
    fetchTask();
  }, []);

  if (!task) {
    return <UpdateTaskSkeleton />;
  }
```

```
  return (
    <Box p='3' maxW='4xl' mx='auto'>
      <Heading
        as='h1'
        fontSize='3xl'
        fontWeight='semibold'
        textAlign='center'
        my='7'
      >
        Update Task
      </Heading>
      <TaskForm type='update' task={task} />
    </Box>
  );
}
```

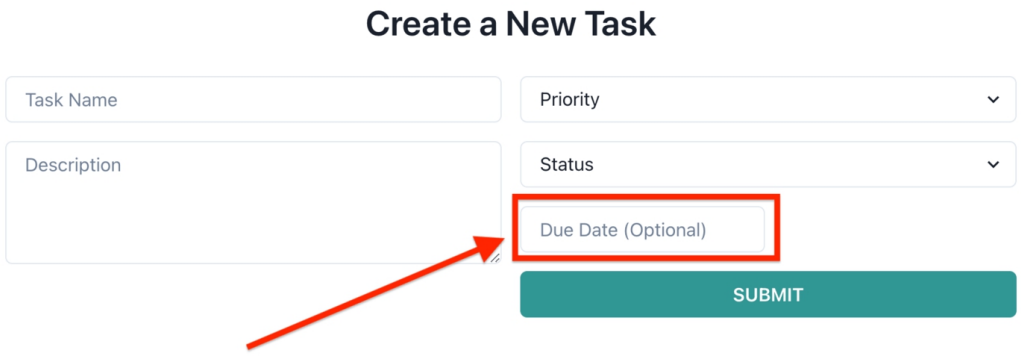
This `UpdateTask` component will take the task id from the URL parameter, and then fetch the data from Express.

It will show the skeleton component until the task data is received and set as the `task` state value.

Now you can update any existing task. Just click on the *Edit Task* link from the single task page.

Fixing DatePicker Style

If you look closely at the `DatePicker` component, you'll see that the input is a bit different than the others:



This is because React Datepicker style overrides Chakra UI style, which has 100% width.

To fix this problem, you need to add a custom style and adjust the `react-datepicker-wrapper` CSS style to 100% width.

In Chakra UI, you can pass a style using the `sx` prop as follows:

```
<FormControl
  sx={{
    '.react-datepicker-wrapper': {
      width: '100%',
    },
  }}
>
  {/* Controller */}
</FormControl>
```

Define the style on the `FormControl` component, and the `DatePicker` component will pick up that style.

Summary

The code added in this chapter is available at https://g.codewithnathan.com/mern-16

In this chapter, you've learned how to develop the create and update task features.

Since we've created the routes in Express before, we only need to provide the user interface for interacting with the routes.

By reusing the `TaskForm` component in the Create and Update Task pages, we greatly reduced the complexity of our application.

The web application is almost complete. There's only one more thing I want to show you in the next chapter. You're almost finished!

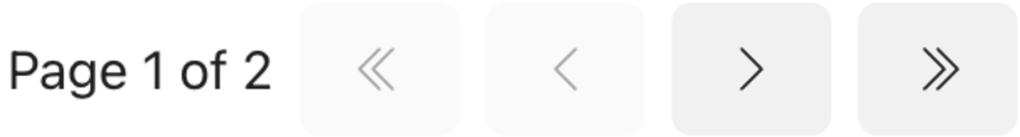
CHAPTER 17: TASKS PAGINATION, FILTERING, AND SORTING

Now that users can create, update, and delete tasks, we can move on to implement pagination, filter, and sort on the All Tasks page.

Pagination is simply dividing the data shown to the users into a set number of data at a time.

It's a common UI pattern seen in websites where a long list of items such as search results, articles, products, or user comments.

The pagination we want to implement looks like this:



Let me show you how to create pagination as a React component

Creating Pagination Component

The pagination will be a separate component that you can plug into any page that requires pagination.

On the `components/` folder, let's add a new file called `pagination.jsx` and import the modules required for the component:

```
import { Button, Flex, Text } from '@chakra-ui/react';
import {
  BsChevronDoubleLeft,
  BsChevronDoubleRight,
  BsChevronLeft,
  BsChevronRight,
} from 'react-icons/bs';
import { useSearchParams } from 'react-router-dom';
```

The `useSearchParams` hook is used to append the `page` query string to the URL, while the other components are used for creating the interface.

Next, create the `Pagination` component that takes three props as follows:

```
export default function Pagination({ itemCount, pageSize, currentPage }) {

}
```

The `itemCount` prop is used to let the component know the total number of items to divide.

The `pageSize` prop defines how many items a page is showing.

And the `currentPage` prop defines the page number that's currently shown.

Inside the component, initialize the `useSearchParams` hook and calculate the total page count like this:

```
const [searchParams, setSearchParams] = useSearchParams();

const pageCount = Math.ceil(itemCount / pageSize);
if (pageCount <= 1){
  return null;
}
```

Here, we calculate the `pageCount` by dividing `itemCount` by `pageSize`. Then we round up the number using the `Math.ceil()` method.

When the `pageCount` is one or less, there's no need to render a pagination, so we return `null`.

Next, let's create a function to append a `page` query string to the URL:

```
const changePage = page => {
  searchParams.set('page', page.toString());
  setSearchParams(searchParams);
};
```

This `changePage()` function sets the `page` query string to the URL.

Components that have pagination implemented should use this `page` value and change the data shown on the screen.

Below the function, create the component UI as follows:

```
return (
  <Flex align='center' gap='2' mt='2'>
    <Text size='2'>
      Page {currentPage} of {pageCount}
    </Text>
```

```
  </Flex>
);
```

This `return` statement simply shows the current page and the total page count.

To allow users to change the page, you need to add 4 buttons that will be used to move the page forward or backward as follows:

```
<Button isDisabled={currentPage === 1} onClick={() => changePage(1)}>
  <BsChevronDoubleLeft />
</Button>
<Button
  isDisabled={currentPage === 1}
  onClick={() => changePage(currentPage - 1)}
>
  <BsChevronLeft />
</Button>
<Button
  isDisabled={currentPage === pageCount}
  onClick={() => changePage(currentPage + 1)}
>
  <BsChevronRight />
</Button>
<Button
  isDisabled={currentPage === pageCount}
  onClick={() => changePage(pageCount)}
>
  <BsChevronDoubleRight />
</Button>
```

The pagination component is now complete. You can test this component by adding it to the tasks page just below the `TableContainer` closing tag:

```
import Pagination from '../components/Pagination';

    </TableContainer>
```

```
  <Pagination itemCount={100} pageSize={5} currentPage={3} />
</Box>
```

Save the changes and head to the tasks page. You should see the pagination shown as follows:

When you press the buttons, the ?page= query string shown in the URL will change, but nothing happens because the tasks page isn't processing that query string yet.

Integrating Pagination to Tasks Page

On the tasks page, you need to get the query string using the `useSearchParams` hook, then send it to Express to control the data returned by the server.

Let's add the `useSearchParams` hook to `Tasks.jsx` as follows:

```
import { Link, useSearchParams } from 'react-router-dom';
import Pagination from '../components/Pagination';

// Inside Tasks() component:
export default function Tasks() {
  const { user } = useUser();
  const [tasks, setTasks] = useState();

  const [searchParams, setSearchParams] = useSearchParams();
```

```
  const [itemCount, setItemCount] = useState(0);
  const page = parseInt(searchParams.get('page')) || 1;
}
```

Here, we initialize the `useSearchParams` hook, add a state for the `itemCount`, and parse the `page` query string as a `page` variable using the `parseInt()` function.

Inside the `useEffect` hook used for fetching tasks data, you need to create a query string and pass it to the `fetch()` function as follows:

```
useEffect(() => {
  const fetchTasks = async () => {
    const query = searchParams.size ? '?' + searchParams.toString() : '';
    const res = await fetch(`${API_BASE_URL}/tasks/user/${user._id}${query}`,
{
      credentials: 'include',
    });
    const { tasks, taskCount } = await res.json();
    setTasks(tasks);
    setItemCount(taskCount);
  };
  fetchTasks();
}, [searchParams]);
```

Now, the `query` variable will contain the query string parameter as a string (for example, `?=page=2`)

We attach this `query` to the end of the `fetch()` URL so that Express can use this data when querying the database.

Express will also return the item count later, so we unpack `taskCount` data from the response and set it as the value of `itemCount` state with `setItemCount()`.

Also, the dependency array for the hook is changed to `searchParams` so that React will fetch new data whenever the query string changes.

The last step is to update the `Pagination` component props to use available data:

```
<Pagination itemCount={itemCount} pageSize={4} currentPage={page} />
```

The frontend changes are finished. Now you need to update the backend part.

Updating Express for Pagination

On the `task.controller.js` page, you can access the query string passed from React in the `req.query` object.

Inside the `getTasksByUser()` function, unpack the page query as follows:

```
const page = parseInt(req.query.page) || 1;
const pageSize = 4;
```

You need to determine the `pageSize` manually, and it must be the same number as the one in React.

Next, the MongoDB `collection.find()` method needs to be updated:

```
const tasks = await collection
  .find(query)
  .limit(pageSize)
  .skip((page - 1) * pageSize)
  .toArray();
```

```
const taskCount = await collection.count(query);
res.status(200).json({ tasks, taskCount });
```

Here, we add the `limit()` method to limit the returned value by `pageSize`.

We also need to skip tasks that have been shown on the previous page by calling the `skip()` method.

The `skip()` method argument is the amount of tasks to skip when running the query. To calculate the right number, we need to decrease the `page` value by one and multiply it by `pageSize`.

Here's how the calculation works:

```
page 1 = 0 * 4 = 0
page 2 = 1 * 4 = 4
page 3 = 2 * 4 = 8
```

This way, you ensure that previously shown data won't be shown again.

The amount of tasks available is also calculated by calling the `collection.count()` method.

The `tasks` and `taskCount` values are sent as a response.

With that, the pagination for the Tasks page is now completed. When a user has more than 4 tasks, the pagination will be shown and the user can navigate to other pages.

You can test this by adding a few more tasks to the existing user as well.

Filtering Tasks By Status

Back to the Tasks page, we still have a status dropdown that's not yet functional.

To make this dropdown work, you need to add the dropdown value to the query string. Add the function below inside the `Tasks` component:

```
const handleStatusFilter = e => {
  const value = e.target.value;
  if (value){
    searchParams.set('status', value);
  }
  else {
    searchParams.delete('status');
  }
  setSearchParams(searchParams);
};
```

This `handleStatusFilter()` function will set a `status` parameter to the query string. When the user wants to see tasks with all statuses, the parameter will be deleted.

Next, call this function on the `Select` component:

```
<Select placeholder='All' onChange={handleStatusFilter}>
  <option value='open'>Open</option>
  <option value='done'>Done</option>
</Select>
```

Now whenever you select one of the options, the `status` parameter will be added to the URL.

Next, you need to update the backend to take into account the status query.

Inside `getTasksByUser` function, unpack the `status` query as follows:

```
const query = { owner: new ObjectId(req.params.id) };
const { status } = req.query;
if (status) {
  query['status'] = status;
}
```

This will cause the `collection.find()` method to filter the returned tasks data according to the `status` value.

Now the Select dropdown should work. You can try creating several tasks with different statuses to see the filter in action.

Sorting Tasks

The final task you need to do is to sort the tasks data by a specific attribute.

For this application, we want to enable sorting by name, priority, status, and due date.

Since you've done the filter tasks feature, this one will be easy.

The first thing we want to do is to make the column headers clickable.

You need to create a function that adds another parameter to the URL. Let's name this function `handleOrderBy`:

```
const handleOrderBy = value => {
  searchParams.set('orderBy', value);
  setSearchParams(searchParams);
};
```

This function simply sets the `orderBy` parameter to the query string.

On each table header `<Th>` element, you need to wrap the header text in a `Flex` component as follows:

```
<Th>
  <Flex
    onClick={() => handleOrderBy('name')}
    cursor='pointer'
  >
    Task
  </Flex>
</Th>
<Th>
  <Flex
    onClick={() => handleOrderBy('priority')}
    cursor='pointer'
  >
    Priority
  </Flex>
</Th>
<Th>
  <Flex
    onClick={() => handleOrderBy('status')}
    cursor='pointer'
  >
    Status
  </Flex>
</Th>
<Th>
  <Flex
    onClick={() => handleOrderBy('due')}
    cursor='pointer'
  >
    Due Date
  </Flex>
</Th>
```

On the `Flex` component, we add the `cursor` prop to show the user that the header text can be clicked.

The `onClick` function will then set the `orderBy` parameter to the URL.

To give visual feedback to the user, we should add an icon next to the header text.

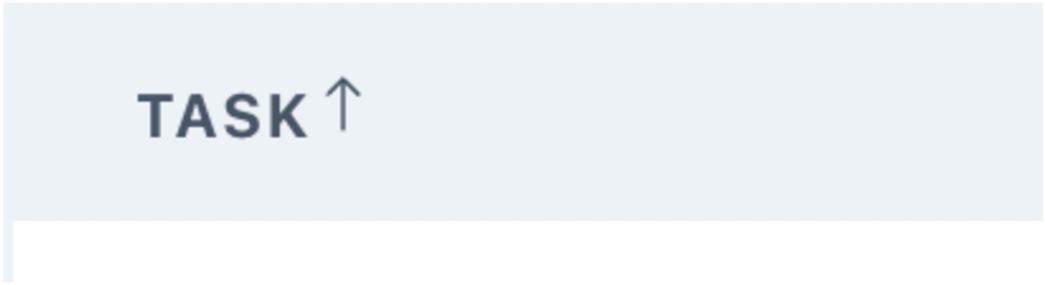
Import the icon from `react-icons` module as follows:

```
import { BsArrowUp } from 'react-icons/bs';
```

Then show the icon only when the `handleOrderBy()` argument for the header matches the `orderBy` parameter:

```
<Flex
  onClick={() => handleOrderBy('name')}
  cursor='pointer'
>
  Task {searchParams.get('orderBy') === 'name' && <BsArrowUp />}
</Flex>
```

Now if you click on the Task header, you'll see the icon appears, but it's not vertically aligned:



To make the icon centered vertically, you need to add the `alignItems='center'` prop to the `Flex` component:

```
<Flex
  onClick={() => handleOrderBy('name')}
```

```
  cursor='pointer'
  alignItems='center'
>
  {/* ... */}
</Flex>
```

Now the icon position is more aligned to the header.

Repeat this code for the remaining header:

```
<Flex
  onClick={() => handleOrderBy('priority')}
  cursor='pointer'
  alignItems='center'
>
  Priority {searchParams.get('orderBy') === 'priority' && <BsArrowUp />}
</Flex>

<Flex
  onClick={() => handleOrderBy('status')}
  cursor='pointer'
  alignItems='center'
>
  Status {searchParams.get('orderBy') === 'status' && <BsArrowUp />}
</Flex>

<Flex
  onClick={() => handleOrderBy('due')}
  cursor='pointer'
  alignItems='center'
>
  Due Date {searchParams.get('orderBy') === 'due' && <BsArrowUp />}
</Flex>
```

The frontend is now completed.

The last thing you need to do is to change the `getTasksByUser()` function in `task.controller.js`.

Get the `orderBy` query string from the `req.query` object, and chain call the `sort()` method when calling the `collection.find()`

method:

```
const { status, orderBy } = req.query;
const sort = orderBy ? { [orderBy]: 1 } : {};

// Initialize other variables...

// Then:
const tasks = await collection
  .find(query)
  .sort(sort)
  .limit(pageSize)
  .skip((page - 1) * pageSize)
  .toArray();
```

And now you can sort tasks by clicking on the header table. Great work!

Summary

The code added in this chapter is available at https://g.codewithnathan.com/mern-17

In this chapter, you've added the pagination, filter, and sorting features to the Tasks page.

These functionalities are some of the most common features you'll find in modern applications, so knowing how to build them from scratch is a great way to improve your coding skills.

CHAPTER 18: DEPLOYING MERN APPLICATION

Now that the application is completed, it's time to deploy the application to production and make it accessible from the internet.

But before deploying the application, there are some small changes you need to do.

Preparing Application for Deployment

First, you need to update Express to at least version 4.19 or later. If you build the project from scratch, you should already be using the latest version.

But if you download the code from the repo, you need to update Express by running `npm install express` from the `server/` folder.

The reason for this is that there's a new policy regarding placing cookies on the browser, and the previous Express version doesn't comply with this policy.

Once Express is updated, open the `server.js` file and adjust the `PORT` constant as follows:

```
const PORT = process.env.PORT || 3000;
```

This is because Railway, the platform we're going to use to deploy the backend, requires the application to be listening at port 3000.

You also need to specify the localhost address '0.0.0.0' when calling the `app.listen()` method as follows:

```
app.listen(PORT, '0.0.0.0', () => {
  console.log(`Server listening on port ${PORT}`);
});
```

Then, create a new route just above the wild card * route for the home URL as follows:

```
app.get('/', (req, res) => {
  res.status(200).json({ message: 'Welcome to Taskly API' });
});
```

Instead of using `use()`, we use `get()` so that only GET requests can access this route. If we show 'not found' on the Home page, it can cause misunderstanding.

The last change you need to make is in the `auth.controller.js` file. Add three more options when sending a cookie using `res.cookie()` as follows:

```
res
.cookie('taskly_token', token, {
  httpOnly: true,
  sameSite: 'None',
```

```
    secure: true,
    partitioned: true,
  })
  .status(200)
  .json(rest);
```

The `sameSite` option is set to 'None' to enable cookie from a different domain. The `secure` option means the cookie is sent only from HTTPS protocol.

The `partitioned` option is a new attribute that you need to add to avoid getting blocked by browsers. This is a new requirement that needs to be satisfied by the browser policy update.

This is also why you need to update Express, because Express version 4.18 doesn't support the `partitioned` option yet.

Once you add these options in `signin()` and `signup()` functions, save the changes.

Pushing Code to GitHub

Deploying an application requires you to grant access to the project files and folders. GitHub is a platform that you can use to host and share your software project.

Head over to https://github.com/ and login or register for a new account if you don't have one already.

From the dashboard, create a new repository by clicking + New on the left sidebar, or the + sign on the right side of the navigation bar:

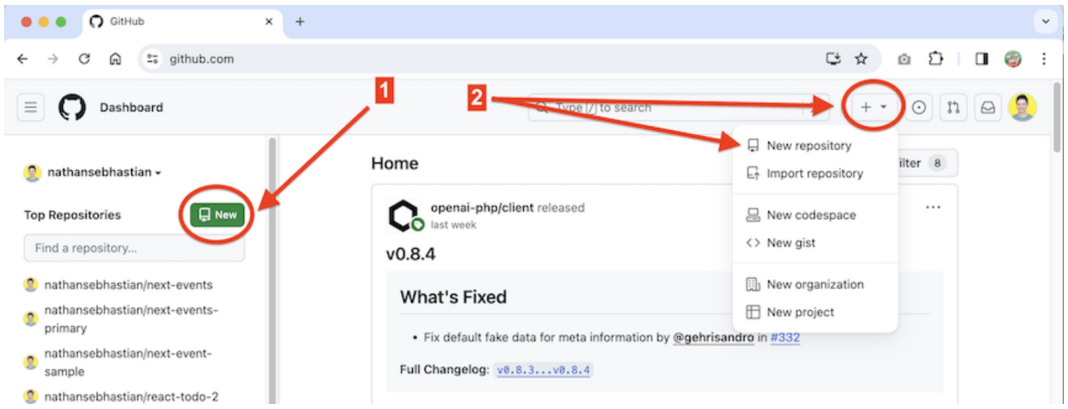
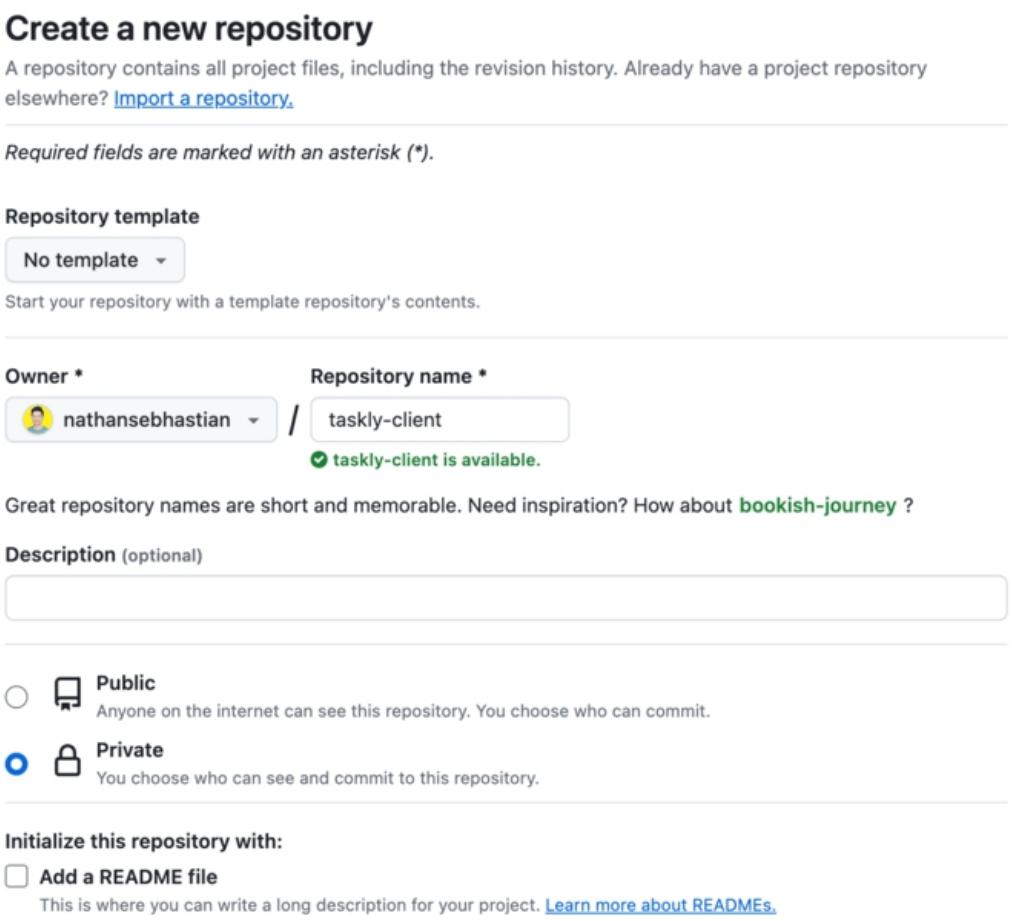


Figure 11. Two Ways To Create a Repository in GitHub

A repository (or repo) is a storage space used to store software project files. Because we have 2 separate projects for Taskly, we're going to need 2 separate repositories.

Let's upload the `client` project first for the frontend.

In the *Create a Repository* page, fill in the details of your project. The only required detail is the repository name:



You can make the repository public if you want this project as a part of your portfolio, or you can make it private.

Once a new repo is created, you will be given instructions on how to push your files into the repository.

You need to follow the instruction for pushing an existing repo:

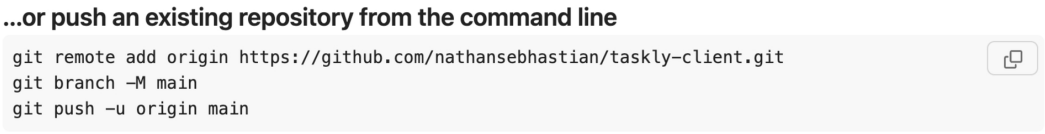


Figure 12. How to Push Existing Repo to GitHub

Now you need to create a repository for your project. Open the command line on the `client/` folder, then run the `git init` command:

```
git init
```

This will turn your project into a local repository. Add all project files into this local repo by running the `git add .` command:

```
git add .
```

Changes added to the repo aren't permanent until you run the `git commit` command. Commit the changes as shown below:

```
git commit -m 'Ready for Deployment'
```

The `-m` option is used to add a message for the commit. Usually, you summarize the changes committed to the repository as the message.

Now you need to push this existing repository to GitHub. You can do so by following the GitHub instructions:

```
git remote add origin <URL>
git branch -M main
git push -u origin main
```

You might be asked to enter your GitHub username and password when running the `git push` command.

Once the push is complete, refresh the GitHub repo page on the browser, and you should see your project files and folders

there.

This means our application is already pushed (uploaded) to a remote repository hosted on GitHub.

You need to do the same thing with the `server` project. Create a new repository named `taskly-server` on Github:

Then open the command line on the `server/` folder to initialize git and push the code to the repo:

```
git init
git add .
git commit -m 'Ready for Deployment'
git remote add origin <URL>
git branch -M main
git push -u origin main
```

With that, both projects are now uploaded to GitHub.

One of the advantages of using the MERN stack is that you can deploy the frontend and backend projects to different hosting or cloud platforms.

I will show you how to deploy the backend first.

Deploying Express Application to Railway

Railway is a cloud platform that you can use for building and deploying a web application.

The platform offers a free tier that you can use to deploy the Express application we've created.

You can sign up for an account at https://railway.app and agree to the privacy policy and fair use agreement.

Here are the next instructions:

1. Click on *Create New Project* to start deploying your application.
2. Select *Deploy from GitHub repo*
3. Select *Configure GitHub App* and allow Railway to access your repos
4. After enabling access, you will be taken back to the Railway page
5. Select the Taskly server repo to deploy it.
6. Click on + *Add variables*

Railway will deploy your application, and you'll be taken into the Variables setting.

Here, you need to click on the *Raw Editor* link to open a text editor.

Open your `server/.env` file, then copy and paste it to the Railway editor. Click Save or Update Variables.

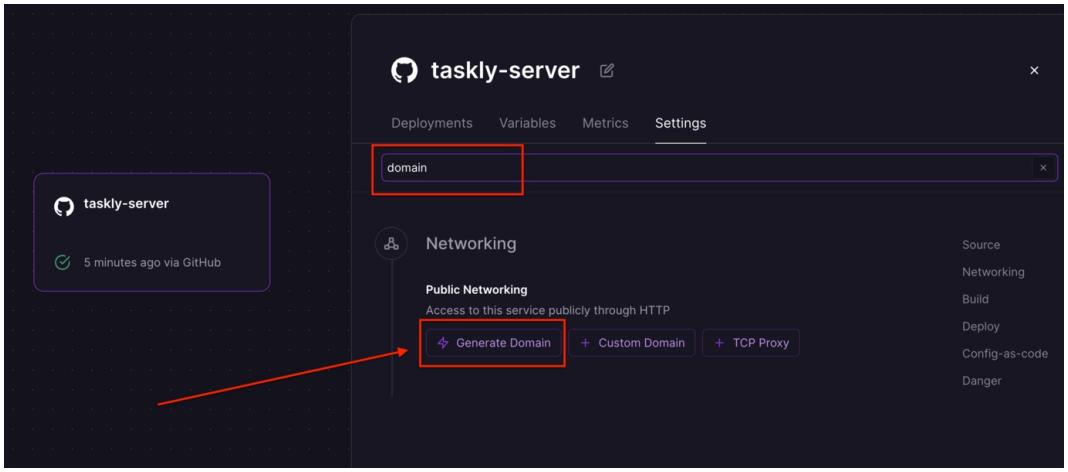
Railway should automatically redeploy the application for you.

The next step is to allow HTTP access by enabling the public network.

On the Railway menu, click the *Settings* page, and type 'domain' into the filter input. You should see the *Public Networking*

option.

Click the *Generate Domain* button as shown below:



Railway will assign a domain for your application and enable access from that domain URL.

With that, the backend deployment is completed. You can try to access the backend using the given domain.

There's a little change that we need to make after deploying the frontend later, so please make sure you keep the Railway tab open as you continue to the next section.

Deploying React Application to Vercel

Vercel is a cloud hosting company that you can use to build and deploy React applications.

You can sign up for a free account at https://vercel.com, then select Create New Project on the Dashboard page:

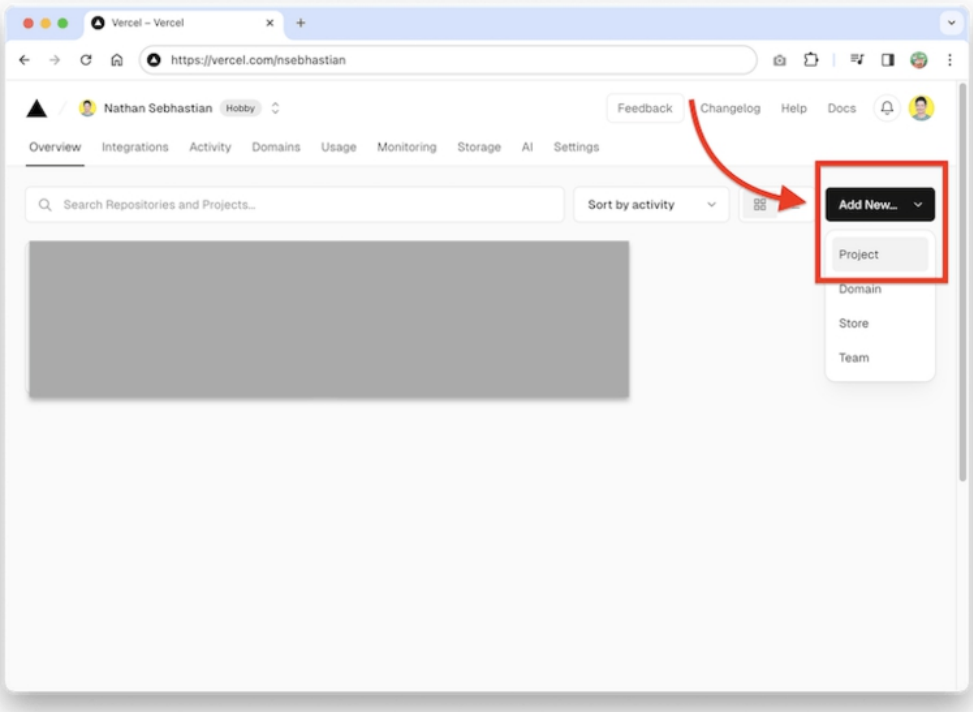


Figure 13. Vercel Create New Project

Next, you will be asked to provide the project that you want to build and deploy.

Since the project is uploaded to GitHub, you can select *Continue With Github* as shown below:

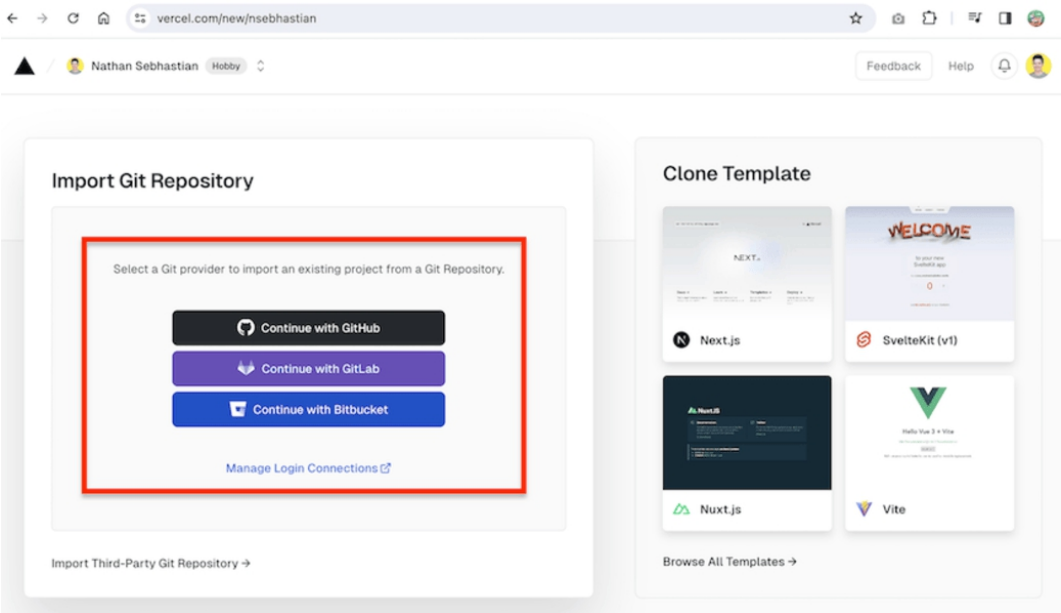


Figure 14. Vercel Import Repository Menu

Once you grant access to your GitHub account, select the project to deploy:

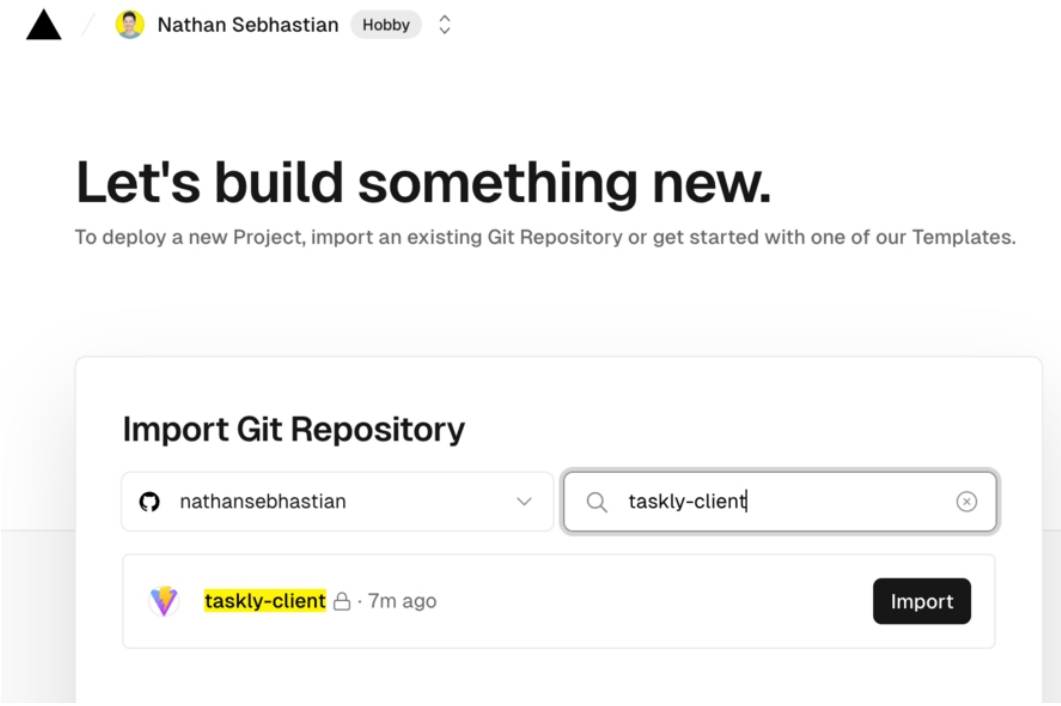
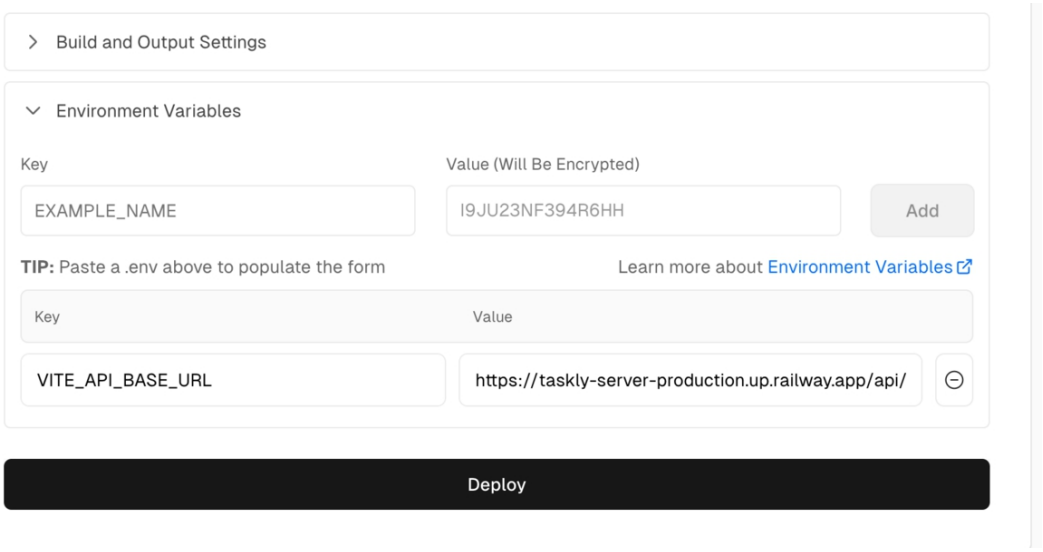


Figure 15. Vercel GitHub Import

Then, you will be taken to the project setup page. Open the *Environment Variable* menu to insert the `VITE_API_BASE_URL` variable.

Here, you need to pass the domain given by Railway for the backend, and include '/api/v1' to the URL as shown below:



Next, click the *Deploy* button. Vercel will build the application for you.

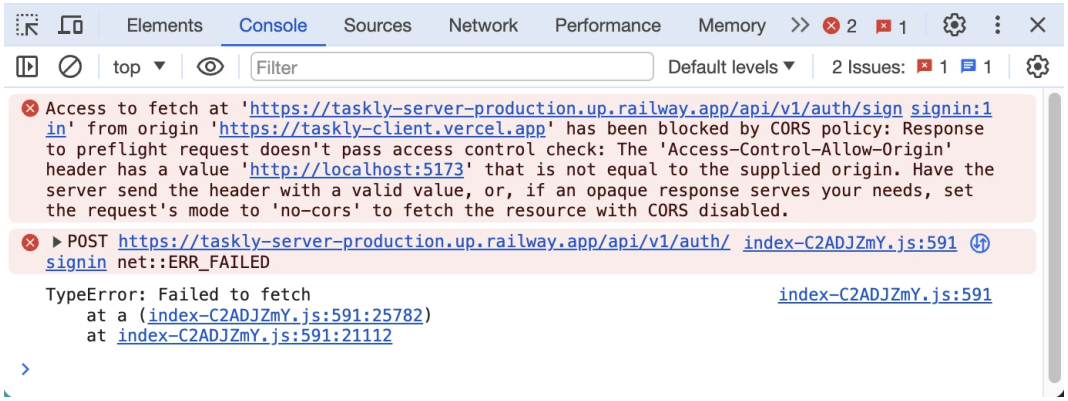
When the build is done, you will be directed to the success page.

You can click on the image preview to open your application. The application will be assigned a free `.vercel.app` domain.

Now that the application is deployed, you can try signing up for an account on the live domain.

But if you try to sign in or sign up, you'll notice that the application encounters an error.

Opening the browser console will show you more details:

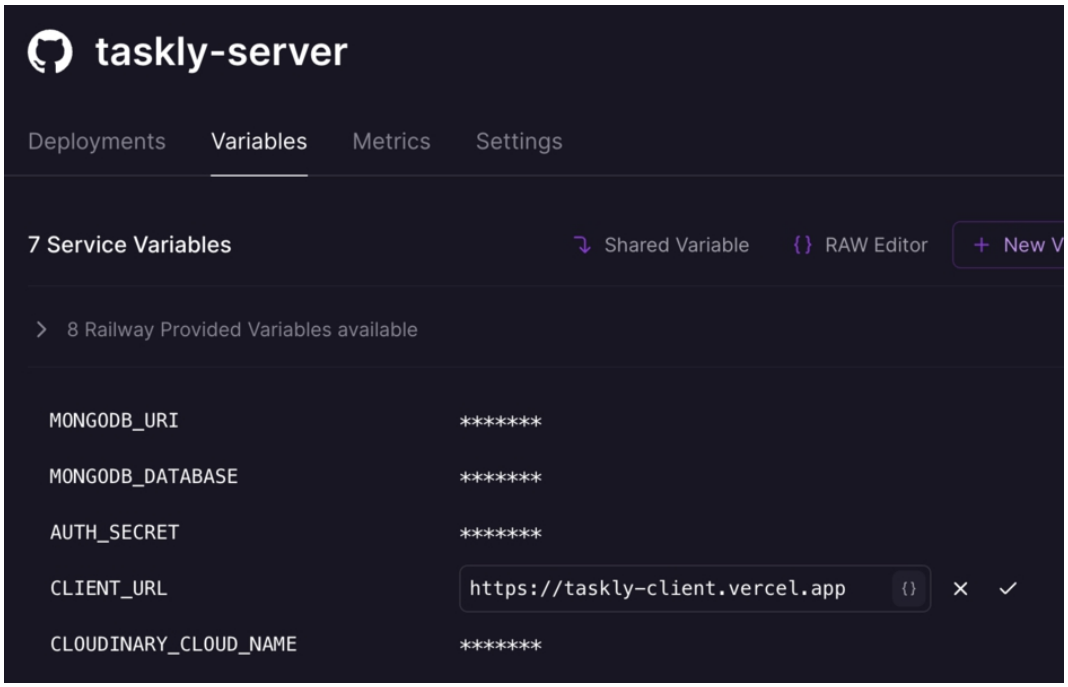


This error happens because of the CORS policy, since the domain given by Vercel is different from the one we use in development.

Changing the CLIENT_URL Value on Railway

Back to the Railway application, you need to change the `CLIENT_URL` variable value to the domain assigned in Vercel.

Open the *Variables* setting again, and change the variable's value as follows:



Make sure that you include the domain name correctly, or the CORS problem won't disappear.

After changing the variable's value, Railway will show a toast notification and tell you to deploy the changes.

Click on the *Deploy* button and wait until Railway finishes the deployment process.

Now you should be able to sign in from the frontend.

Summary

Changes added in this chapter can be found at https://g.codewithnathan.com/mern-18

Well done! You've completed the lessons in this book, built a full stack application using the MERN stack, and deployed it to a production environment.

All of this wasn’t easy, but you made it. Give yourself a pat on the back, because you just passed a very important milestone in your knowledge gain. I’m so proud of you!

WRAPPING UP

Congratulations on finishing the book!

You have learned how to use the MERN stack to build a complex full-stack web application, and then deploy it to production.

I hope you have fun learning the MERN stack as I did writing this book.

Though the book ends here, your journey to master the MERN stack is not over yet. You can practice by building more applications using the technology.

The Next Step

Now that you have learned the MERN stack, you might want to continue your learning journey.

For example, you can learn how to add TypeScript for static type checking, use PostgreSQL as the database, Prisma ORM as the bridge between Express and MongoDB, and use Radix for the UI component library.

Another way to develop a full-stack React application is to use the Next.js framework, a modern web framework for building

production-grade web applications.

I have a book on Next.js that you can get at https://codewithnathan.com/beginning-nextjs

I'm currently planning to write more books on web development topics, so you might consider subscribing to my newsletter to know when I release a new book at https://codewithnathan.com/newsletter

I also have a 7-day free email course on Mindset of The Successful Software Developer which is free at https://sebhastian.gumroad.com/l/mindset

Each email unveils a slice of wisdom that I got after working as a web developer and programmer for 8+ years in the tech industry, from startups to mega-corporations. It offers a fresh perspective on what it means to be a happy and successful software developer.

If you like the book, I would appreciate it if you could leave me a review on Amazon because it would help others to find this book.

If there are some things you wish to tell me about this book, feel free to email me at nathan@codewithnathan.com. I'm very open to feedback and eager to improve my book so that it can be a great resource for people learning to code.

If you didn't like the book, or if you feel that I should have covered certain additional topics, please let me know in the email.

I wish you luck on your software developer career onward, and I'll see you again in other books.

Until next time!

ABOUT THE AUTHOR

Nathan Sebhastian is a senior software developer with 8+ years of experience in developing web and mobile applications.

He is passionate about making technology education accessible for everyone and has taught online since 2018

OceanofPDF.com

MERN Stack Web Development For Beginners

A Step-By-Step Guide to Build a Full Stack Web Application With React, Express, Node.js, and MongoDB

By Nathan Sebhastian

https://codewithnathan.com
