



ULTIMATE

Full-Stack Web Development with MERN

Design, Build, Test and Deploy
Production-Grade Web Applications
with MongoDB, Express, React
and NodeJS



Nabendu Biswas



ULTIMATE

Full-Stack Web Development with MERN

Design, Build, Test and Deploy
Production-Grade Web Applications
with MongoDB, Express, React
and NodeJS



Nabendu Biswas

**Ultimate Full-Stack
Web Development
with MERN**



Design, Build, Test and Deploy
Production-Grade Web Applications
with MongoDB, Express,
React and NodeJS



Nabendu Biswas



www.orangeava.com

Copyright © 2023 Orange Education Pvt Ltd, AVA™

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Orange Education Pvt Ltd or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Orange Education Pvt Ltd has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, Orange Education Pvt Ltd cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

First published: December 2023

Published by: Orange Education Pvt Ltd, AVA™

Address: 9, Daryaganj, Delhi, 110002

ISBN: 978-81-19416-42-4

www.orangeava.com

Dedicated To

My wife Shikha and My Son Hriday

About the Author

Nabendu Biswas is a full-stack JavaScript developer and has been working in the IT industry for the past 19 years with some of the world's top development firms and investment banks, including J.P. Morgan, Oracle, and L&T Infotech.

He is a passionate tech blogger and YouTuber with an active following on both platforms, where he teaches everyone to be a web developer. Currently, he works as an Architect in an IT firm in Bhopal. Additionally, he is the author of seven books focusing on topics such as Gatsby, MERN, TypeScript, GraphQL, and React Firebase, all of which are available on Amazon.

About the Technical Reviewer

Samarth is a seasoned Full Stack Developer with over 5 years of experience in software development, specializing in multiple technology stacks, including the popular MERN (MongoDB, Express.js, React, Node.js) stack. His career journey has been marked by a diverse range of projects across industries such as Community-based apps, Legal, Crypto/Blockchain, Ed-Tech, HealthCare, and IOT.

Samarth is widely recognized for his expertise in creating solutions to real-world problems through innovative software development, with a specific focus on JavaScript and React. His commitment to improving the world through technology is a driving force in his work, and his comprehensive experience in diverse industries demonstrates his ability to deliver value in a variety of contexts.

Outside of software development, Samarth's interests include science and philosophy, reflecting his diverse set of passions and interests.

Acknowledgements

There are a few people I want to thank for the continued and ongoing support during the writing of this book. First and foremost, I would like to thank my wife for continuously encouraging me to write the book. I could have never completed this book without her support.

I am also grateful to the courses and the companies that supported me throughout the learning process of web development. Thank you for the all hidden support you provided.

My gratitude also goes to the team at Orange AVA for being supportive enough and providing me with quite a long time to finish and publish the book.

Preface

This book covers all the top technologies in the JavaScript ecosystem: MongoDB, ExpressJS, ReactJS and NodeJS. With these technologies, we can create a completely functional web application that includes frontend, backend and database.

We will build a fairly large production-ready app in this book. Here, we will first learn to set up the frontend with React, then create the backend APIs with NodeJS and Express. The backend will be connected to our database which is MongoDB. We will learn to use authentication in our project by implementing JWT. We will also learn to test both frontend and backend using the Unit testing framework of Jest. Finally, we will learn how to deploy both frontend and backend with ease to use free apps.

After going through the book, the readers will be able to create any MERN app with ease and can also modify the project in the book to create other apps.

Chapter 1: This book will begin with an introduction to all technologies related to MERN (MongoDB, Express, React and NodeJS) followed by a complete project overview of the app, which we are going to build in this book. We will also cover the backend setup required for the same.

Chapter 2: This chapter covers the creation of an Express app. We will create our first routes and controllers in this chapter. We will also learn about error handling.

Chapter 3: This chapter covers connecting to the MongoDB database and creating a model in our backend app. We will also start creating routes for our task app.

Chapter 4: This chapter deals with the implementation of JWT authentication and also the implementation of hashing passwords.

Chapter 5: This chapter covers the implementation of Auth middleware and protecting routes. Here, we will learn how to allow only authorized users to access certain routes.

Chapter 6: This chapter deals with the creation of frontend in ReactJS. We will also cover the setup for React Router version 6 in the project, which is used for navigation in React. We will also create the Register and Login pages on the frontend.

Chapter 7: This chapter covers the implementation of Redux, with the latest slice in the frontend. We will also set up a reducer for the registration form.

Chapter 8: This chapter deals with the login and logout functionalities with Redux.

Chapter 9: This chapter covers the creation of a dashboard in ReactJS. We will also learn how to restrict the dashboard to authorized users.

Chapter 10: This chapter covers the implementation of thunk in tasks and

also fetching tasks from the server. It also completes our app, in which we display and delete tasks.

Chapter 11: This chapter covers the testing of the frontend. We will test the frontend created in React using Jest and React Testing Library.

Chapter 12: This chapter deals with the testing of the backend. We will test the backend created in NodeJS using Jest.

Chapter 13: This chapter covers the deployment of both frontend and backend using freely available services.

Downloading the code bundles and colored images

Please follow the link to download the

Code Bundles of the book:

<https://github.com/ava-orange-education/Ultimate-Full-Stack-Web-Development-with-MERN>

The code bundles and images of the book are also hosted on

<https://rebrand.ly/bde02d>

In case there's an update to the code, it will be updated on the existing GitHub repository.

Errata

We take immense pride in our work at Orange Education Pvt Ltd and follow best practices to ensure the accuracy of our content to provide an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any

unforeseen errors, please write to us at :

errata@orangeava.com

Your support, suggestions, and feedback are highly appreciated.

DID YOU KNOW

Did you know that Orange Education Pvt Ltd offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.orangeava.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: info@orangeava.com for more details.

At www.orangeava.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on AVA™ Books and eBooks.

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at info@orangeava.com with a link to the material.

ARE YOU INTERESTED IN AUTHORING WITH US?

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please write to us at business@orangeava.com. We are on a journey to help developers and tech professionals to gain insights on the present technological advancements and innovations happening across the globe and build a community that believes Knowledge is best acquired by

sharing and learning with others. Please reach out to us to learn what our audience demands and how you can be part of this educational reform. We also welcome ideas from tech experts and help them build learning and development content for their domains.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at Orange Education would love to know what you think about our products, and our authors can learn from your feedback. Thank you!

For more information about Orange Education, please visit
www.orangeava.com.

Table of Contents

[1. Getting Started with MERN and Setup](#)

[Introduction](#)

[Structure](#)

[About MERN](#)

[MongoDB](#)

[Express](#)

[ReactJS](#)

[NodeJS](#)

[Other Popular stacks beyond MERN](#)

[REST APIs](#)

[POST](#)

[GET](#)

[PUT](#)

[DELETE](#)

[Alternatives to REST API](#)

[Complete Project Overview](#)

[Setting up a NodeJS Application](#)

[Project Folder Structure](#)

[Package installation](#)

[Running the project](#)

[Conclusion](#)

[2. Starting a NodeJS App](#)

[Introduction](#)

[Structure](#)

[Using Express](#)

[Creating Routes](#)

[Initial Routes](#)

[All Routes](#)

[Testing through Postman](#)

[Creating Controllers](#)

[Using JSON](#)

[Error Handler](#)

[Conclusion](#)

[3. MongoDB Connection and Models](#)

[Introduction](#)

[Structure](#)

[MongoDB Database](#)

[Basic Setup](#)

[Connecting through Mongoose](#)

[Model Creation](#)

[Creating Routes](#)

[POST Route](#)

[PUT Route](#)

[DELETE Route](#)

[Conclusion](#)

[Points to remember](#)

4. JWT Authentication and Hashing Password

[Introduction](#)

[Structure](#)

[User Model and Controller](#)

[Register User with Hashed Password](#)

[Login User](#)

[Understanding JWT](#)

[Using JWT](#)

[Conclusion](#)

[Points to Remember](#)

5. Auth Middleware and Protecting Routes

Introduction

Structure

Creating Auth middleware

Understanding Protect Routes

Protecting Task Routes

GET and POST Route

PUT and DELETE Route

Conclusion

Points to remember

Multiple Choice Questions

Answers

6. Creating Frontend and React Router

Introduction

Structure

Creating Frontend with ReactJS

Basic Project Setup

React Router Setup

Creating Components and Pages

Header Component

[Register Page](#)

[Login Page](#)

[Conclusion](#)

[Points to remember](#)

7. Redux Setup with Slice

[Introduction](#)

[Structure](#)

[Redux setup with a toolkit](#)

[Auth service and slice](#)

[Registration form hook up](#)

[Testing user registration](#)

[Conclusion](#)

[Points to remember](#)

8. Login and Logout Functionalities

[Introduction](#)

[Structure](#)

[Implementing Logout](#)

[Implementing Login](#)

[Login form hook up](#)

[Testing user login](#)

[Conclusion](#)

[Points to remember](#)

9. Dashboard Creation and Task Form

[Introduction](#)

[Structure](#)

[Changing Dashboard Logic](#)

[Creating Task Slice](#)

[Creating Task Form](#)

[Conclusion](#)

[Points to Remember](#)

[Multiple Choice Questions](#)

[Answers](#)

10. Using Thunk and Completing App

[Introduction](#)

[Structure](#)

[Creating task with Async Thunk](#)

[Fetching tasks from server](#)

[Displaying the tasks](#)

[Deleting tasks](#)

[Conclusion](#)

[Points to remember](#)

[11. Frontend Testing](#)

[Introduction](#)

[Structure](#)

[Setup testing in Frontend](#)

[Task Slice and Service Test with Jest](#)

[Testing with React Testing Library](#)

[Configuring and Checking Coverage](#)

[Conclusion](#)

[Points to remember](#)

[12. Backend Testing](#)

[Introduction](#)

[Structure](#)

[Setting up testing in the backend](#)

[Testing registerUser controller](#)

[Testing getTasks and setTask controller](#)

[Testing updateTask controller](#)

[Conclusion](#)

[Points to remember](#)

[13. Deployment](#)

[Introduction](#)

[Structure](#)

[Frontend code repo creation](#)

[Backend code repo creation](#)

[Backend deployment in Back4app](#)

[Backend integration with frontend](#)

[Frontend deployment in Netlify](#)

[Fixing CORS errors and final deployments](#)

[Conclusion](#)

[Points to remember](#)

[Index](#)

CHAPTER 1

Getting Started with MERN and Setup

Introduction

This chapter talks about the basic principles of MERN and REST API. We will talk about the technology stack behind MERN and also the way REST API works. Here, we will also do a complete project overview. We are going to learn about the complete project, which we are going to create in this book. Beside this, we are also going to do the basic setup of the backend part of the application. Here, we are going to set up the NodeJS application.

Structure

In this chapter, we are going to discuss the following topics:

MERN

REST APIs

Complete Project Overview

Setting up a NodeJS Application

About MERN

MERN stands for MongoDB, Express, React, and NodeJS. It is one of the most popular stacks for creating full-stack apps. Now, full-stack apps are fully functional apps in which we create both the frontend and the backend.

Traditionally, creating the backend and frontend are tasks for two different teams. But with startups coming into picture, a lot of companies have the complete app created by a single team.

Before the creation of NodeJS in 2009, most of the backend programming was done in Java. But with the creation of NodeJS, a lot of companies started using NodeJS in the backend.

The reason is that they can have a JavaScript engineer build the frontend in ReactJS, and the backend in NodeJS. Both ReactJS and NodeJS are JavaScript libraries and are coded using common JavaScript.

We will learn about each tech in MERN and its usage in the MERN stack.

MongoDB

The M in MERN stands for MongoDB. Now, MongoDB is a NoSQL database that stores the data in BSON format. The full form of BSON is Binary JSON and it is quite similar to JSON (JavaScript Object Notation).

It is an object which contains key-value pairs. The strings should be in double quotes, which is similar to JSON. But BSON also contains types like datetime.

A sample BSON document from a MongoDB database is as follows:

```
{  
  "_id": {  
    "$oid": "646c4c0b3ed1bed3064c20e2"  
  },  
  "company": "JP Morgan",  
  "position": "Architect",  
  "jobType": "full-time",  
  "createdBy": {  
    "$oid": "6450b1cbfb52c1a172e790e2"  
  },
```

```
“createdAt”: {  
    “$date”: {  
        “$numberLong”: “1684245010932”  
    }  
},  
“updatedAt”: {  
    “$date”: {  
        “$numberLong”: “1684245010932”  
    }  
},  
“__v”: {  
    “$numberInt”: “0”  
}  
}
```

MongoDB is totally different from traditional Relational databases, which store data in rows and tables. There are also complex relations between tables, which is not required in MongoDB.

MongoDB is a very fast database, with data stored as BSON. Therefore, it is used in most modern web-apps, like an e-commerce site or ride-sharing app. These apps are less concerned about the success of a transaction, like banking apps.

They are more concerned about getting millions of concurrent connections and handling them. Here, MongoDB excels as a NoSQL database, which makes it so popular in the MERN stack.

Express

The E in MERN stands for Express. Now, Express or ExpressJS is not a JavaScript framework but a NodeJS framework, which in turn is a JavaScript backend framework.

Now, NodeJS is used for a variety of backend tasks, including socket programming. But the major task of a backend language, which is to create API endpoints, can also be done in NodeJS.

We will learn more about the REST API in the next section. But they are basically a bridge between the database and the frontend. Writing API endpoints is the main feature of Express.

But why do we need Express, when we also have NodeJS to write API endpoints? The answer is that it is easier to write API code in Express than in Vanilla NodeJS.

So, when Express was created in 2010, it always started to be used with NodeJS for API development.

Following is the express code for a GET API. Here, we are first importing the express in our app. Then we are using it. Next, we are creating the home route '/' with the simple app.get() function.

Lastly, we are listening at port 8000.

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

app.get('/', (req, res) => {
  res.send('Welcome to Express');
});

app.listen(8000, () => console.log(`Server is up on port 8000`));
```

Once we run the app and goto <http://localhost:8000> from our browser, we will get Welcome to Express text.



Figure 1.1: Express app on browser

ReactJS

The R in MERN stands for ReactJS. ReactJS is the world's most popular JavaScript Library/Framework. It is just a small JavaScript Library, but way more popular than the JavaScript Framework of Angular.

Almost all major sites on this planet have frontend as ReactJS or some part in ReactJS. It powers Facebook, Instagram, Netflix, Airbnb, BBC, Paypal, Reddit, and almost every other web-app on this planet.

ReactJS was released in 2013 by Facebook, and before that AngularJS was released in 2010, by Google. Angular powers most of Google web-apps and many corporate apps also.

But even after a late release, ReactJS became the de facto library for startups and most companies due to its ease of learning in comparison to Angular. Being a small library, a lot of major tasks like routing and global state management are done by external packages.

But due to its popularity, ReactJS has more than 200,000 open source packages available. You can find a package for everything, from creating beautiful forms to drag drop packages.¹

Sites made with ReactJS are very fast, even with extreme loads. The best examples are Facebook, Instagram, and Netflix. This power of ReactJS comes from using Virtual DOM instead of the real DOM.

The DOM manipulation is generally slow, so ReactJS changes the Virtual DOM at first. Later, it changes the DOM behind the scene.

Code written in ReactJS uses a special syntax called JSX (JavaScript XML). In ReactJS, we basically combine the HTML and JavaScript file into one file, called the JSX file.

In ReactJS, we divide the project into small, manageable components. This also helps to divide a large project into different teams. Following is the code for a simple Greet component.

Here, we are receiving data from a Parent component. This is called props. And after that, we are showing it using html. Notice that in a js extension file, we are directly showing html. This kind of file gets converted using the in-built Babel and web-pack to HTML, CSS and JavaScript code because the browser only understands them and not ReactJS.

```
import React from 'react'

const Greet = (props) => {
  const {lang, children} = props;
  return (
    <>
    <h1>Greet from {lang}</h1>
    <p>{children}</p>
```

</>

)

}

export default Greet

[NodeJS](#)

The N in MERN stands for NodeJS. It's a backend scripting language created for writing backend code. It allows you to write backend code using JavaScript.

NodeJS was created in 2009 by Ryan Dahl. It uses the famous V8 engine of Chrome, which is used to run JavaScript on the browser. But NodeJS runs JavaScript outside the browser and runs on all Windows, Linux, Unix, and MacOS machines.

It can handle a lot of concurrent connections, which is around 70,000 requests per second. When a web-app becomes popular, more users connect to it, and hence the load on the server increases.

NodeJS is the second-fastest backend language after Java, which can handle 1 million requests per second. So, NodeJS can be easily used in medium-level apps which get less than or equal to 70,000 requests per second.

NodeJS can be used in all kinds of backend programming; from creating API endpoints to creating real-time apps with socket programming. Further API programming can be done using ExpressJS.

The following is a small piece of code from socket programming. This kind of programming is used to create messaging apps like WhatsApp.

In this kind of app, we need to use a socket library. After that, the server creates the connections between different clients which emits messages from other clients through it.

```
const socketio = require('socket.io')

io.on('connection', socket => {

  socket.on('join', ({ username, room }, callback) => {

    const { error, user } = addUser({ id: socket.id, username, room })

    if(error) return callback(error)

    socket.join(user.room)

    socket.emit('message', generateMsg('Admin','Welcome to the Chat App!'))

    callback()

  })

  server.listen(port, () => console.log(`Server is up on port ${port}`))
})
```

Other Popular stacks beyond MERN

There are two other popular stacks beyond MERN. These stacks use the same backend technologies and database. Only the frontend technology gets changed.

These are MEAN (MongoDB, Express, Angular, NodeJS) and MEVN (MongoDB, Express, VueJS, NodeJS). In these stacks, the other two popular JavaScript frontend frameworks of Angular and VueJS are used.

REST APIs

REST APIs, also known as RESTful endpoints, are the way the modern internet works in most cases. In the Internet, there is a client which nowadays is mostly a browser or a mobile app.

All web-apps like Facebook, Instagram, Netflix or mobile apps get the data from a database. But they don't read the data directly from a database. They interact with a middleware through HTTP (HyperText Transfer Protocol) protocol, using these predefined RESTful endpoints.

There are four basic REST APIs - POST, GET, PUT, and DELETE. They are used to perform the CRUD (Create, Read, Update, Delete) operation on a database.

As shown in Figure 1.2, the client machine sends REST API requests and then gets back the response from the database. The REST APIs are created using languages like NodeJS, Spring Boot (Java), C# .NET, and many others.

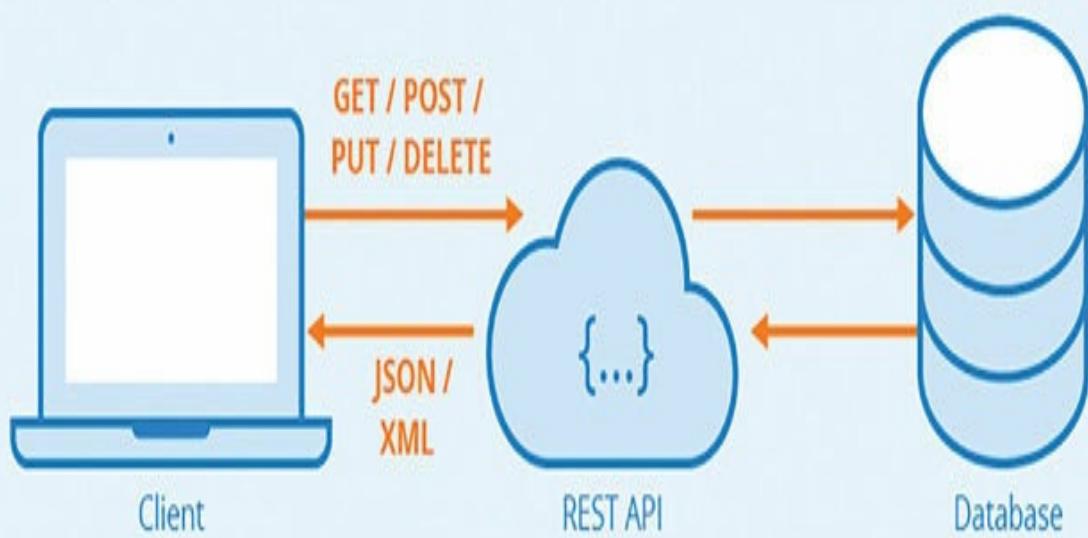


Figure 1.2: REST API

Next, we will look into each REST API endpoint in detail.

POST

The POST API is used to implement the Create operation in CRUD. It is one of the most used REST APIs, which we use from the moment we start using a web-app.

The first time you go to a social media web-app like Facebook, it asks you to register. Now, you enter your name, age, and other details in a form and hit the Register button.

The data will be sent through a POST API to the Facebook database and will be stored there. Next, once we log in, we will create a new post and hit the post button on Facebook.

Again, the POST API will come into picture and send the data to Facebook server. So, whether you purchase some products on Amazon and hit the buy button or upload some photos on Instagram, a POST API comes into picture and sends the data to the respective database.

The data is sent to the backend server in JSON format. The JSON (JavaScript Object Notation) format is similar to JavaScript objects. But here, double quotes are mandatory for strings.

A sample POST API, JSON data for an e-commerce purchase is shown here:

```
[ {"userID": 23,  
 "username": "Keshav",  
 {  
 "id": 1,  
 "qty": 2,  
 "price": 109.95  
 },  
 {  
 "id": 11,  
 "qty": 3,  
 "price": 119.95  
 }  
 ]
```

GET

The GET API is used to implement the Read operation in CRUD. It is the most commonly used API in the world.

Continuing with the Facebook example from POST, when you go to Facebook for the first time, you see a lot of posts. These posts are read from a database through the GET API.

Whether you are scrolling through your Instagram feed, or browsing through Amazon products, or checking articles on Medium, the GET API comes into picture. It reads all the data from the database and shows it in the frontend.

The data is received from the REST APIs in JSON format. Following is a sample JSON data received from an e-commerce site. Notice that most of the time the data is an array of objects. This is the case because arrays are easy to iterate and objects can store data about the items.

After receiving this data, it is the responsibility of the frontend developer to iterate over it and show it on the website. So, this is how you see those beautiful items on the Amazon home page or structured articles on the Medium home page.

Also, when we click a particular item on Amazon or a post on Medium, another GET API is done for that individual item to get all the data.

```
[  
 {  
   "id": 1,  
   "title": "Fjallraven",  
   "price": 109.95,  
   "description": "Your perfect pack for everyday use and walks in the forest.",  
   "image": "https://fakestoreapi.com/img/81fPKd-2AYL._AC_SL1500_.jpg",  
 },  
 {  
   "id": 2,  
   "title": "Mens Casual T-Shirts ",  
   "price": 22.3,  
   "description": "Slim-fitting style, contrast raglan long sleeve",  
   "image": "https://fakestoreapi.com/img/71-  
3HjGNDUL._AC_SY879._SX._UX._SY._UY_.jpg",  
 },  
 {  
   "id": 3,  
   "title": "Mens Cotton Jacket",  
   "price": 55.99,
```

“description”: “great outerwear jackets for Spring/Autumn/Winter, suitable for many occasions.”,

“image”: “https://fakestoreapi.com/img/71li-ujtlUL._AC_UX679_.jpg”,

}

]

PUT

The PUT API is used to implement the Update operation in CRUD. It is used when some existing data needs to be updated.

Continuing with the Facebook example, after we create some posts. We might need to edit a post due to some mistakes So, we click the edit and make the changes.

Again we hit the post button and the update is done. But for the update to happen in the database the exact identification of the post is required. Each post contains a unique id, as in the GET example we can see the unique id for each item.

So, the PUT API sends the id along with the data to be updated to the backend server, which in turn updates the exact data in the database.

The following is a sample PUT API json, where we update our small post. Note that the id is mandatory here, since this will be used to update the correct post in the database.

```
{  
  "id": 101,  
  "title": "ReactJS tutorials",  
  "description": "ReactJS is the most popular JavaScript frontend library."}
```

}

DELETE

The DELETE API is used to implement the Delete operation in CRUD. This API is used when we want to delete data from the database.

Continuing with the Facebook example, after we create some posts. Suppose, we want to delete one of our Facebook posts, then we hit the delete button on the post.

Now, to delete the post, we just need to send the id of the post to the Facebook server. The Delete API only needs the id to be sent to the backend server, and it gets deleted from the database.

Alternatives to REST API

In the past few years, two alternatives have emerged to the REST API. One is the tool of Firebase, in which the frontend directly interacts with the database without any need for a middleware written in language like NodeJS.

Here, instead of API endpoints we have some boiler plate code which directly does the CRUD operations. But Firebase is not as popular as REST endpoints created using backend languages.

Another technology is GraphQL, which is gaining a lot of momentum for the past two years. It also has an endpoint but we can get selective data from the database. Let's consider in the GET example, where we want only the title and price for all items.

If we are using REST then we either need to write this logic in the frontend after receiving all data or we need to create a new endpoint. But this can be easily achieved through GraphQL query.

Sample GraphQL query to achieve the same. With this query, we will only receive the title and price for each item.

```
{
```

```
  items {
```

```
    title
```

price

}

}

Complete Project Overview

In this book, we are going to create a complete MERN project. This will be a task app in which we will be able to add our daily tasks.

The frontend will be made in ReactJS, and we will also do authentication through JWT (JSON Web Token). Here, we will have the complete register and login functionality also.

For global state management, we will be using the latest Redux toolkit in the project. Coming to the backend, we will use ExpressJS and NodeJS. MongoDB cloud database is used to store our data.

Besides this, we are also going to Unit test our frontend using Jest and React Testing library and the backend using Jest and Supertest.

Lastly, we are going to deploy the frontend using Netlify and backend using the Heroku alternative of back4app.

Setting up a NodeJS Application

We are going to start with the backend first. So, we will create a NodeJS app. As we are not creating the front end first, so we need Postman to test our API endpoints.

We are going to learn about it later. And we are going to use the MongoDB cloud database to store our data. The MongoDB installation can be done locally as well, but then it's become difficult to deploy it.

Project Folder Structure

While working with the backend, knowledge of the terminal is required. MacOS and Linux come in-built with the terminal, but now Windows also comes with a superior terminal app.

So, open up any terminal in any operating system. In Windows, the old Command Prompt will also work, but it's better to use the new terminal. We will be using a Windows terminal in this book.

So, open the Windows terminal app. We have opened it on the desktop. After that, create a new folder with the mkdir command. We have named the folder Task-MERN.

Next, change to the folder using the cd command. Now, we will be inside this folder and create another folder called backend with mkdir command here.

```
MINGW64 ~/Users/naben/Da + v
haben@habendu MINGW64 ~/Desktop
$ mkdir Task-MERN
haben@habendu MINGW64 ~/Desktop
$ cd Task-MERN/
haben@habendu MINGW64 ~/Desktop/Task-MERN
$ mkdir backend
haben@habendu MINGW64 ~/Desktop/Task-MERN
$
```

Figure 1.3: Project structure

Now, we will change to the backend folder first with the cd command. Now, since the backend in NodeJS and frontend in ReactJS require node and npm to be installed on the system.

We are checking the same with node -v and npm -v commands.

```
naben@Kabendu MINGW64 ~/Desktop/Task-MERN
$ cd backend/
naben@Kabendu MINGW64 ~/Desktop/Task-MERN/backend
$ node -v
v16.28.0

naben@Kabendu MINGW64 ~/Desktop/Task-MERN/backend
$ npm -v
8.19.4
```

Figure 1.4: Node version

If NodeJS is not installed on the system, then head over to <https://nodejs.org> to install the same. Just download the LTS version on your system and double click the package, which will be downloaded.

The installation for Mac and Windows is almost the same, just some screens and location for installation will change.

Note that when we install NodeJS on our operating system, then npm (Node Package Manager) also gets installed on the system automatically.

A screenshot of a web browser window displaying the official Node.js website at nodejs.org/en. The browser interface includes a title bar with the Node.js logo, a toolbar with various icons, and a menu bar with options like HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. The main content area features a large heading 'Node.js® is an open-source, cross-platform JavaScript runtime environment.' Below this, there is a section titled 'Download for Windows (x64)' with two prominent buttons: '18.16.0 LTS' (Recommended For Most Users) and '20.2.0 Current' (Latest Features). At the bottom of the page, there is a footer with copyright information and links to various legal documents.

Node.js

nodejs.org/en

node®

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | CERTIFICATION | NEWS

Node.js® is an open-source, cross-platform JavaScript runtime environment.

Download for Windows (x64)

18.16.0 LTS
Recommended For Most Users

20.2.0 Current
Latest Features

Other Downloads | Changelog | API Docs Other Downloads | Changelog | API Docs

For information about supported releases, see the [release schedule](#).

Copyright OpenJS Foundation and Node.js contributors. All rights reserved. The OpenJS Foundation has registered trademarks and uses trademarks. For a list of trademarks of the OpenJS Foundation, please see our [Trademark Policy](#) and [Trademark List](#). Trademarks and logos not indicated on the list of OpenJS Foundation trademarks are trademarks™ or registered® trademarks of their respective holders. Use of them does not imply any affiliation with or endorsement by them.

The OpenJS Foundation | [Terms of Use](#) | [Privacy Policy](#) | [Bylaws](#) | [Code of Conduct](#) | [Trademark Policy](#) | [Trademark List](#) | [Cookie Policy](#)

Figure 1.5: NodeJS installation

Now in the backend folder give the command `npm init -y` to create an empty `package.json` file with default options. The `package.json` file is a must for any modern frontend or backend project, because it is used to install node packages in the project.

Next, we will give the command code to open the project in VS Code. This is the default code editor recommended for this book. You can download it for free from <https://code.visualstudio.com/>.

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$ npm init -y
Write to C:\Users\naben\Desktop\Task-MERN\backend\package.json:
```

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

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$ code .
```

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$ |
```

Figure 1.6: Creating package.json

Package installation

Once VS Code is opened, first create a .gitignore file. Now, whatever we put in this file will not go to the GitHub repository.

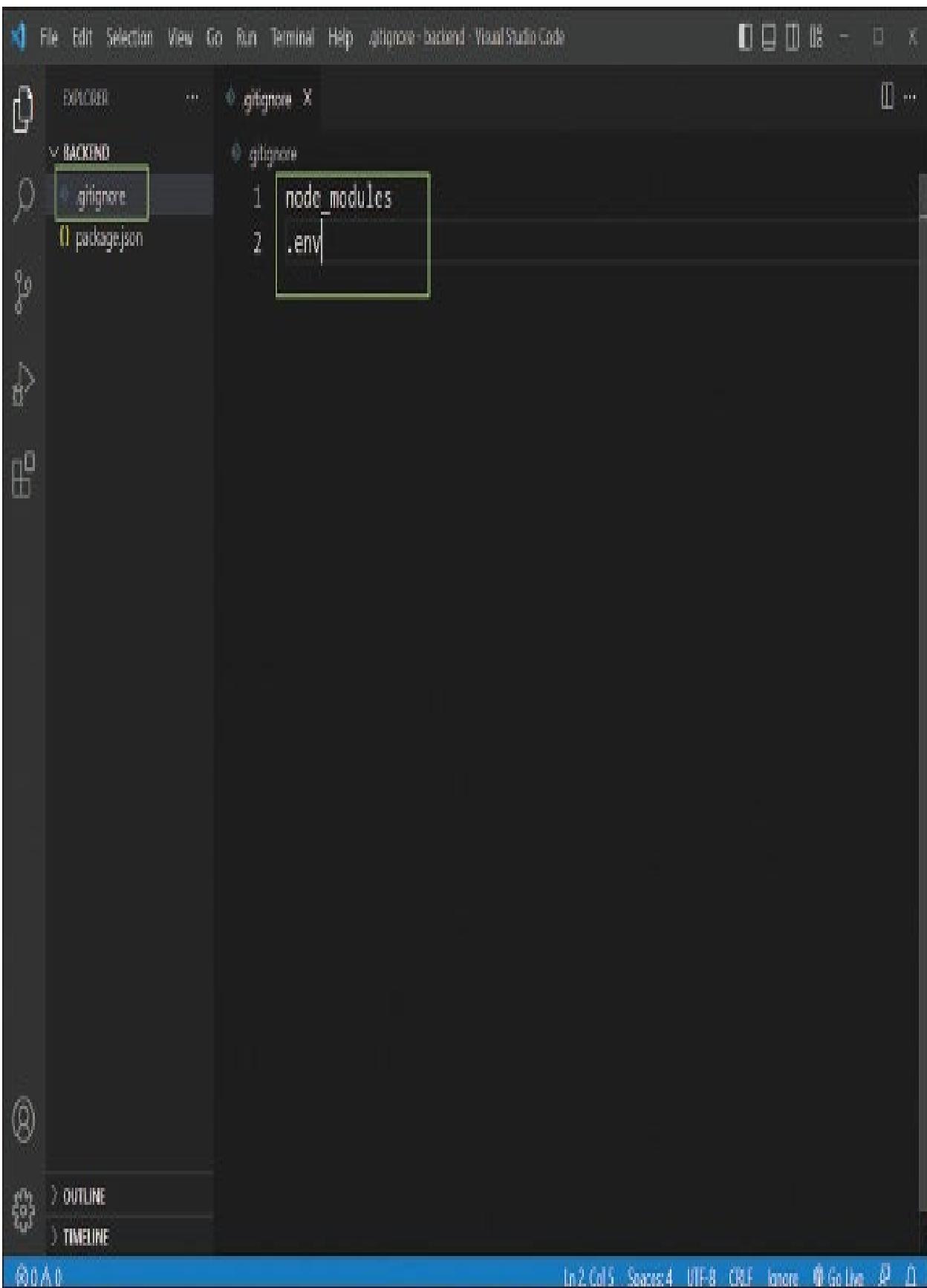


Figure 1.7: Creating .gitignore

Now, we will start installing the packages required in our project. Here, we have opened the Integrated terminal in VS Code, by pressing Ctrl + J. Next, we have installed the package of express, dotenv and mongoose by giving the following command in the terminal:

```
npm i express dotenv mongoose
```

Here, express is the NodeJS framework which we are going to use to create API endpoints. The package of mongoose will be used to connect to the MongoDB database. And the dotenv package uses an .env file to save secrets in a project.

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, ...
- Title Bar:** package.json - backend - Visual Studio Code
- Sidebar:** Includes icons for File, Find, Replace, Go To, and Terminal.
- Editor:** The package.json file is open. Lines 13, 14, and 15 are highlighted with a green box:

```
1 "description": "",  
2 "main": "index.js",  
3 // Debug  
4 "scripts": {  
5   "test": "echo \\\"Error: no test specified\\\" & exit 1"  
6 },  
7 "keywords": [],  
8 "author": "",  
9 "license": "ISC",  
10 "dependencies": {  
11   "dotenv": "^16.0.3",  
12   "express": "^4.18.2",  
13   "mongoose": "^7.2.0"  
14 },  
15 }  
16 }
```
- Bottom Navigation:** PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL.
- Terminal:** Shows the command \$ npm i express dotenv mongoose and its output:

```
nenben@Kabendu MINGW64 ~/Desktop/Task-MERN/backend  
$ npm i express dotenv mongoose  
added 83 packages, and audited 84 packages in 11s  
9 packages are looking for funding  
  run 'npm fund' for details  
found 0 vulnerabilities
```

At the bottom of the terminal, there is a small icon of a person with a gear.
- Status Bar:** Line 13, Col 25, Spaces 2, UTF-8, {}, JSON, Go line, and a few other icons.

Figure 1.8: Installing dependencies

Now, we will also install dev dependencies. These types of packages are not used in production, so they are installed through dev dependencies.

Here, we are going to install the package of nodemon. This package will restart our backend server the moment we make any change in the code. If we run the server code using node command, it needs to be restarted every time manually for changes to appear.

So, give the following command in the terminal. Note that we need to use the --save-dev flag to install a package as dev dependency:

```
npm i --save-dev nodemon
```

Notice that in package.json the nodemon package is installed as dev dependency.

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the file structure with `package.json` highlighted.
- Code Editor:** Displays the `package.json` file content. A yellow box highlights the `devDependencies` section, which contains the entry `"nodemon": "^2.0.22"`.
- Terminal:** Shows the command `$ npm i --save-dev nodemon` being run, followed by the output:
 - added 32 packages, and audited 116 packages in 3s
 - 12 packages are looking for funding
run `'npm fund'` for details
 - found 0 vulnerabilities
- Status Bar:** Shows the current file is `package.json - Visual Studio Code`, and the bottom right includes status icons for file changes, git, and other development tools.

Figure 1.9: Installing dev dependencies

Running the project

To run the project, we need to make some modifications in the package.json file. First, create a server.js file. And then in the package.json file change the main to server.js.

Next, create two scripts for production and development. Give the following lines inside scripts, as shown in Figure 1.10:

“start”: “node server.js”,

“dev”: “nodemon server.js”

Now, the production code will run on deployment server back4app(heroku alternative) and it will run the node command.

But while doing development, we will run the nodemon command.

File Edit Selection View Go Run ... package.json - Visual Studio Code

EXPLORER package.json server.js

BACKEND node_modules .gitignore package-lock.json package.json server.js

```
1 [
2   "name": "backend",
3   "version": "1.0.0",
4   "description": "",
5   "main": "server.js",
6   "scripts": {
7     "start": "node server.js",
8     "dev": "nodemon server.js"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "dependencies": {
14    "dotenv": "^16.0.3",
15    "express": "^4.18.2",
16    "mongoose": "^7.2.0"
17  },
18  "devDependencies": {
19    "nodemon": "^2.0.22"
20  }
21 ]
22
```

OUTLINE TIMELINE

Ln 9 Col 5 Spaces 2 UTF-8 LF JSON Goline ⌂ ⌂

Figure 1.10: Creating Scripts

Now, open the server.js file and put a simple console log in this file.

```
console.log("Welcome to server");
```

After that, open the Integrated Terminal and run the command npm run dev, to run the nodemon command from package.json file.

Here, if everything is ok, we will see the console log of Welcome to server.

The screenshot shows a Visual Studio Code interface with a dark theme. In the top left, there's a file tree icon. The top bar includes the standard menu options: File, Edit, Selection, View, Go, Run, Terminal, Help, and a tab for 'server.js - backend - Visual Studio Code'. On the right side of the top bar are icons for close, minimize, maximize, and exit.

The main workspace contains two tabs: 'server.js' and another unnamed tab. The 'server.js' tab is active, showing the following code:

```
1 console.log("Welcome to server");
```

Below the editor is a toolbar with icons for Problems, Output, Debug Console, and Terminal. The Terminal tab is selected, indicated by a blue underline. The terminal window displays the following command and its output:

```
rajan@Kabendu MINGW64 ~/Desktop/Task-MERN/backend
$ npm run dev
```

The output shows the execution of the command and the logs from nodemon:

```
> backend@1.0.0 dev
> nodemon server.js

[nodemon] 2.0.22
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching path(s): *
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js'
Welcome to server
[nodemon] clean exit - waiting for changes before restart
```

Figure 1.11: Starting Backend

Conclusion

In this chapter, we learned about MERN in detail. First, we learned about all four technologies: MongoDB, Express, ReactJS, and NodeJS. Beside this, we also learned about other popular stacks beyond MERN.

Next, we covered the REST API in detail. We learned about POST, GET, PUT, and DELETE in detail which are used to implement the CRUD operations. We also learned about two alternatives to REST API, namely, Firebase and GraphQL.

Next, we learned about the project we are going to create in this book. We ended this chapter with setting up the basic NodeJS app.

In the next chapter, we will learn about creating the Express server, then creating routes, controllers, and also error handling.

CHAPTER 2

Starting a NodeJS App

Introduction

This chapter talks about the Express server. Here, we will learn to use Express in our backend app. Next, we will be creating routes with Express. We are also going to add controllers to those routes. Lastly, we will learn to do error handling in our Express app.

Structure

In this chapter, we are going to discuss the following topics:

Using Express

Creating routes

Creating Controllers

Error Handling

Using Express

Back in the server.js file, we will first import Express which we installed in the previous chapter. After that, we will import the dotenv package. Then we created a variable port and gave it a value 8000.

Now, we will create an app variable and use Express in it. Next, we will use the listen function of NodeJS, which will be used next on the passed port.

Lastly, we have opened the integrated terminal and given the command npm run dev to run the server.js file. If everything is okay, we will get the message Server listening on 8000.

```
const express = require('express');
const dotenv = require('dotenv').config();
const port = 8000;

const app = express();

app.listen(port, () => console.log(`Server listening on ${port}`));
```

The screenshot shows a Visual Studio Code interface with the following details:

- File Bar:** File Edit Selection View Go Run Terminal Help severjs-backend - Visual Studio Code
- Code Editor:** A file named `server.js` is open, containing the following code:

```
1 const express = require('express');
2 const dotenv = require('dotenv').config();
3 const port = 8000;
4
5 const app = express();
6
7 app.listen(port, () => console.log(`Server listening on ${port}`));
```
- Terminal:** The terminal window shows the command `npm run dev` being run in the directory `/Desktop/Task-MERN/backend`. The output indicates the process is starting with nodemon:

```
raben@Ababdu:~/Desktop/Task-MERN/backend$ npm run dev
> backend@1.0.0 dev
> nodemon server.js

[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *
[nodemon] watching extensions: js,ejs,json
[nodemon] starting `node server.js'
Server listening on 8000
```
- Bottom Status Bar:** Shows the system tray with icons for battery, signal, and volume.

Figure 2.1: Starting with express

Next, we will create a .env file. Now, this file is used to keep all the secrets that we don't want to put in GitHub or any public view.

Here, we have created two variables NODE_ENV and PORT and given them development and 8000 values, respectively.

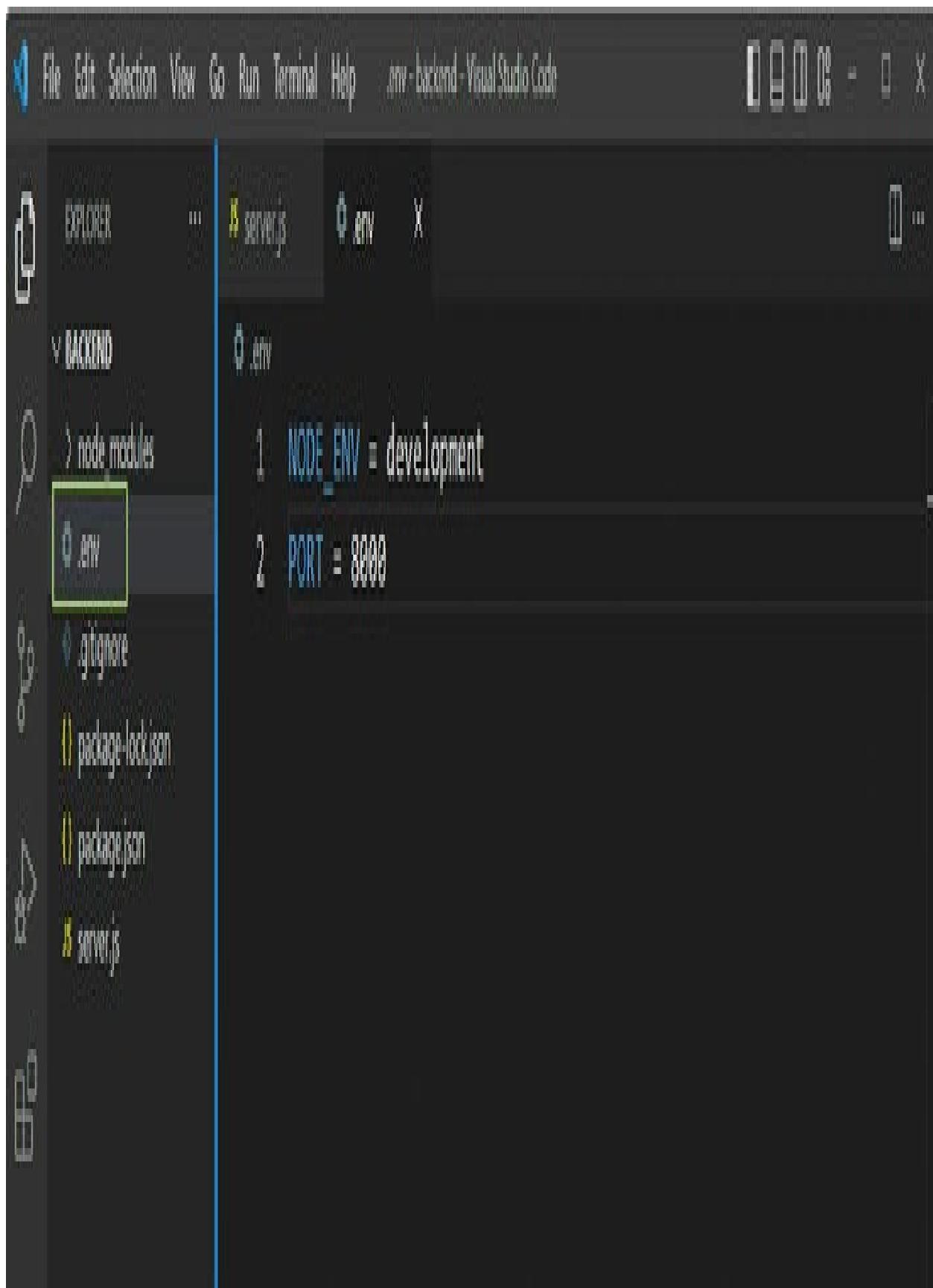


Figure 2.2: Adding environment file

Back in the server.js file, we have updated the port variable to use the value from the .env file first. Here, we have given process.env.PORT and also given 5000 with OR(||) operator.

And restarted the server by giving the npm run dev command. Notice that the port 8000 is taken from the .env file.

The screenshot shows a Visual Studio Code interface with a dark theme. The top bar includes the standard menu (File, Edit, Selection, View, Go, Run, Terminal, Help) and the title "server.js - backend - Visual Studio Code". The left sidebar has icons for file operations like Open, Save, Find, and Refresh. The main editor area contains the following code:

```
JS server.js X
JS server.js > [0] port
1 const express = require('express');
2 const dotenv = require('dotenv').config();
3 const port = process.env.PORT || 5000;
4
5 const app = express();
6
7 app.listen(port, () => console.log(`Server listening on ${port}`));
```

Below the editor is a navigation bar with tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, showing a command-line interface. The terminal window displays the following session:

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$ npm run dev
> backend@1.0.0 dev
> nodemon server.js

[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js'
Server listening on 8000
```

Figure 2.3: Using environment variable

[Creating Routes](#)

Now, we will create our routes. Routes are endpoints that are used by the frontend to perform the CRUD operations. This is achieved through the REST APIs of GET, POST, PUT, and DELETE.

Initial Routes

Back in our server.js file we will first add the following code to create a simple GET route:

```
app.get('/api/tasks', (req, res) => {  
  res.send('Get All Tasks');  
})
```

Here, we are using the get method from Express. The first parameter is the route path and it can be anything. We have given it /api/tasks.

The second parameter is a function call with req and res. In this arrow function, we are using res.send() to send back Get All Tasks text. Later on, we are going to modify this route to get all tasks.

The screenshot shows a dark-themed instance of Visual Studio Code. The title bar displays "File Edit Selection View Go Run Terminal Help server.js · backend · Visual Studio Code". The main editor area contains the following code:

```
server.js ✘
1 const express = require('express');
2 const dotenv = require('dotenv').config();
3 const port = process.env.PORT || 5000;
4
5 const app = express();
6
7 app.get('/api/tasks', (req, res) => {
8     res.send('Get All Tasks');
9 })
10
11 app.listen(port, () => console.log(`Server listening on ${port}`));
```

The code implements a simple Express.js application. It starts by requiring 'express' and 'dotenv'. It sets the port to 5000 or the environment variable PORT. It creates an Express application and defines a GET endpoint at '/api/tasks' that returns the string 'Get All Tasks'. Finally, it listens on the specified port and logs a message to the console.

Figure 2.4: Creating GET route

Now, head over to a browser and go to `http://localhost:8000/api/tasks` and the Get All Tasks will be displayed.

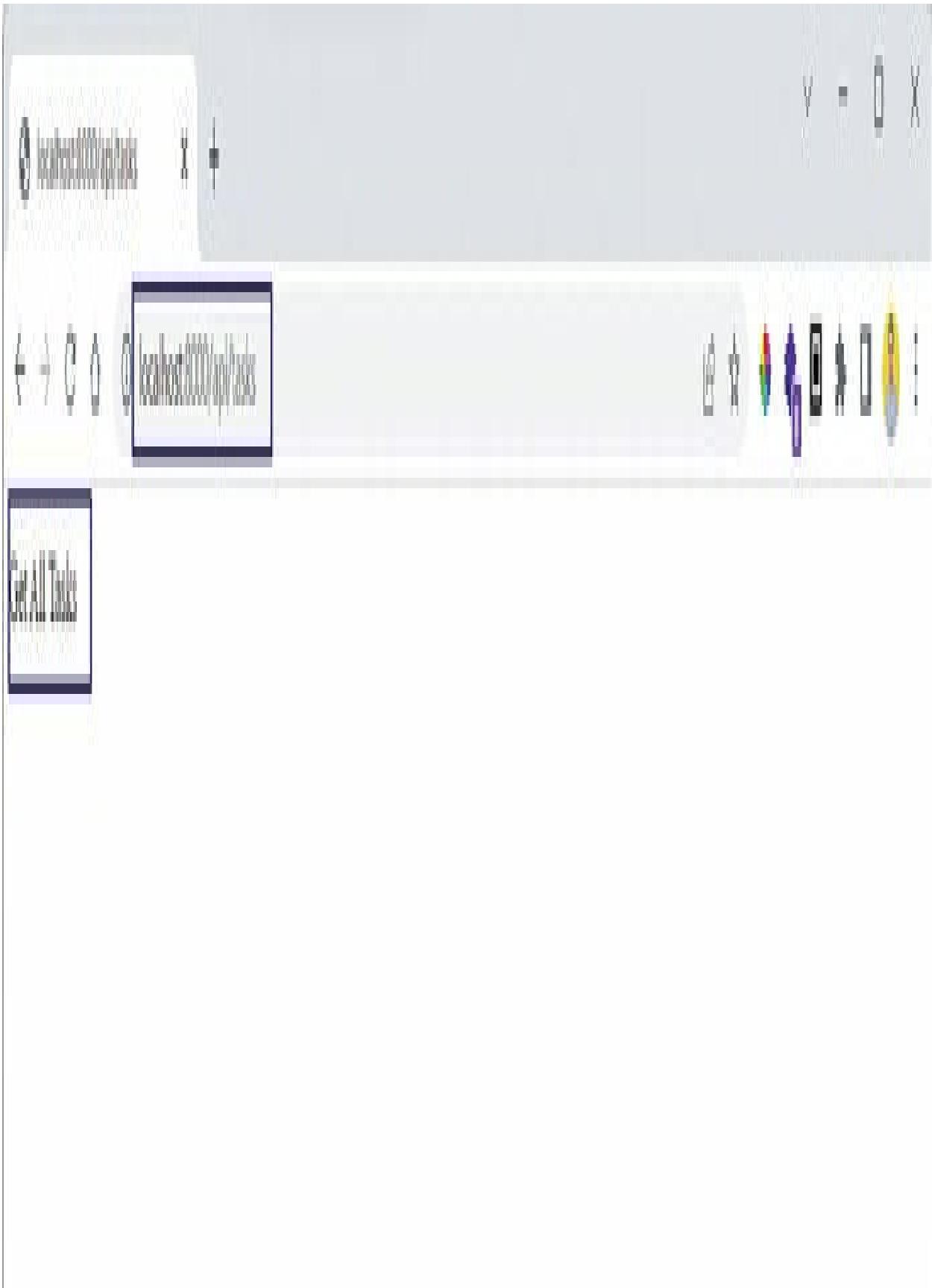


Figure 2.5: Getting back data

Now, in all cases we receive data from the server in JSON format. So, we will update the code in server.js file to send back a json.

Also, note that we are sending back a status code of 200.

A screenshot of the Visual Studio Code interface showing a file named `server.js`. The code is written in JavaScript and defines a simple Express application. It includes importing `express` and `dotenv`, setting a port, creating an Express app, defining a route to return a JSON message, and finally listening on the specified port.

```
File Edit Selection View Go Run Terminal Help server.js - backend · Visual Studio Code
```

```
server.js
```

```
1 const express = require('express');
2 const dotenv = require('dotenv').config();
3 const port = process.env.PORT || 5000;
4
5 const app = express();
6
7 app.get('/api/tasks', (req, res) => {
8     res.status(200).json({ message: 'Get All Tasks' });
9 }
10
11 app.listen(port, () => console.log(`Server listening on ${port}`));
```

Figure 2.6: Returning JSON data

Now, again in the browser go to `http://localhost:8000/api/tasks` and we will get the JSON back.



Figure 2.7: Getting JSON data

[All Routes](#)

Now, we will add more routes in our project. But first, we will separate the route logic to a different file.

So, create a routes folder and inside it, create a taskRoutes.js file. Put the following content in it:

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.status(200).json({ message: 'Get All Tasks' });
}

module.exports = router;
```

Here, we have first imported the express package. And we have also imported the router from Express. Next, we have moved the get route created in the server.js file here.

But notice that we are using a router instead of an app. And we are exporting the

router at the end.

A screenshot of the Visual Studio Code interface. The title bar shows "taskRoutes.js - backend · Visual Studio Code". The left sidebar displays a file tree with a dark theme. The "routes" folder is expanded, showing "taskRoutes.js" which is selected and highlighted with a green border. Other files visible include "server.js", "package.json", "package-lock.json", "gitignore", and ".env". The main editor area shows the code for "taskRoutes.js":

```
1 const express = require('express');
2 const router = express.Router();
3
4 router.get('/', (req, res) => {
5   res.status(200).json({ message: 'Get All Tasks' });
6 }
7
8 module.exports = router;
```

Figure 2.8: Separating Routes

Back in the server.js file, we have removed the get method. Also, we are using the app.use() to hit the /api/tasks route. Here, the second parameter is the taskRoutes file.

```
app.use('/api/tasks', require('./routes/taskRoutes'));
```

A screenshot of the Visual Studio Code interface showing a file named `server.js`. The code is written in JavaScript and defines a web server using Express. It includes imports for `express`, `dotenv`, and `taskRoutes`. The line `app.use('/api/tasks', require('./routes/taskRoutes'));` is highlighted with a green rectangular selection.

```
File Edit Selection View Go Run ... server.js - backend - Visual Studio Code
server.js X taskRoutes.js ...
server.js ...
1 const express = require('express');
2 const dotenv = require('dotenv').config();
3 const port = process.env.PORT || 5000;
4
5 const app = express();
6
7 app.use('/api/tasks', require('./routes/taskRoutes'));
8
9 app.listen(port, () => console.log(`Server listening on ${port}`));

```

Figure 2.9: Adding route path

Now again, in the browser, go to `http://localhost:8000/api/tasks`, and we will get the JSON back. This means our new changes are working fine.

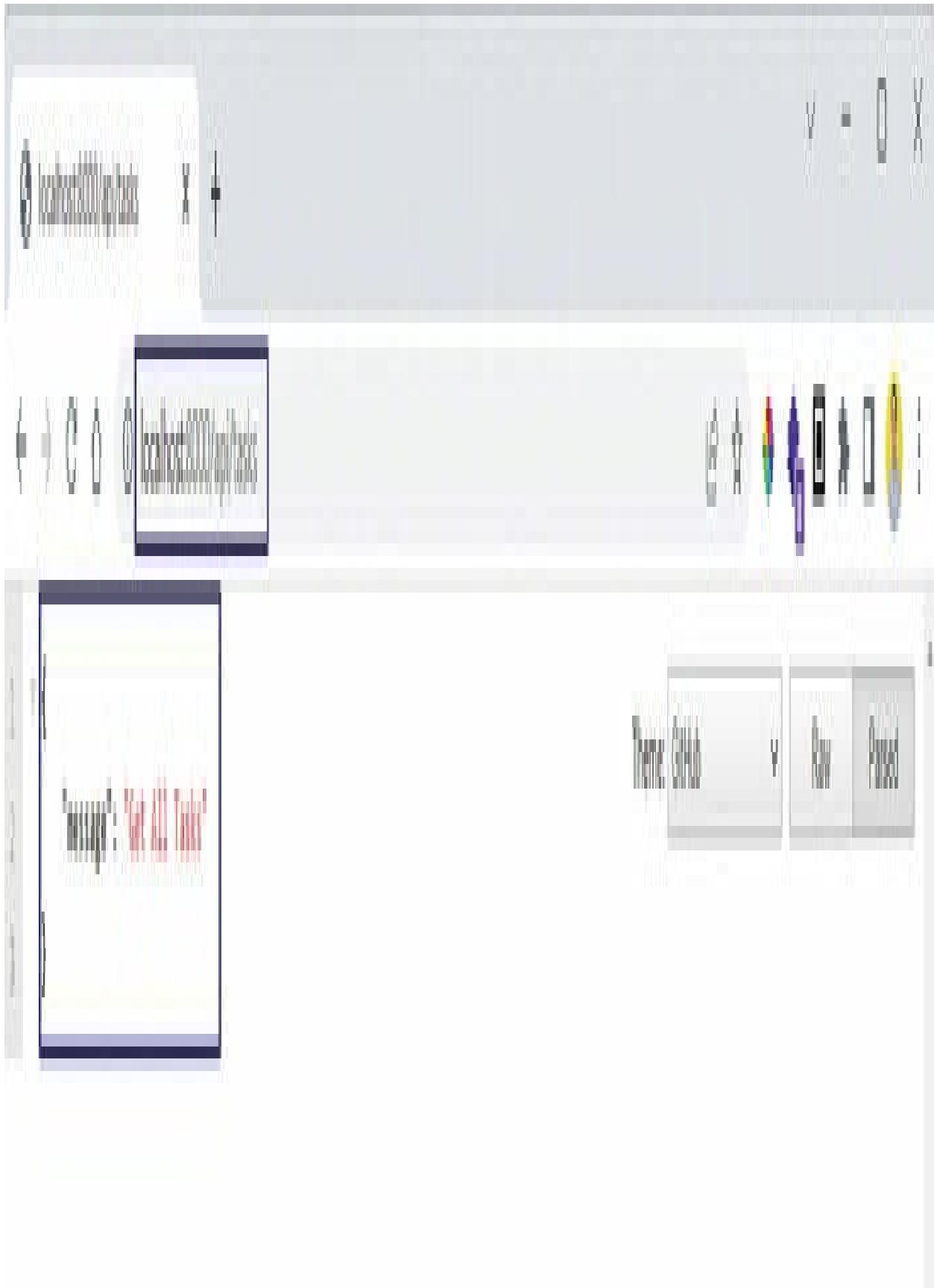


Figure 2.10: Getting all routes

Back in the taskRoutes.js file, we will add a POST, PUT, and DELETE route. Here, we are just using the respective methods.

All these methods, for now, will return simple json. The real logic will be added later on. The post() is similar to get() as of now.

But the put() and delete() take a wild route of /:id. This special route can take any provided item. This id will be sent through the browser address bar. We can get access to it with the req.params.id variable.

```
router.post('/', (req, res) => {
```

```
  res.status(200).json({ message: 'Create Task' });
```

```
})
```

```
router.put('/:id', (req, res) => {
```

```
  res.status(200).json({ message: `Task ${req.params.id} updated.` });
```

```
})
```

```
router.delete('/:id', (req, res) => {
```

```
  res.status(200).json({ message: `Task ${req.params.id} deleted.` });
```

}

A screenshot of the Visual Studio Code interface showing the file `taskRoutes.js`. The code defines a router for a task management application.

```
File Edit Selection View Go Run Terminal Help taskRoutes.js - backend - Visual Studio Code
server.js taskRoutes.js X ...
routes > taskRoutes.js > ...
1 const express = require('express');
2 const router = express.Router();
3
4 router.get('/', (req, res) => {
5   res.status(200).json({ message: 'Get All Tasks' });
6 }
7
8 router.post('/', (req, res) => {
9   res.status(200).json({ message: 'Create Task' });
10 }
11
12 router.put('/:id', (req, res) => {
13   res.status(200).json({ message: `Task ${req.params.id} updated.` });
14 }
15
16 router.delete('/:id', (req, res) => {
17   res.status(200).json({ message: `Task ${req.params.id} deleted.` });
18 }
19
20 module.exports = router;
```

Figure 2.11: Adding more routes

Testing through Postman

The GET route can be seen in the browser, but to check the other routes, we need the frontend. Now, since the front end has not even started, we will use the POSTMAN tool.

This tool is always used by backend engineers because they don't wait for the frontend to be completed. This tool mimics the way the request will be sent from a frontend app through the browser.

The free tool Postman can be installed from <https://www.postman.com/>. After this, once it is opened, we will make the method as POST and in the address bar, give `http://localhost:8000/api/tasks`.

After that, hit the Send button. We will receive back the correct json and also the Status of 200.

Home Workspaces API Network Explore

Search Postman

My Workspace New Import MERN Task Post Task No Environment

Collection + MERN Task / Post Task POST http://localhost:8000/api/tasks Send

JobTracker

MERN Task

Post Task

Netflix API

Postman Echo

spectacular API

Task Manager

Tinder MERN

yelp

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description	Bulk Edit

Body Cookies Headers (7) Test Results Status: 200 OK Time: 18 ms Size: 200 B Save as Example

HTTP Raw Preview Visualize JSON

```
1: "message": "Create Task"
```

Online Find and Replace Console Cookies Capture requests Runner Trash

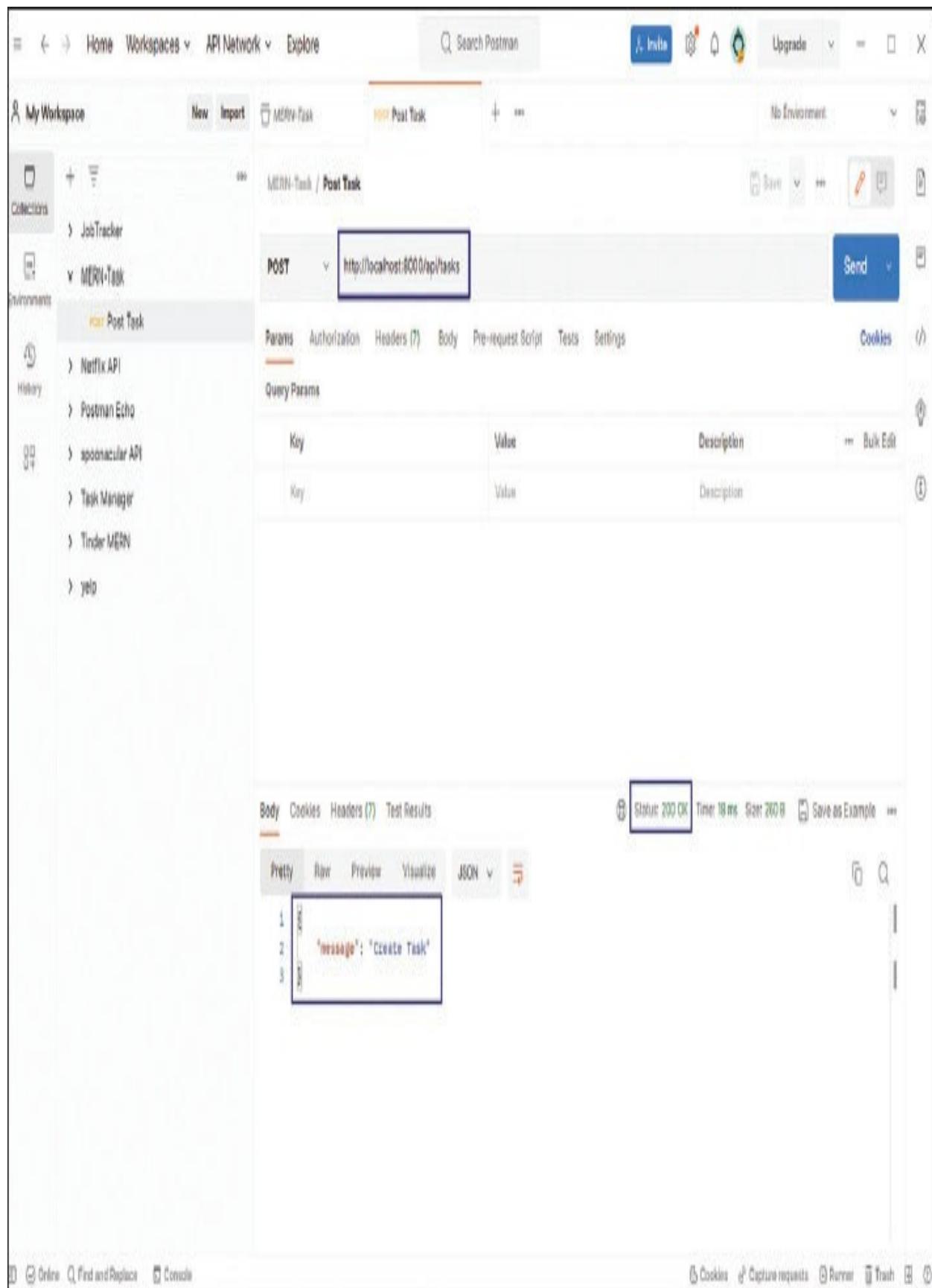


Figure 2.12: POST request

Now, for the PUT method, we also have to provide any id. So, we are visiting the endpoint `http://localhost:8000/api/tasks/23`.

Here, we are also getting back this id in the json, along with the status of 200.

Home Workspaces API Network Explore

Search Postman

Test Upgrade

My Workspace

New Import Run API Task

Post Task

No Environment

Collection: MERN-Task / Put Task

Environments: MERN-Task

PUT https://localhost:8000/api/tasks/23

Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description	Edit

History: MERN API, Postman Echo, socket.io API, Task Manager, Tinder MERN, jHipster

Body Cookies Headers (7) Test Results

Status: 200 OK Time: 4 ms Size: 203 B Save as Example

Pretty Raw Preview Variables JSON

```
1 { "message": "Task 23 updated."}
```

Figure 2.13: PUT request

For the DELETE method also, we have to provide any id. Again, we are visiting the endpoint <http://localhost:8000/api/tasks/23>.

Here, we are also getting back this id in the json, along with status of 200.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar says "Search Postman". On the right side of the header are icons for Help, Upgrade, and a close button.

The main workspace has tabs for "My Workspace", "New", "Import", "Put Task", "Post Task" (which is selected), "Delete Task", and "To Environment".

The left sidebar shows "Collections" with "JobTracker" and "MERN-Task" expanded, and "Environments" with "Post Task", "Put Task", and "Delete Task" (which is selected). Below that is a history section with items like "Netflix API", "Postman Echo", "sporacular API", "Task Manager", "Tinder MERN", and "yelp".

The central area displays a "DELETE" request to "http://localhost:5000/api/tasks/23". The "Params" tab is selected. The "Query Params" table has two rows:

Key	Value	Description	Link

The "Body" tab is selected, showing a JSON response:

```
1 {  
2   "message": "Task 23 deleted."  
3 }
```

At the bottom, status information shows "Status: 200 OK Time: 0ms Size: 231" and "Show as Example".

Figure 2.14: DELETE request

Creating Controllers

Now, we will separate the controller part. For this create a controllers folder. Inside it create a taskController.js file and add the following content:

```
const getTasks = (req, res) => {  
  res.status(200).json({ message: 'Get All Tasks' });  
}  
  
module.exports = { getTasks }
```

Here, the getTasks function will contain the main functionality. Right now, it is sending the response json back.

A screenshot of the Visual Studio Code (VS Code) interface. The window title is "taskController.js - backend - Visual Studio Code". The menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The status bar shows the file path "taskController.js - backend".

The left sidebar displays the project structure:

- BACKEND (highlighted)
- controllers (highlighted)
- taskController.js (highlighted)
- node_modules
- routes
- env
- .gitignore
- package-lock.json
- package.json
- server.js

The main editor area contains the following code:

```
1 const getTasks = (req, res) => {
2     res.status(200).json({ message: 'Get All Tasks' });
3 }
4
5 module.exports = [ getTasks ];
```

Figure 2.15: Creating controllers

Back, in taskRoutes.js file we will remove the code for the controller. And add the function of getTasks in it. We have also imported the getTasks file.

A screenshot of the Visual Studio Code interface showing the file `taskRoutes.js`. The code defines an Express router with various methods for managing tasks.

```
File Edit Selection View Go Run Terminal Help taskRoutes.js - backend - Visual Studio Code
server.js taskRoutes.js taskController.js ...
routes > taskRoutes.js > ...
1 const express = require('express');
2 const router = express.Router();
3 const { getTasks } = require('../controllers/taskController');
4
5 router.get('/', getTasks)
6
7 router.post('/', (req, res) => {
8   res.status(200).json({ message: 'Create Task' });
9 })
10
11 router.put('/:id', (req, res) => {
12   res.status(200).json({ message: `Task ${req.params.id} updated.` });
13 })
14
15 router.delete('/:id', (req, res) => {
16   res.status(200).json({ message: `Task ${req.params.id} deleted.` });
17 })
18
19 module.exports = router;
```

Figure 2.16: Adding Controller in Routes

Now, again in the browser, go to `http://localhost:8000/api/tasks` and we will get the JSON back. This means our new changes are working fine.

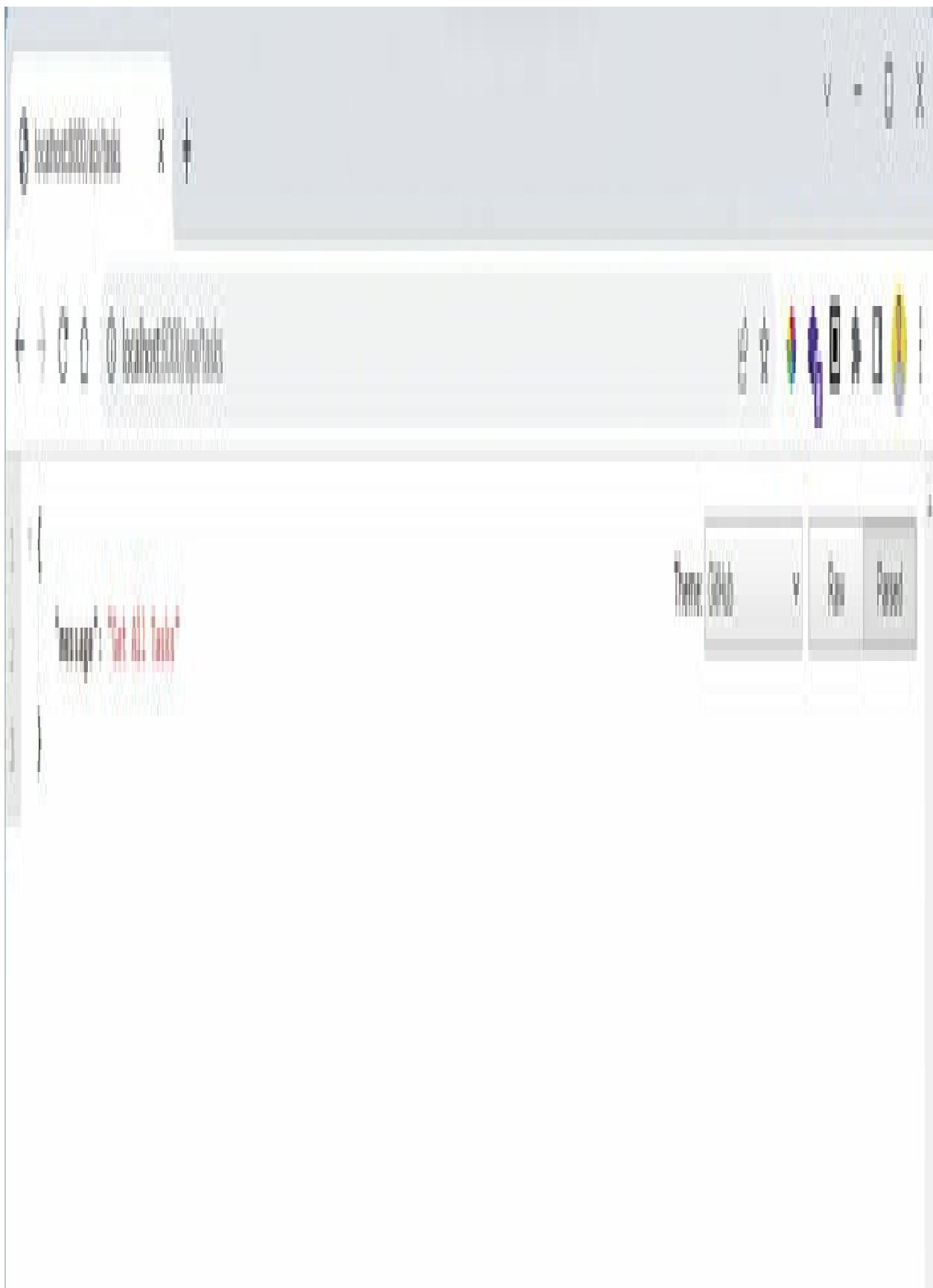


Figure 2.17: Getting JSON back

Now, in the taskController.js file, we will add the rest of the controllers for setTask, updateTask and deleteTask. We have also added the exports for the same.

```
const setTask = (req, res) => {  
  res.status(200).json({ message: 'Create Task' });  
}  
  
const updateTask = (req, res) => {  
  res.status(200).json({ message: `Task ${req.params.id} updated.` });  
}  
  
const deleteTask = (req, res) => {  
  res.status(200).json({ message: `Task ${req.params.id} deleted.` });  
}
```

File Edit Selection View Go Run Terminal Help taskController.js - backend - Visual Studio Code

JS server.js JS taskRoutes.js JS taskController.js X ⌂ ...

controllers > JS taskController.js > [?] <unknown>

```
1 const getTasks = (req, res) => {
2   res.status(200).json({ message: 'Get All Tasks' });
3 }
4
5 const setTask = (req, res) => {
6   res.status(200).json({ message: 'Create Task' });
7 }
8
9 const updateTask = (req, res) => {
10   res.status(200).json({ message: `Task ${req.params.id} updated.` });
11 }
12
13 const deleteTask = (req, res) => {
14   res.status(200).json({ message: `Task ${req.params.id} deleted.` });
15 }
16
17 module.exports = { getTasks, setTask, updateTask, deleteTask }
```

Figure 2.18: Adding more controllers

Back in the taskRoutes.js file, we will add these function calls. Now, our taskRoutes file has become small and manageable.

```
router.post('/', setTask)
```

```
router.put('/:id', updateTask)
```

```
router.delete('/:id', deleteTask)
```

A screenshot of the Visual Studio Code interface showing the file `taskRoutes.js`. The code defines a router for a task management application. It includes imports for Express and a controller module, and defines routes for listing, creating, updating, and deleting tasks.

```
File Edit Selection View Go Run Terminal Help taskRoutes.js - backend - Visual Studio Code

server.js taskRoutes.js taskController.js

router > taskRoutes.js

1 const express = require('express');
2 const router = express.Router();
3 const { getTasks, setTask, updateTask, deleteTask } = require('../controllers/taskController');

4
5 router.get('/', getTasks)
6 router.post('/', setTask)
7 router.put('/:id', updateTask)
8 router.delete('/:id', deleteTask)

9
10 module.exports = router;
```

Figure 2.19: Updating task routes

We are also checking the PUT route through Postman, to check if everything is working fine.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces (with a dropdown menu), API Network (with a dropdown menu), and Explore. To the right of these are search fields for "Search Postman" and "Invite", along with icons for "Upgrade", "Logout", and "X".

The main workspace is titled "My Workspace". It contains a list of environments and tasks:

- Environments:** JobTracker, MERN-Task (selected).
- Tasks:** Post Task, PUT Put Task (selected), PUT Delete Task.

The "PUT Put Task" task is currently selected. Its details are displayed in the center:

- Method:** PUT
- URL:** <http://localhost:8000/api/tasks/23>
- Buttons:** Save, Send (highlighted in blue), and a copy icon.

The "PUT Put Task" tab is active, showing the following configuration:

- Params:** Authorization, Headers (7), Body, Pre-request Script, Tests, Settings.
- Query Params:** A table with columns: Key, Value, Description, and Bulk Edit.

Key	Value	Description	Bulk Edit

At the bottom, the response section shows:

- Status:** Status: 200 OK, Time: 8 ms, Size: 261 B.
- Body:** A JSON response with one item: { "message": "Task 23 updated."}
- Buttons:** Save as Example, and a copy icon.

Figure 2.20: PUT route

Using JSON

We will add the logic to use JSON in our project. Here in the server.js file, we will add the following code:

```
app.use(express.json());  
app.use(express.urlencoded({ extended: false }));
```

Here, through app.use() we have added the functions to use json and urlencoded bodies.

The screenshot shows a Visual Studio Code interface with a dark theme. The menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a tab labeled server.js - backend - Visual Studio Code. The code editor displays the following JavaScript code:

```
1 const express = require('express');
2 const dotenv = require('dotenv').config();
3 const port = process.env.PORT || 5000;
4
5 const app = express();
6
7 app.use(express.json());
8 app.use(express.urlencoded({ extended: false }));
9
10 app.use('/api/tasks', require('./routes/taskRoutes'));
11
12 app.listen(port, () => console.log(`Server listening on ${port}`));
```

Lines 7 and 8 are highlighted with a green rectangular selection.

Figure 2.21: Using JSON

Back in the taskController.js file, in the setTask function, we are console logging the req.body.

A screenshot of the Visual Studio Code interface showing the file `taskController.js`. The code defines four functions: `getTasks`, `setTask`, `updateTask`, and `deleteTask`. The `setTask` function includes a `console.log` statement for debugging. The code is part of a module that exports these functions.

```
File Edit Selection View Go Run Terminal Help taskController.js - Visual Studio Code
server.js taskRoutes.js taskController.js X ...
controllers > taskController.js > ...
1 const getTasks = (req, res) => {
2   res.status(200).json({ message: 'Get All Tasks' });
3 }
4
5 const setTask = (req, res) => {
6   console.log(req.body)
7   res.status(200).json({ message: 'Create Task' });
8 }
9
10 const updateTask = (req, res) => {
11   res.status(200).json({ message: `Task ${req.params.id} updated.` });
12 }
13
14 const deleteTask = (req, res) => {
15   res.status(200).json({ message: `Task ${req.params.id} deleted.` });
16 }
17
18 module.exports = [ getTasks, setTask, updateTask, deleteTask ]
```

Figure 2.22: Console log in controller

Now, in Postman we will update our POST request. Here, first go to Body and then select x-www-form-urlencoded. After that, make a key as text. Also, give any value and we are giving Learn Flutter.

Also, make sure the checkbox is selected and then click the Send button.

Home Workspaces API Network Export Search Postman Help Upgrade - X

MERN-Task / Post Task

POST http://localhost:3000/api/tasks

Body

Form-data

Content-Type: application/x-www-form-urlencoded

Key Value Description

task Loan Future

Body Cookies Headers TestResults

Pretty Raw Preview Validator JSON

```
1 { "message": "Create Task" }
```

Status: 200 OK Time: 14 ms Size: 201 B Save as Example

Online Find and Replace Console Cookies Copy response Run Test 0

Figure 2.23: JSON message in POST

Now, in the terminal where the server is running, we will get { text: ‘Learn Flutter’ } as shown in Figure 2.24:

```
⚡ C:\WINDOWS\system32\cmd X + v - X

Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
Server listening on 8000
[Object: null prototype] { text: 'Learn Flutter' }
```

Figure 2.24: JSON message console log

Error Handler

Lastly, in this chapter we will add error handling in our project. So, in the setTask.js in taskController.js file we will add the following code:

```
if(!req.body.text){  
    res.status(400).json({ message: 'Please enter a task' })  
}
```

Here we are checking if req.body.text doesn't exist, then we are sending back the error message:

A screenshot of Visual Studio Code showing a file named `taskController.js`. The code is written in Node.js and defines two functions: `getTasks` and `setTask`.

```
const getTasks = (req, res) => {
  res.status(200).json({ message: 'Get All Tasks' });
};

const setTask = (req, res) => {
  if(!req.body.text){
    res.status(400).json({ message: 'Please enter a task' });
  }
  res.status(200).json({ message: 'Create Task' });
};
```

Figure 2.25: Error handling in controller

Now, back in Postman, remove the checkbox beside the text. Then click the Send button. We will get the 400 Bad Request and the following JSON.

```
{ "message": "Please enter a task" }
```

The screenshot shows the Postman application interface for API testing. The top navigation bar includes links for Home, Workspaces, API Network, and Explore, along with a search bar and various tool icons. The main workspace displays a collection named "MERN-Task / Post Task". A specific POST request is selected, targeting the URL `http://localhost:8000/api/tasks`. The request details tab is active, showing the method (POST), URL, and a "Send" button. Below this, the Headers tab is selected, containing a single header entry: "Content-Type: application/json". The Body tab is also visible. The response section at the bottom shows a status of 400 Bad Request, with the message "message": "Please enter a task".

Home Workspaces API Network Explore

Search Postman

Upgrad

MERN-Task / Post Task

POST http://localhost:8000/api/tasks

Send

Headers

Content-Type: application/json

Key	Value	Description	...
task	Learn Postman	Description	...
key	value	Description	...

Body Cookies Headers Test Results

Status: 400 Bad Request Time: 20 ms Size: 277 B

message": "Please enter a task"

Figure 2.26: Error in Postman

Now, we have updated the code in taskController.js file. Here, we will be using the in-built error handler in NodeJS. In the following code, we are throwing the error from the if statement:

```
res.status(400)  
throw new Error('Please enter a task')
```

A screenshot of the Visual Studio Code interface showing a file named `taskController.js`. The code is a Node.js script for a task management application. It defines two functions: `getTasks` and `setTask`.

```
File Edit Selection View Go Run Terminal Help taskController.js · backend · Visual Studio Code

server.js taskRoutes.js taskController.js X

controller) taskController.js (1 task

1 const getTasks = (req, res) => {
2   res.status(200).json({ message: 'Get All Tasks' });
3 }

4
5 const setTask = (req, res) => {
6   if(!req.body.text){
7     res.status(400)
8     throw new Error('Please enter a task');
9   }
10
11   res.status(200).json({ message: 'Create Task' });
12 }
```

Figure 2.27: Throwing error from controller

But in Postman, we are not getting the correct error result after hitting the Send button.

We are getting a big HTML back. Next, we will update the code to fix this issue.

Home Workspaces API Network Explore Search Postman Invite Upgrade

API Put Task POST Post Task Delete Task No Environment

MERN-Task / Post Task Save

POST http://localhost:3001/api/tasks Send

Params Authorization Headers Body Pro-request Script Tests Settings Cookies

History none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description
task	Learn Flutter	

Key	Value	Description
-----	-------	-------------

Body Cookies Headers Test Results Status: 400 Bad Request Time: 10 ms Size: 162 KB Save as Example

Pretty Raw Preview Visualize HTML

```
4 <head>
5   <meta charset="utf-8">
6   <title>Error</title>
7 </head>
8
9 <body>
10  <p>Error: Please enter a task<br>    at setTask (C:\Users\naben\Desktop\Task-MERN\backend\controllers\taskController.js:8:15) <br>    at Layer.handle [as handle_request] (C:\Users\naben\Desktop\Task-MERN\backend\node_modules\express\lib\router\layer.js:95:5) <br>    at next (C:\Users\naben\Desktop\Task-MERN\backend\node_modules\express\lib\router\route.js:144:13) <br>    at Route.dispatch (C:\Users\naben\Desktop\Task-MERN\backend\node_modules\express\lib\router\route.js:114:3) <br>    at Layer.handle [as handle_request] (C:\Users\naben\Desktop\Task-MERN\backend\node_modules\express\lib\router\layer.js:93:5) <br>    at Function.process_params (C:\Users\naben\Desktop\Task-MERN\backend\node_modules\express\lib\router\index.js:284:15) <br>    at Function.handle (C:\Users\naben\Desktop\Task-MERN\backend\node_modules\express\lib\router\index.js:277:5)
```

Online Find and Replace Console Cookies Capture response Runner Trash

Figure 2.28: Wrong data in Postman

To fix the same we need to have middleware in our NodeJS code. So, create a middleware folder and inside it a file errorMiddleware.js file, add the following code:

```
const errorHandler = (err, req, res, next) => {  
  const statusCode = res.statusCode ? res.statusCode : 500  
  res.status(statusCode)  
  res.json({ message: err.message })  
}  
  
module.exports = { errorHandler }
```

Here, we have an errorHandler function, which we are exporting also. Inside the function, we will take the passed status code or by default 500. We are also setting the error to the json object.

A screenshot of the Visual Studio Code interface. The title bar reads "errorMiddleware.js - backend - Visual Studio Code". The left sidebar shows a file tree with a dark theme. The "errorMiddleware.js" file is selected and highlighted with a green border. The code editor on the right contains the following JavaScript code:

```
1 const errorHandler = (err, req, res, next) => {
2   const statusCode = res.statusCode || res.statusCode : 500
3   res.status(statusCode)
4   res.json({ message: err.message })
5 }
6
7 module.exports = [ errorHandler ]
```

Figure 2.29: Error Middleware

Back in the server.js file, we are first importing the error middleware. After that, we are using it with app.use():

The screenshot shows a dark-themed interface of Visual Studio Code. At the top, the menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a tab labeled "server.js - backend". The status bar on the right shows icons for file operations like Save, Undo, Redo, and Close.

The main editor area displays the following code:

```
1 const express = require('express');
2 const { errorHandler } = require('./middleware/errorMiddleware');
3 const dotenv = require('dotenv').config();
4 const port = process.env.PORT || 5000;
5
6 const app = express();
7
8 app.use(express.json());
9 app.use(express.urlencoded({ extended: false }));
10
11 app.use('/api/tasks', require('./routes/taskRoutes'));
12
13 app.use(errorHandler);
14
15 app.listen(port, () => console.log(`Server listening on ${port}`));
```

Line 13, which contains the call to `app.use(errorHandler);`, is highlighted with a green rectangular selection.

Figure 2.30: Using error handler

Back to Postman, we will again send the Post request. This time we will get the correct error response back.

Home Workspaces API Network Explore

Search Postman

Invite Upgrade - X

PUT Put Task POST Post Task Delete Task + No Environment

MERN-Task / Post Task

Save ...

POST http://localhost:3000/api/tasks Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

History none form-data x-www-form-urlencoded raw binary GraphQL

Key Value Description Bulk Edit

text Learn Flutter

Key Value Description

Body Cookies Headers (7) Test Results

Status: 400 Bad Request Time: 14 ms Size: 277 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 "Message": "Please enter a task"
```

Figure 2.31: Perfect Error message

Conclusion

In this chapter, we have learned about using Express in our app. After that, we created routes in our project, which included the GET, POST, PUT, and DELETE routes. Next, we learned to test routes through Postman.

Next, we separated the controller part and created a controller for getting the tasks, creating the tasks, updating the tasks, and deleting the tasks. Then, we learned to use JSON in our project. Finally, we learned to use the in-built NodeJS error handler in our project.

In the next chapter, we will learn about the MongoDB database, and how to connect our server to MongoDB using Mongoose. We will also learn about Model creation and more routes.

CHAPTER 3

MongoDB Connection and Models

Introduction

This chapter talks about the MongoDB database. Here, we will first learn to use the MongoDB database in our backend app. Next, we will be connecting MongoDB to the backend NodeJS code through the package of mongoose. Lastly, we will learn to create routes in our Express app.

Structure

In this chapter, we are going to discuss the following topics:

MongoDB database

Connecting through Mongoose

Model creation

Creating routes

MongoDB Database

Before creating the MongoDB database, we have a small task remaining. We will add the package of express-async-handler. Now this package helps us to get rid of try..catch block, which is common express code.

```
npm i express-async-handler
```

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$ npm i express-async-handler
added 1 package, and audited 117 packages in 2s

12 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities

naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$
```

Figure 3.1: Installing express async handler

Now, we will first import the express-async-handler in our project. After that, we will wrap all functions in taskController.js file with asyncHandler.

```
const asyncHandler = require('express-async-handler')
```

File Edit Selection View Go Run Terminal ... taskController.js - backend - Visual Studio Code

JS server.js JS taskRoutes.js JS taskController.js X JS errorMiddleware.js

```
controllers > JS taskController.js > [§] deleteTask
1 const asyncHandler = require('express-async-handler')
2
3 const getTasks = [asyncHandler(async (req, res) => {
4     res.status(200).json({ message: 'Get All Tasks' });
5 })
6
7 const setTask = [asyncHandler(async (req, res) => {
8     if(!req.body.text){
9         res.status(400)
10        throw new Error('Please enter a task')
11    }
12    res.status(200).json({ message: 'Create Task' });
13 })
14
15 const updateTask = [asyncHandler(async (req, res) => {
16    res.status(200).json({ message: `Task ${req.params.id} updated.` });
17 })
18
19 const deleteTask = [asyncHandler(async (req, res) => {
20    res.status(200).json({ message: `Task ${req.params.id} deleted.` });
21 })
22
23 module.exports = { getTasks, setTask, updateTask, deleteTask }
```

Ln 21, Col 3 Spaces:4 UTF-8 CRLF () JavaScript Go Live ⌂ ⌂

Figure 3.2: Adding express async handler

Now, we will use MongoDB on the cloud. So, go to <https://cloud.mongodb.com/>, and click Google and sign in or sign up with your Gmail account.

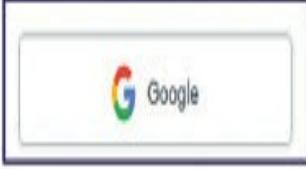
Log in | MongoDB

https://account.mongodb.com/account/login

MongoDB.

Log in to your account

Don't have an account? [Sign Up](#)

 Google

 GitHub

Or with email and password

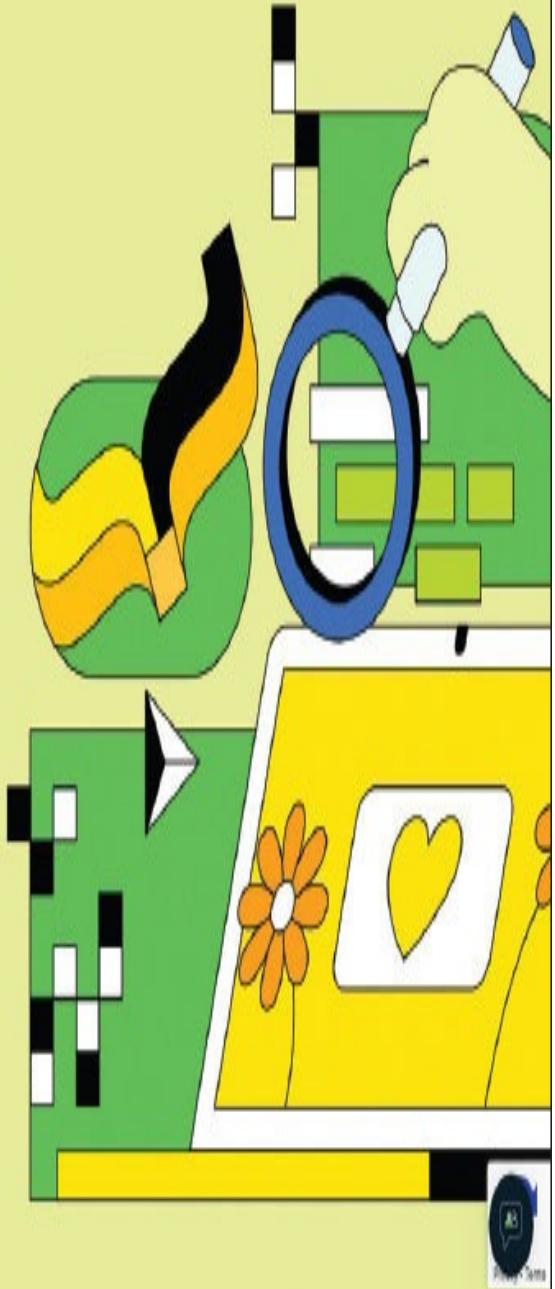
Email Address

Next

MongoDB.local is coming to a city near you!

Enjoy technical deep-dives, one-on-one expert advice, product announcements, and more. Join us for a day-long event to elevate your skills.

[View the Schedule →](#)



Privacy Terms

Figure 3.3: MongoDB Cloud

Basic Setup

If you have a lot of projects, then it will open a screen as follows. Here, click the New Project button. Even if you are logging into the MongoDB cloud for the first time, then also find the New Project button and click it.

Projects | Cloud: MongoDB Cluster X +

https://cloud.mongodb.com/v2#/org/5fd8e346c412f6786eda/projects

Atlas Nabendu's... Access Manager Billing All Clusters Get Help Nabendu

NABENDU'S ORG - 2020-12-15

ORGANIZATION Projects New Project

Alerts

Activity Feed Find a project...

Settings	Project Name	Database Deployments	Users	Teams	Alerts	Actions
Integrations	JobTrackerInitial	1 Deployment	2 Users	0 Teams	0 Alerts	
Access Manager	MEAN	1 Deployment	1 User	0 Teams	0 Alerts	
Billing	blog-nextjs	1 Deployment	1 User	0 Teams	0 Alerts	
Support	dating-app-mern	1 Deployment	1 User	0 Teams	0 Alerts	
Live Migration	ecommerce-backend-db	1 Deployment	1 User	0 Teams	0 Alerts	
	employeeDB	1 Deployment	1 User	0 Teams	0 Alerts	
	facebook-mern-backend	1 Deployment	1 User	0 Teams	0 Alerts	
	graphql-react	1 Deployment	1 User	0 Teams	0 Alerts	
	instagram-mern-backend	1 Deployment	1 User	0 Teams	0 Alerts	
	mean-sequatich	1 Deployment	1 User	0 Teams	0 Alerts	
	mern-netflix-db	1 Deployment	1 User	0 Teams	0 Alerts	

Figure 3.4: New Project

On the next screen, we need to give the project a name, which is MERN-Task-App in our case. After that, click the Next button.

Create Project Cloud:Mongo X +

https://cloud.mongodb.com/v2#/org/5fd8e3f4fc41f2f5706en0a/projects/create

Atlas Nebendu's... Access Manager Billing All Clusters Get Help Nabendu

NABENDU'S ORG - 2020-12-15 > PROJECTS

Projects Create a Project

Alerts 0

Activity Feed Name Your Project Add Members

Settings

Integrations Name Your Project

Access Manager Project names have to be unique within the organization (and other restrictions).

Billing MERN-TaskApp

Support

Live Migration

Cancel Next

System Status: All Good Last Login: 17:49:53,240

©2023 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Feedback

Figure 3.5: Project Name

On the next screen, it will confirm your email id. Here, just click the Create Project button.

Create Project | Cloud: MongoDB +

https://cloud.mongodb.com/v2#/org/Sfd8e346c41f216786ea0a/projects/create

Atlas Nebendu's... Access Manager Billing All Clusters Get Help Nabendu

NABENDU'S ORO - 2020-12-11 > PROJECTS

Projects Create a Project

Alerts

Activity Feed

Settings

Integrations

Access Manager

Billing

Support

Live Migration

Name Your Project Add Members

Add Members and Set Permissions

Invite new or existing users via email address...

Give your members access permissions below.

nabendu.biswas@gmail.com (you) Project Owner

Cancel Create Project

Project Member Permissions

Project Owner Has full administration access

Project Cluster Manager Can update clusters

Project Data Access Admin Can access and modify a cluster's data and indexes, and kill operations

Project Data Access Read/Write Can access a cluster's data and indexes, and modify data

Project Data Access Read Only Can access a cluster's data and indexes

Project Search Index Editor Can view and manage a cluster's search indexes

Project Read Only May only modify personal

Figure 3.6: Project Permissions

The next screen will allow us to create a database. Here, click the Build a Database button.

Database Deployments - Cloud X

https://cloud.mongodb.com/v2/6477e3d31046d1302a525528#/clusters

Atlas Nebendu's... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

DEPLOYMENT NABENDU'S ORO · 2020-12-15 > MERN-TASK-APP

Database Data Lake REVIEW

SERVICES Triggers Data API Data Federation Search

SECURITY Backup Database Access Network Access Advanced

Create a database

Choose your cloud provider, region, and specs.

Build a Database

Once your database is up and running, live migrate an existing MongoDB database into Atlas with our Live Migration Service.

New On Atlas 1

Goto System Status: All Good

©2023 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

AI

Figure 3.7: Build a Database

On the next screen, click the M0 tier which is free. Also, keep the provider as AWS.

The screenshot shows the MongoDB Cloud Starter Templates page. At the top, there's a navigation bar with a magnifying glass icon and the text "Deploy your database". Below the navigation bar, the URL is https://cloud.mongodb.com/v2/6477e3d31046d1302a525528/clusters/starter/templates.

MongoDB.

Deploy your database

Use a template below or set up advanced configuration options. You can also edit these configuration options once the cluster is created.

Cluster Type	Cost
M10	\$0.08/hour
SERVERLESS	\$0.10/1M reads
M0	FREE

M10 (\$0.08/hour)
For production applications with sophisticated workload requirements.
STORAGE: 10 GB, RAM: 2 GB, vCPU: 2 vCPUs

SERVERLESS (\$0.10/1M reads)
For application development and testing, or workloads with variable traffic.
STORAGE: Up to 1TB, RAM: Auto-scale, vCPU: Auto-scale

M0 (FREE)
For learning and exploring MongoDB in a cloud environment.
STORAGE: 512 MB, RAM: Shared, vCPU: Shared

Provider: AWS (selected), Google Cloud, Azure

FREE

Create

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

[I'd like to see more](#)

[Access Advanced Configuration](#)

Figure 3.8: Deploy a Database

Scroll a bit down and choose the Region in your country. We have chosen Mumbai as we are located in India. We have also changed the Name to NabenduCluster. After that, click the Create button.

Deploy your database | Cloud X +

https://cloud.mongodb.com/v2/6477e3d31046d1302a525528#/clusters/starterTemplates

For production applications with sophisticated workload requirements.

STORAGE	RAM	vCPU
10 GB	2 GB	2 vCPUs

For application development and testing, or workloads with variable traffic.

STORAGE	RAM	vCPU
Up to 1 TB	Auto-scale	Auto-scale

For learning and exploring MongoDB in a cloud environment.

STORAGE	RAM	vCPU
512 MB	Shared	Shared

Provider

AWS **Google Cloud** **Azure**

Region ★ Recommended region ⓘ

Mumbai(ap-south-1) ▼

Name

You cannot change the name once the cluster is created.

NabenduCluster

FREE

Create

Free forever! Your MO cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

[I'll deploy my database later](#)

[Access Advanced Configuration](#)

Figure 3.9: Region of Database

On the next screen, it will ask for username and password. To keep it simple for the tutorial we have given it as admin. After that, click the Create User button.

Security Quickstart | MongoDB X +

https://cloud.mongodb.com/v2/5477e3d31046d1302a525528#setup/access?includeToast=true

Atlas Nabendu's... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

To access data stored in Atlas, you'll need to create users and set up network security controls. Learn more about security setup.

DEPLOYMENT 1 How would you like to authenticate your connection?

Database Your first user will have permission to read and write any data in your project.

Data Lake PREVIEW

SERVICES

Triggers

Data API

Data Federation

Search

SECURITY

Quickstart

Backup

Database Access

Network Access

Advanced

New On Atlas 1

Goto

Username and Password Certificate

We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information.

Create a database user using a username and password. Users will be given the `read` and `write` to any database privilege by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username

Password Autogenerate Secure Password Copy

Creates User

Figure 3.10: Create User

Creating the user, will automatically add your current IP address. If you try to click Add Entry, it will give an error that This IP address has already been added. Lastly, click the Finish and Close button.

Security Quickstart | MongoDB X +

https://cloud.mongodb.com/v2/647e3d31046d1302a525528#setup/access?includeToast=true

Atlas Nebendu's... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

Add entries to your IP Access List

DEPLOYMENT

Database Data Lake PREVIEW

SERVICES

Triggers Data API Data Federation Search

SECURITY

Quickstart Backup Database Access Network Access Advanced

New On Atlas 1

Goto System Status: All Good

©2023 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Only an IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the Network Access Page.

IP Address Description

Enter IP Address Enter description Add My Current IP Address

Add Entry

This IP address has already been added.

IP Access List Description

1/32 My IP Address EDIT REMOVE

Finish and Close

The screenshot shows the MongoDB Cloud interface for managing an IP Access List. On the left, a sidebar lists deployment components like Database, Data Lake, and various services. The main area is titled 'Add entries to your IP Access List'. It includes fields for 'IP Address' and 'Description', and a button to 'Add Entry'. A message box states that the current IP address has already been added. Below this, a table shows the current IP access list entry: 'My IP Address' with edit and remove buttons. At the bottom right, a large green 'Finish and Close' button is highlighted with a purple border.

Figure 3.11: IP Address

Now, we need to do one more setup for our application to run properly. Click the Network Access tab. After that click the +ADD IP ADDRESS button.

Network Access | Cloud MongoDB

https://cloud.mongodb.com/v2/6477e3d31046d1302a525528/security/network/accessList

Atlas Nabendu's... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

DEPLOYMENT NABENDU'S ORG - 2020-12-15 > MERN-TASK-APP

Database Network Access

Data Lake PREVIEW IP Access List Peering Private Endpoint

SERVICES Triggers ADD IP ADDRESS

Data API You will only be able to connect to your cluster from the following list of IP Addresses:

Data Federation

Search IP Address Comment Status Actions

SECURITY 1/32 (includes your current IP address) My IP Address Active EDIT DELETE

Backup

Database Access

Network Access

Advanced

New On Atlas 1

Goto System Status: All Good

©2023 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Figure 3.12: Network Address Configuration

In the pop-up, click on ALLOW ACCESS FROM ANYWHERE button and then the Confirm button.

Network Access Cloud: MongoDB

https://cloud.mongodb.com/v2/6477e3d31046d1302a525528#security/network/accessList/addToAccessList

All Clusters Get Help Nabendu

MERN-Task-App Data Services

NABENDU'S ORG - 2023

DEPLOYMENT Database Dot Lake PREVIEW

SERVICES Triggers Data API Data Federation Search SECURITY Backup Database Access Network Access Advanced New On Atlas 1

System Status: All Good

©2023 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Add IP Access List Entry

Atlas only allows client connections to a cluster from entries in the project's IP Access List. Each entry should either be a single IP address or a CIDR-notated range of addresses. Learn more.

ALLOW ACCESS FROM ANYWHERE

Access List Entry: 0.0.0.0/0

Comment: Optional comment describing this entry

This entry is temporary and will be deleted in 6 hours

Cancel Confirm

+ ADD IP ADDRESS

Actions EDIT DELETE

The screenshot shows the 'Add IP Access List Entry' dialog box overlaid on the MongoDB Atlas interface. The dialog box contains a single entry '0.0.0.0/0'. A green 'CONFIRM' button is highlighted with a blue border. The background shows the main interface with tabs like 'Data Services', 'Deployment', and 'Network Access'.

Figure 3.13: Allow access from Anywhere

Adding Collection and Connection URL

Now, we will add a Collection in our project. For this, click the Database tab and after that click Browse Collections button.

Database Deployments Cloud X +

← → C https://cloud.mongodb.com/v2/5477e3d31046d1302a525528#clusters

Atlas Nabendu's... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

DEPLOYMENT NABENDU'S ORG - 2020-12-15 > MERN-TASK-APP

Database PREVIEW Find a database deployment... + Create

SERVICES

Triggers NabenduCluster Connect View Monitoring Browse Collections ... FREE SHARED

DATA LAKE

Search Visualize Your Data R: 0 Connections: 0 In: 0.0 B/s Data Size: 0.0 B

W: 0 Lost: 40 seconds Lost: 40 seconds Lost: 40 seconds Last: 40 seconds

SECURITY Build dashboards and charts, and embed them in your apps with MongoDB Charts.

Backup Dismiss Explore Charts

Database Access

Network Access

Advanced

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED APP SERVICES	ATLAS SQL	ATLAS SEARCH
6.0.6	AWS / Mumbai (ap-south-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Connect	Create Index

New On Atlas 1

Date

System Status: All Good

©2023 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Figure 3.14: Browse Collections

On the next screen, click the Add My Own Data button.

DATA | Cloud: MongoDB Cloud X +

https://cloud.mongodb.com/v2/6477e3d31046d1302a525528#metrics/replicaSet/6477e4c5a432a4367ef23:1

Atlas Nabendu's... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

NABENDU'S ORG · 2020-12-16 > MERN-TASK-APP > DATABASES

DEPLOYMENT Database PREVIEW VERSION REGION

NabenduCluster 6.0.6 AWS Mumbai (ap-south-1)

Data Lake SERVICES Overview Real Time Metrics Collections Search Profiler Performance Advisor Online Archive Cmd Line Tools

DATABASES: 0 COLLECTIONS: 0 REFRESH

Triggers

Data API

Data Federation

Search

SECURITY

Backup

Database Access

Network Access

Advanced

Ops

Explore Your Data

- Find: run queries and interact with documents
- Indexes: build and manage indexes
- Aggregation: test aggregation pipelines
- Search: build search indexes

Load a Sample Dataset Add My Own Data

Learn more in Docs and Tutorials ⓘ

Feedback

Figure 3.15: Add My Own Data

In the pop-up, give the database name and collection name. We have given it taskApp and tasks, respectively.

Data | Close MongoDB Cloud +

https://cloud.mongodb.com/v2/5477e3d31046d1302a525528#metrics/replicaSet/5477e4c5a432a43670ef23a

Atlas Nebendu's ... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

DEPLOYMENT NABENDU'S ORG - 2020-12-15 MERN-TASK-APP > DATABASES VERSION REGION 6.0.6 AWS Mumbai (ap-south-1)

Database Nabendu

Data Lake PREVIEW Overview Recs

SERVICES DATABASES: 0 COLLECTIONS: 0 Triggers Refresh

Data API Data Federation Search

SECURITY Backup Database Access Network Access Advanced

Create Database

Database name taskApp

Collection name tasks

Additional Preferences Select

Cancel Create

- Search: build search indexes

Load a Sample Dataset Add My Own Data

Learn more in Docs and Tutorials

Goto

This screenshot shows the MongoDB Cloud interface for creating a new database. A modal window titled 'Create Database' is open, prompting for a 'Database name' (set to 'taskApp') and a 'Collection name' (set to 'tasks'). Below these fields is an 'Additional Preferences' dropdown set to 'Select'. At the bottom right of the modal are 'Cancel' and 'Create' buttons, with 'Create' being highlighted with a purple border. The background shows the main MongoDB Cloud dashboard for the 'MERN-Task-App' deployment, including tabs for Data Services, App Services, and Charts, and sections for Deployment, Services, Security, and Metrics.

Figure 3.16: Create Database

Now, click the Overview tab again and then the Connect button.

NabenduCluster Overview | C

https://cloud.mongodb.com/v2/6477e3d31046d1302a525528#/clusters/detail/NabenduCluster

Atlas Nabendu's... Access Manager Billing All Clusters Get Help Nabendu

MERN-Task-App Data Services App Services Charts

DEPLOYMENT NABENDU'S ORG - 2020-12-15 > MERN-TASK-APP > DATABASES

NabenduCluster

VERSION 6.0.6 REGION AWS Mumbai (ap-south-1) CLUSTER TIER M0 Sandbox (General)

Data Lake PREVIEW SERVICES Triggers Data API

REGION Mumbai (ap-south-1)

SEARCH: 00...shard-00-00.a.. SECONDARY
00...shard-00-01.a.. SECONDARY
00...shard-00-02.a.. PRIMARY

SECURITY Backup Database Access Network Access Advanced

New On Atlas 1 Goto

This is a Shared Tier Cluster
If you need a database that's better for high-performance production applications, upgrade to a dedicated cluster.

Upgrade

Operations R: 0 W: 0 Last 6 Hours

Logical Size 0.0 B 512.0 MB max Last 6 Hours

Connections 7 500 max Last 6 Hours

Connect Configuration ...

The screenshot displays the MongoDB Atlas cluster overview for 'NabenduCluster'. At the top, it shows the deployment path: NABENDU'S ORG - 2020-12-15 > MERN-TASK-APP > DATABASES. The main cluster information is summarized: VERSION 6.0.6, REGION AWS Mumbai (ap-south-1), and CLUSTER TIER M0 Sandbox (General). Below this, there are tabs for Data Lake (selected) and PREVIEW, along with a SERVICES section containing Triggers, Data API, and Data Federation (REGION Mumbai [ap-south-1]). The shard status section lists three shards: shard-00-00.a.. (SECONDARY), shard-00-01.a.. (SECONDARY), and shard-00-02.a.. (PRIMARY). A prominent message states 'This is a Shared Tier Cluster' and suggests upgrading to a dedicated cluster for high-performance production applications. Monitoring sections include Operations (R: 0 W: 0), Logical Size (0.0 B / 512.0 MB max), and Connections (7 / 500 max). A central 'Upgrade' button is visible. The bottom of the page includes links for New On Atlas and Goto, and a feedback icon.

Figure 3.17: Connect to cluster

Now, in the pop-up click the MongoDB for VS Code button.

NabenduCluster Overview | Cloud

https://cloud.mongodb.com/v2/6477e3d31046d1302a525528#/clusters/detail/NabenduCluster/connect?clusterId=6477e3d31046d1302a525528

Atlas Nebendu's...

MERN-Task-App

DEPLOYMENT

Database

Data Lake PREVIEW

OVERVIEW

SERVICES

Triggers

Data API

Data Federation

Search

SECURITY

Backup

Database Access

Network Access

Advanced

New On Atlas 1

Goto

Set up connection security

Choose a connection method

Connect

Clusters Get Help Nabendu

CLUSTER TIER M0 Sandbox (General)

Online Archive Cmd Line Tools

Connect Configuration ...

Connect to your application

Drivers

Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)

Compass

Explore, modify, and visualize your data with MongoDB's GUI

Shell

Quickly add & update data using MongoDB's Javascript command-line interface

MongoDB for VS Code

Work with your data in MongoDB directly from your VS Code environment

Atlas SQL

Easily connect SQL tools to Atlas for data analysis and visualization

Go Back Close

Figure 3.18: MongoDB for VS Code

In the next screen, we will get the Connection String. This connection string will be used to connect to MongoDB from the backend in the next section.

Atlas Nebendu's... Connect

MERN-Task-App Data

DEPLOYMENT NABENDU'S

Database

Data Lake PREVIEW

SERVICES

Triggers

Data API

Data Federation

Search

SECURITY

Backup

Database Access

Network Access

Advanced

New On Atlas 1 Goto

Setup connection security Choose a connection method Connect

Connecting with MongoDB for VS Code

1. Install MongoDB for VS Code.

In VS Code, open "Extensions" in the left navigation and search for "MongoDB for VS Code." Select the extension and click install.

2. In VS Code, open the Command Palette.

Click on "View" and open "Command Palette." Search "MongoDB: Connect" on the Command Palette and click on "Connect with Connection String."

3. Connect to your MongoDB deployment.

Paste your connection string into the Command Palette.

```
mongodb+srv://admin:<password>@nabenducluster.ampq7ug.mongodb.net/
```

Replace `<password>` with the password for the `admin` user.
When entering your password, make sure all special characters are URL encoded.

4. Click "Create New Playground" in MongoDB for VS Code to get started.

Learn more about Playgrounds.

RESOURCES

Connect to MongoDB through VSCode Explore your data with playgrounds
Access your Database Users Troubleshoot Connections

Go Back Close

Figure 3.19: MongoDB Connection url

[Connecting through Mongoose](#)

Back to VS Code, open the .env file and create a variable MONGO_URI. Put the connection url in it, which we have got in the previous section. Notice that we have added the password and the correct database name, which is taskApp in it.

A screenshot of a dark-themed code editor, likely Visual Studio Code, showing a file named `env-backed-VisualStudioCode.js`. The file contains three environment variable assignments:

```
1 NODE_ENV = development
2 PORT = 8080
3 MONGO_URI = mongodb+srv://admin:admin@habencloud-test-app/test?log=mongodb.net/tas/app
```

Figure 3.20: MONGO_URI in env file

Now, create a new folder called connect and new file database.js inside connect. Then place the following content in database.js file:

```
const mongoose = require('mongoose')

const connectDB = async () => {
  try {
    const connect = await mongoose.connect(process.env.MONGO_URI)
    console.log(`MongoDB Connected: ${connect.connection.host}`)
  } catch (err) {
    console.log(err)
    process.exit(1)
  }
}

module.exports = connectDB
```

Here, we are first importing mongoose. After that, we have a connectDB function, inside which we have a try...catch block. Here, in the try block we are

first connecting through mongoose to our MONGO_URI stored in the .env file. We are also console logging the connected message.

In the catch block, we are just showing the error and exit the process.

The screenshot shows a Visual Studio Code interface with the following details:

- File Bar:** File Edit Selection View Go Run Terminal Help database.js - backend - Visual Studio Code
- Explorer:** Shows the project structure under BACKEND:
 - connect (highlighted with a green border)
 - database.js (highlighted with a blue background)
 - controllers
 - middleware
 - node_modules
 - routes
 - .env
 - .gitignore
 - package-lock.json
 - package.json
 - server.js
- Code Editor:** Displays the contents of the database.js file:

```
1 const mongoose = require('mongoose')
2
3 const connectDB = async () => {
4   try {
5     const connect = await mongoose.connect(process.env.MONGO_URI)
6     console.log(`MongoDB Connected: ${connect.connection.host}`)
7   } catch (err) {
8     console.log(err)
9     process.exit(1)
10 }
11 }
12
13 module.exports = connectDB
```

Figure 3.21: database.js file

Next, go to server.js file and first import the connectDB function. Next, we are just calling it.

The screenshot shows a dark-themed instance of Visual Studio Code with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** server.js - backend - Visual Studio Code
- Editor:** The main editor area contains the `server.js` file. The code is as follows:

```
1 const express = require('express');
2 const { errorHandler } = require('./middleware/errorMiddleware');
3 const dotenv = require('dotenv').config();
4 const connectDB = require('./connect/database');
5 const port = process.env.PORT || 5000;
6
7 [connectDB()];
8 const app = express();
9
10 app.use(express.json());
11 app.use(express.urlencoded({ extended: false }));
12
13 app.use('/api/tasks', require('./routes/taskRoutes'));
14
15 app.use(errorHandler);
16
17 app.listen(port, () => console.log(`Server listening on ${port}`));
```

Two specific lines of code are highlighted with green boxes:

- `const connectDB = require('./connect/database');`
- `[connectDB()];`

Figure 3.22: Using database in server.js

Back in the terminal, restart the server with the npm run dev command. If everything is right we will also get the MongoDB Connected with the server name log.

```
C:\WINDOWS\system32\cmd. X + X

naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$ npm run dev

> backend@1.0.0 dev
> nodemon server.js

[nodemon] 2.0.22
[nodemon] to restart at any time, enter 'rs'
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,jsx,json
[nodemon] starting 'node server.js'
Server listening on 8080
MongoDB Connected: ac-6m2kq0p-shard-00-00.anpq7ug.mongodb.net
```

Figure 3.23: Command line

Model Creation

Models are required for the backend to interact with the database. They are used to do the CRUD operation in the database. The Models are the way to represent the data in the database. So, create a new folder models in the project and a file taskModel.js in models folder. Then, place the following content in taskModel.js file to create Task model:

```
const mongoose = require('mongoose')

const taskSchema = mongoose.Schema(
  {
    text: { type: String, required: [true, 'Please add a text value'] },
  },
  {
    timestamps: true
  }
)

module.exports = mongoose.model('Task', taskSchema)
```

Here, we have a taskSchema, where we have a field of text. This field is of type String and also required. It will give the error Please add a text value if we don't send it on request.

File Edit Selection View Go Run Terminal Help taskModel.js - backend - Visual Studio Code

EXPLORER ⌂ server.js taskModel.js X ...

BACKEND models > taskModel.js > ...

connect

controllers

middleware

models

taskModel.js

node_modules

routes

env

.gitignore

package-lock.json

package.json

server.js

```
const mongoose = require('mongoose')
const taskSchema = mongoose.Schema({
  text: { type: String, required: [true, 'Please add a text value'] },
  timestamps: true
})
module.exports = mongoose.model('Task', taskSchema)
```

Figure 3.24: Creating models

Now, in the taskController.js file we will first import the taskModel. And after that in the getTasks() method, we will use the find() function of mongoose to get all tasks from the database. Lastly, we are sending the status 200 and the tasks back.

```
const Task = require('../models/taskModel')
```

```
const tasks = await Task.find()
```

```
res.status(200).json(tasks);
```

A screenshot of the Visual Studio Code interface showing the file `taskController.js`. The code implements two endpoints: `getTasks` and `setTask`. The `getTasks` endpoint uses `express-async-handler` to wrap the `Task.find()` method from the `taskModel` model. The `setTask` endpoint checks if a task body is provided; if not, it returns a 400 error. Otherwise, it creates a new task and returns a 200 response with a success message.

```
1 const asyncHandler = require('express-async-handler')
2 const Task = require('../models/taskModel')
3
4 const getTasks = asyncHandler(async (req, res) => {
5   const tasks = await Task.find()
6   res.status(200).json(tasks)
7 })
8
9 const setTask = asyncHandler(async (req, res) => {
10   if(!req.body.text){
11     res.status(400)
12     throw new Error('Please enter a task')
13   }
14   res.status(200).json({ message: 'Create Task' });
15 })
16
```

Figure 3.25: Updating getTasks

Now, in any browser go to `http://localhost:8000/api/tasks`, and we will get an empty array, since we don't have any tasks.

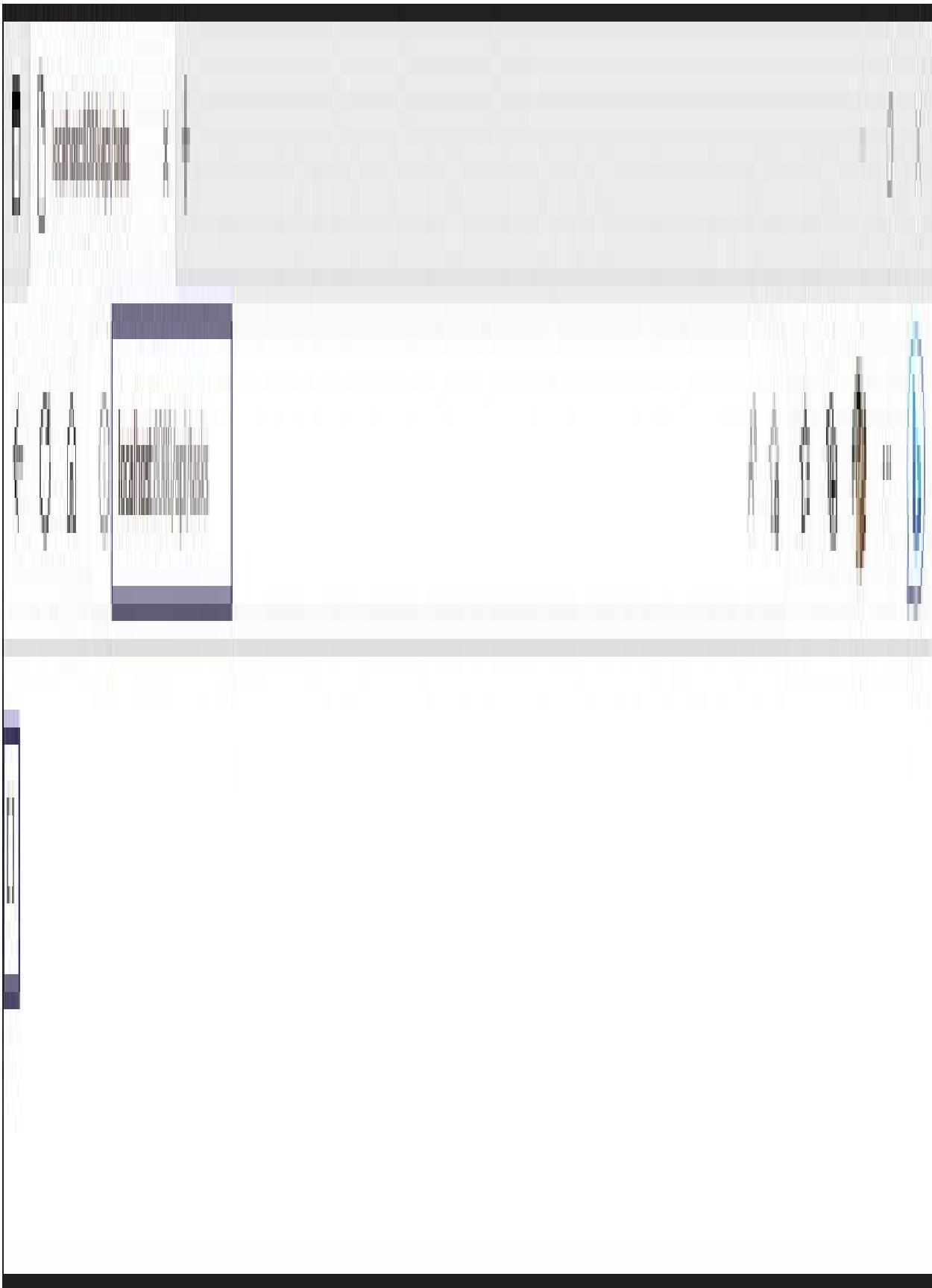


Figure 3.26: Checking all Tasks

Creating Routes

We have already updated the getTasks route and now we will create more routes. Now, we will update the rest of the routes, so that we get basic working routes.

POST Route

First, we will update the PUT route. So, in the setTask() function we will use the create() method from mongoose and pass the text. Notice, that we have to use the Task model to run the create() method. Lastly, we are sending the status 200 and the task back.

```
const task = await Task.create({ text: req.body.text })  
res.status(200).json(task);
```

File Edit Selection View Go Run Terminal Help taskController.js - backend - Visual Studio Code

JS server.js JS taskModel.js JS taskController.js X ⋮

controllers > JS taskController.js > [0] setTask

```
1 const asyncHandler = require('express-async-handler')
2 const Task = require('../models/taskModel')
3
4 const getTasks = asyncHandler(async (req, res) => {
5   const tasks = await Task.find()
6   res.status(200).json(tasks);
7 })
8
9 const setTask = asyncHandler(async (req, res) => {
10   if(!req.body.text){
11     res.status(400)
12     throw new Error('Please enter a task')
13   }
14
15   const task = await Task.create({ text: req.body.text })
16   res.status(200).json(task),
17 })
18
19 const updateTask = asyncHandler(async (req, res) => {
20   res.status(200).json({ message: `Task ${req.params.id} updated.` });
21 })
22
```

Figure 3.27: Updating setTasks

Back in Postman, we will send a text with value and we will get back a Status of 200. Also, the json will be returned back to us.

Home Workspaces API Network Explore

Search Postman

Post Task

MERN-Task / Post Task

POST http://localhost:8000/api/tasks

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

History none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description	Bulk Edit
text	Learn Flutter		
Key	Value	Description	

Body Cookies Headers (7) Test Results

Status: 200 OK Time: 100 ms Size: 370 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 "text": "Learn Flutter",
2 "_id": "64784251039ff185881ea812",
3 "createdAt": "2023-06-01T07:01:37.357Z",
4 "updatedAt": "2023-06-01T07:01:37.357Z",
5 "__v": 0
```

Online Find and Replace Console

Cookies Capture requests Runner Trash

Figure 3.28: Sending Post Request

We have also added one more task through Postman. And after that, we checked in the MongoDB Cloud database. Here, we can find both tasks being saved in the database.

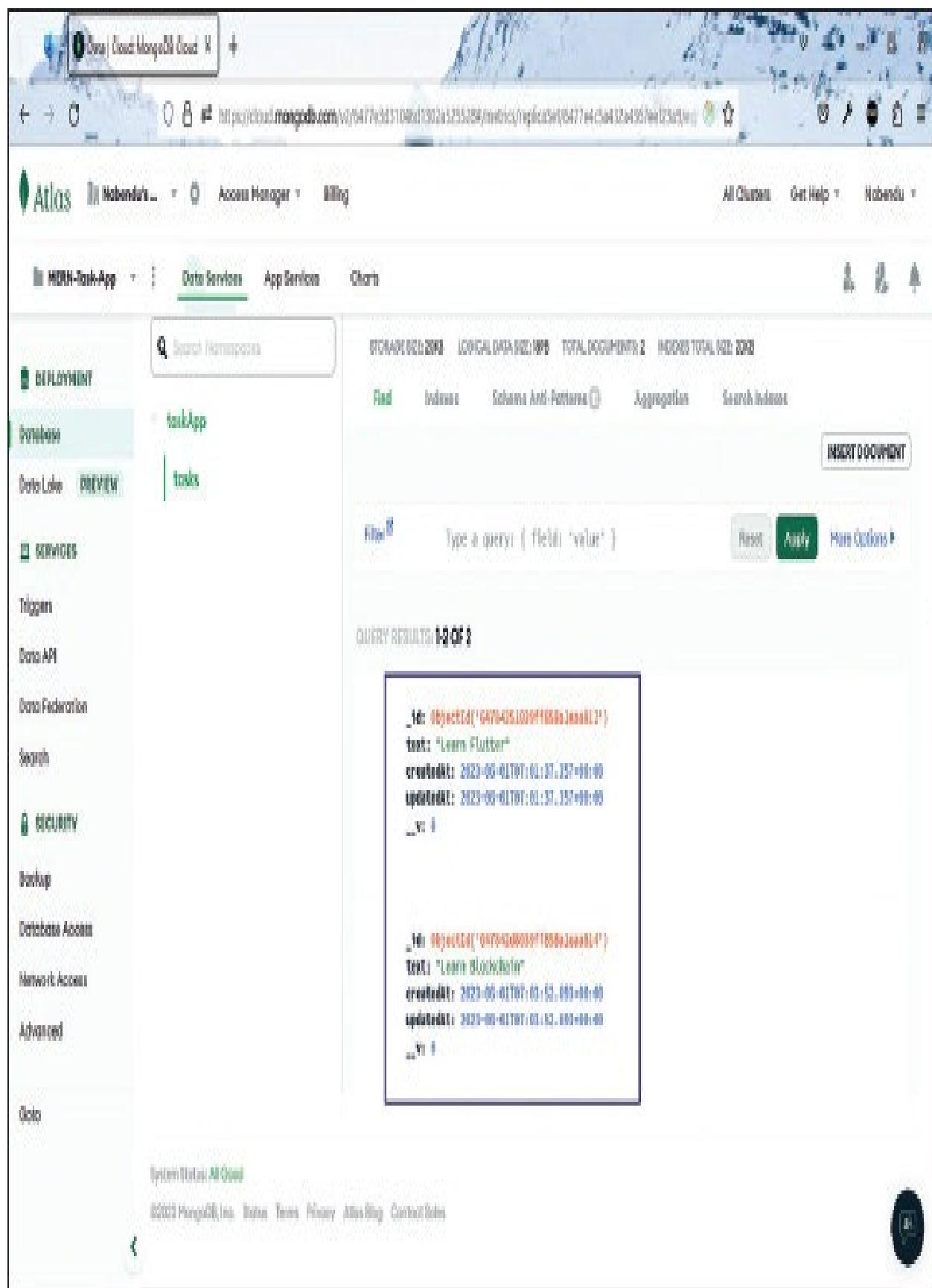


Figure 3.29: Checking Tasks in MongoDB

Again, in any browser goto <http://localhost:8000/api/tasks> and we will get an array of objects. It contains the two tasks saved previously.



Figure 3.30: Checking Tasks in Browser

PUT Route

Now, we will update the complete code for the PUT route. In the updateTask() function, add the following code:

```
const task = await Task.findById(req.params.id)
```

```
if (!task) {  
  res.status(400)  
  throw new Error('Task not found')  
  
}  
  
}
```

```
const updatedTask = await Task.findByIdAndUpdate(req.params.id, req.body,  
{ new: true })
```

```
res.status(200).json(updatedTask)
```

Here, we are first using the findById method to find the task. Here, we are passing the req.params.id in it, which we get from the request header. After that, we are checking if no task is there and here we are throwing an error.

Next, we are updating the task with findByIdAndUpdate method. Here, we are

passing the params, body and also new: true. The new: true returns us the latest updated value from the database. Lastly, we are sending the status 200 and the updatedTask back.

File Edit Selection View Go Run Terminal Help taskController.js - backend - Visual Studio Code

server.js taskModel.js taskController.js

```
controllers > taskController.js > updateTask > asyncHandler() callback
```

```
13
14
15     const task = await Task.create({ text: req.body.text })
16     res.status(200).json(task);
17 }
18
19 const updateTask = asyncHandler(async (req, res) => {
20     const task = await Task.findById(req.params.id)
21
22     if (!task) {
23         res.status(400)
24         throw new Error('Task not found')
25     }
26
27     const updatedTask = await Task.findByIdAndUpdate(req.params.id, req.body, { new: true })
28     res.status(200).json(updatedTask)
29 }
30
31 const deleteTask = asyncHandler(async (req, res) => {
32     res.status(200).json({ message: `Task ${req.params.id} deleted.` })
33 }
34
35 module.exports = { getTasks, setTask, updateTask, deleteTask }
```

In 28, Col 37 Spaces: 4 UTF-8 CR/LF {} JavaScript Go Live

Figure 3.31: Updating updateTask

Back in Postman, we are sending a PUT request. Here, we have updated the text of a task. On sending it we are getting a Status 200 and also the updated task back.

Home Workspaces API Network Explore

Search Postman

Logout Upgrade - X

MERN-Task / Put Task

PUT http://localhost:3001/api/tasks/647842d8030ff658a1ea814

Save Send

Params Authorization Headers Body Pre-request Script Tests Settings Cookies

Method: PUT
URL: http://localhost:3001/api/tasks/647842d8030ff658a1ea814

Body Type: form-data

Key	Value	Description	Edit
text	Learn Solidity		

Body Cookies Headers (7) Test Results

Status 200 OK Time: 03 ms Size: 180 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {  
2   "_id": "647842d8030ff658a1ea814",  
3   "text": "Learn Solidity",  
4   "createdAt": "2023-06-01T07:09:52.693Z",  
5   "updatedAt": "2023-06-01T07:11:36.236Z",  
6   "__v": 0  
7 }
```

Online Print and Paste Console

Cookies Cache Network Run Test

Figure 3.32: Sending PUT Request

We can also confirm the same by getting all tasks at
<http://localhost:8000/api/tasks> and we are getting the updated task.



Figure 3.33: Checking Updated data in Browser

DELETE Route

Now, we will update the complete code for the DELETE route. In the deleteTask() function, add the following code:

```
const task = await Task.findById(req.params.id)
```

```
if (!task) {  
  res.status(400)  
  throw new Error('Task not found')  
}
```

```
await Task.findByIdAndUpdateDelete(req.params.id)  
res.status(200).json({ id: req.params.id })
```

Here, we are first using the findById method to find the task. After that, we are checking if no task is there and here we are throwing an error.

Next, we are deleting the task with findByIdAndDelete method. Here, we are passing the params. Lastly, we are sending the status 200 and the delete id back.

taskController.js X

backend > controllers > taskController.js > deleteTask > callback

```
44 const deleteTask = asyncHandler(async (req, res) => {
45   const task = await Task.findById(req.params.id)
46
47   if (!task) {
48     res.status(400)
49     throw new Error('Task not found')
50   }
51
52   await Task.findByIdAndUpdate(req.params.id)
53
54   res.status(200).json({ id: req.params.id })
55 }
56
57 module.exports = { getTasks, setTask, updateTask, deleteTask }
```

Figure 3.34: Updating deleteTask

Back in Postman we are sending a Delete request with a task id. The task is successfully deleted and we are getting a Status 200 and also the id back.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar is followed by several icons: Invite, Help, Notifications, Upgrade, and a close button. Below the header, a toolbar has buttons for POST Task, PUT Put Task, and Delete Task, along with a plus sign for creating new tasks.

The main area displays a collection named "MERN-Task / Delete Task". Underneath, a specific request is shown:

- Method:** DELETE
- URL:** <http://localhost:8000/api/tasks/64784251039ff858a1eaa812>
- Send** button (blue)

Below the URL, tabs for Params, Authorization, Headers (7), Body, Pre-request Script, Tests, and Settings are visible. The Params tab is selected. The Headers section shows two entries:

Key	Value	Description	...	Bulk Edit
Key	Value	Description	...	i

At the bottom, the Response section shows a status of 200 OK with a response time of 96 ms and a size of 260 B. The response body is displayed in JSON format:

```
1: {  
2:   "id": "64784251039ff858a1eaa812"  
3: }
```

Figure 3.35: Sending DELETE Request

We can also confirm the same by getting all tasks at <http://localhost:8000/api/tasks> and we are getting only one task back. The deleted task is not there anymore.

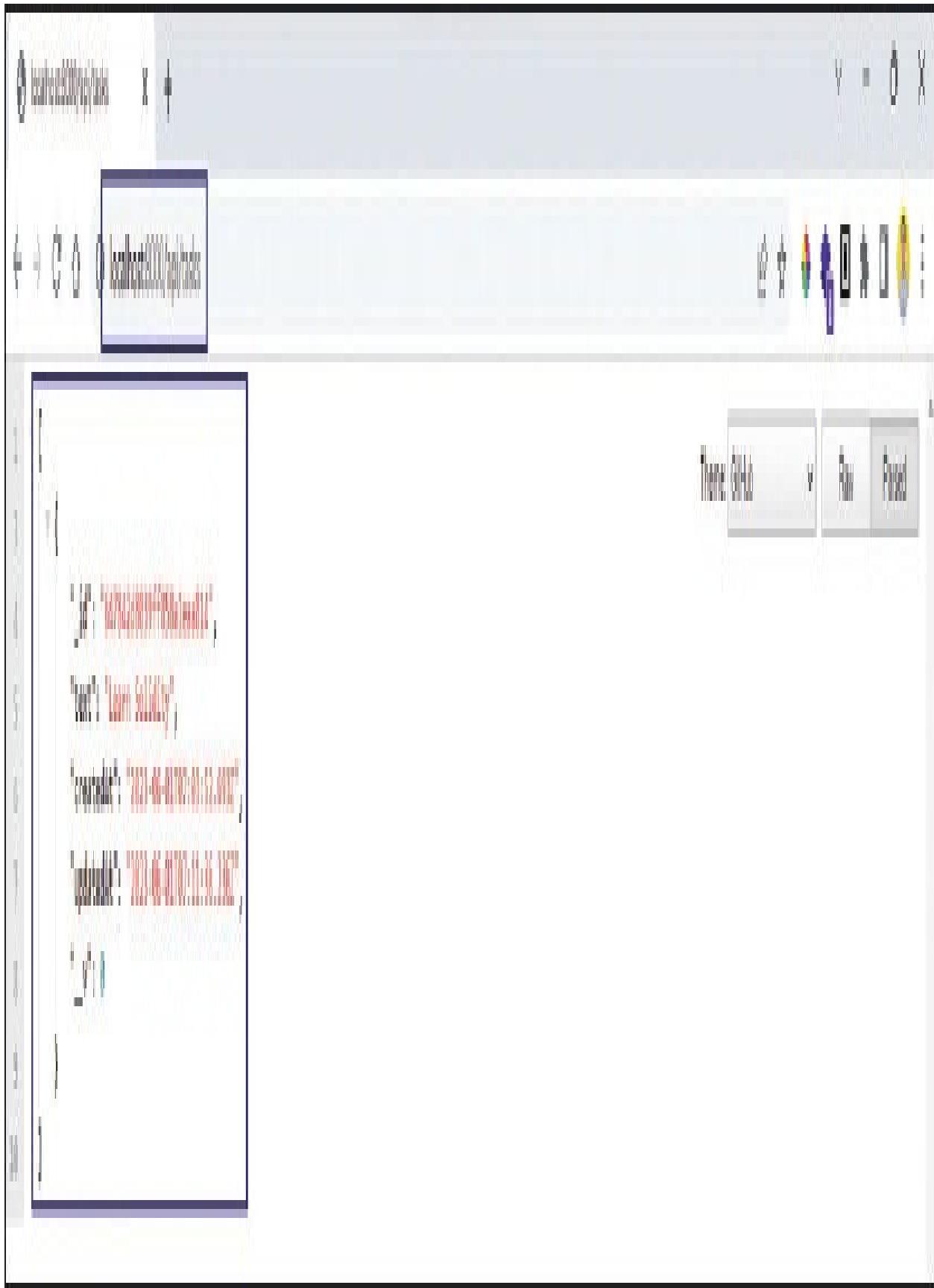


Figure 3.36: Confirming deleted data in Browser

Conclusion

In this chapter, we have learned about creating a MongoDB database through MongoDB Cloud. After that, we connected MongoDB to the backend code through the package of mongoose. Next, we did model creation which is required for CRUD operation in MongoDB through Mongoose.

Next, we have updated the GET, POST, PUT, and DELETE routes to work and do CRUD operations in the database.

In the next chapter, we will learn about JWT authentication. We will also create user model, routes, and controllers. Lastly, we will learn about hashing passwords.

Points to remember

We use the folder structure consisting of connect, controllers, middleware, routes and a server.js file, when using Express.

We are using MongoDB in the Cloud, which is easily setup using the GUI.

Routes are created using different functions from Express, like create(), findById(), findByIdAndUpdate(), findById(), and Delete() in the Controller file.

We test the different routes through the Postman app, because our frontend is not yet created.

CHAPTER 4

JWT Authentication and Hashing Password

Introduction

This chapter talks about JWT(JSON Web Token) and Hashing passwords. Here, we will first create a new user model in our backend app. We will create the routes and the controller for the same. Next, we learn to hash the password with the library of bcrypt. Lastly, we use the package of jsonwebtoken to generate JWT and add authentication to our app.

Structure

In this chapter, we are going to discuss the following topics:

User Model and Controller

Register User with Hashed Password

Login User

Understanding JWT

Using JWT

User Model and Controller

Now, we will create the user model because we are going to store the User data also in our database. So, create a new file userModel.js inside the models folder and add the following code in userModel.js:

```
const mongoose = require('mongoose')
```

```
const userSchema = mongoose.Schema(
```

```
{
```

```
  name: {
```

```
    type: String,
```

```
    required: [true, 'Name is required'],
```

```
  },
```

```
  email: {
```

```
    type: String,
```

```
    required: [true, 'Email is required'],
```

```
    unique: true,
```

```
  },
```

```
  password: {
```

```
    type: String,
```

```
required: [true, 'Password is required'],  
},  
},  
{ timestamps: true }  
)  
  
module.exports = mongoose.model('User', userSchema)
```

Here, in the userSchema we have three fields of name, email, and password. All of them are strings and required. The email is also unique, which means we can't have the same email twice.

File Edit Selection View Go Run Terminal Help userModels-backend - Visual Studio Code

EXPLORER ⌂ server.js taskModel.js userModel.js X

RACKEND models > userModel.js > (0) userSchema > timestamps

> connect
controllers
models
taskModel.js
userModel.js

```
1 const mongoose = require('mongoose')
2
3 const userSchema = mongoose.Schema(
4   {
5     name: {
6       type: String,
7       required: [true, 'Name is required'],
8     },
9     email: {
10       type: String,
11       required: [true, 'Email is required'],
12       unique: true,
13     },
14     password: {
15       type: String,
16       required: [true, 'Password is required'],
17     },
18   },
19   [ timestamps: true ]
20 )
21
22 module.exports = mongoose.model('User', userSchema)
23
```

CODE OUTLINE ⌂ TIMELINE

In 10 Col 21 Spans 4 UTF-8 GFM () JavaScript ⌂ Go Live ⌂ Q

Figure 4.1: Creating User Model

Now, each of the tasks will also be associated with a user. So, we will update our taskModel.js file to have a field of the user. The type of it will be the ObjectId and it will refer to the User.

```
user: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User' }
```

A screenshot of the Visual Studio Code interface showing the file `taskModel.js`. The code defines a Mongoose schema for a `Task` model, which includes fields for text, user reference, and timestamps.

```
1 const mongoose = require('mongoose')
2
3 const taskSchema = mongoose.Schema(
4   {
5     text: { type: String, required: [true, 'Please add a text value'] },
6     user: { type: mongoose.Schema.Types.ObjectId, required: true, ref: 'User' }
7   },
8   {
9     timestamps: true
10 }
11 )
12
13 module.exports = mongoose.model('Task', taskSchema)
14
```

Figure 4.2: Updating Task Model

Next, we will create the new userRoutes.js file in the routes folder. In it, we are first importing express and router.

After that, we are also importing some functions from the userController file, which we are going to create next. Using the router, we have created two POST routes and one GET route.

```
const express = require('express')
const router = express.Router()

const { registerUser, loginUser, getCurrentUser } =
require('../controllers/userController')

router.post('/', registerUser)
router.post('/login', loginUser)
router.get('/current', getCurrentUser)

module.exports = router
```

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File Edit Selection View Go Run Terminal Help
- Title Bar:** userRoutes.js - backend - Visual Studio Code
- Code Editor:** The code editor displays the following content for `userRoutes.js`:

```
1 const express = require('express')
2 const router = express.Router()
3 const { registerUser, loginUser, getCurrentUser } = require('../controllers/userController')
4
5 router.post('/', registerUser)
6 router.post('/login', loginUser)
7 router.get('/current', getCurrentUser)
8
9 module.exports = router
10
```

- File Explorer:** Shows the project structure:
 - BACKEND
 - connect
 - controllers (highlighted)
 - middleware
 - models
 - taskModel.js
 - userModels.js
 - node_modules
 - routes
 - taskRoutes.js
 - userRoutes.js (highlighted)
 - env
 - .gitignore
 - package-lock.json
 - package.json
 - server.js
- Search Bar:** A search bar is located at the top right of the interface.

Figure 4.3: Creating User Routes

Now, in the controllers folder, we will create a file userController.js file. Here, we are right now creating the three functions of registerUser, loginUser, and getCurrentUser and returning simple json messages.

```
const asyncHandler = require('express-async-handler')
```

```
const registerUser = asyncHandler(async (req, res) => {
  res.json({ message: 'Register User successful' })
})
```

```
const loginUser = asyncHandler(async (req, res) => {
  res.json({ message: 'Login User successful' })
})
```

```
const getCurrentUser = asyncHandler(async (req, res) => {
  res.json({ message: 'Current user data' })
})
```

```
module.exports = { registerUser, loginUser, getCurrentUser }
```


File Edit Selection View Go Run Terminal Help userController.js - backend - Visual Studio Code

EXPLORER ... server.js userRoutes.js userController.js X userModel.js

BACKEND controllers > userController.js <unknown>

connect
controllers
taskController.js
userController.js
middleware
models
taskModel.js
userModel.js
node_modules
router
taskRoutes.js
userRoutes.js
env
ignore
package-lock.json
package.json
server.js

```
1 const asyncHandler = require('express-async-handler')
2
3 const registerUser = asyncHandler(async (req, res) => {
4   res.json({ message: 'Register User sucessful' })
5 })
6
7 const loginUser = asyncHandler(async (req, res) => {
8   res.json({ message: 'Login User sucessful' })
9 })
10
11 const getCurrentUser = asyncHandler(async (req, res) => {
12   res.json({ message: 'Current user data' })
13 })
14
15 module.exports = [ registerUser, loginUser, getCurrentUser ]
```

Figure 4.4: Creating User Controllers

Finally, we are going to create a new /api/users route in the server.js file.

```
app.use('/api/users', require('./routes/userRoutes'));
```

The screenshot shows a Visual Studio Code interface with the following details:

- File Bar:** File Edit Selection View Go Run Terminal Help
- Title Bar:** server.js - backend - Visual Studio Code
- Toolbar:** Includes icons for file operations like Open, Save, Find, and Run.
- Code Editor:** Displays the `server.js` file content. The file uses ES6 syntax and imports modules from the project's `./middleware`, `./connect`, and `./routes` directories.
- Code Content:**

```
1 const express = require('express');
2 const { errorHandler } = require('./middleware/errorMiddleware');
3 const dotenv = require('dotenv').config();
4 const connectDB = require('./connect/database');
5 const port = process.env.PORT || 5000;
6
7 connectDB();
8 const app = express();
9
10 app.use(express.json());
11 app.use(express.urlencoded({ extended: false }));
12
13 app.use('/api/tasks', require('./routes/taskRoutes'));
14 app.use('/api/users', require('./routes/userRoutes'));
15
16 app.use(errorHandler);
17
18 app.listen(port, () => console.log(`Server listening on ${port}`));
```

Figure 4.5: Updating server.js

In Postman do a POST request to `http://localhost:8000/api/users` and we will get back the correct JSON back and 200 status.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar says "Search Postman". On the right, there are icons for Invite, Help, Notifications, Upgrade, and a close button.

In the main area, there are two items listed: "Register User" and "Login User". Below them is a collection named "MERN-Task / Register User".

A specific POST request is selected, with the URL "http://localhost:8000/api/users". The "Send" button is highlighted in blue. Below the URL, tabs for Params, Authorization, Headers, Body, Pre-request Script, Tests, and Settings are visible, with "Params" being the active tab.

The "Body" section is expanded, showing a table for "Query Params". It has two rows:

Key	Value	Description	Bulk Edit

Below the table, there are tabs for Body, Cookies, Headers, and Test Results. The "Pretty" tab is selected in the Body section. The response body is displayed as:

```
1 {
2   "message": "Register User successful"
3 }
```

The status bar at the bottom right shows "Status: 200 OK Time: 27 ms Size: 272 B Save as Example".

Figure 4.6: Register route in Postman

Similarly, a POST request to `http://localhost:8000/api/users/login` will give back the correct JSON back and 200 status.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar says "Search Postman". On the right, there are icons for Invite, Help, Upgrade, and a close button.

In the main area, there's a list of collections: "MERN-Task / Login User". Below it, a specific request is selected: "POST http://localhost:3000/api/users/login".

The request details are as follows:

- Params:** Authorization, Headers (7), Body, Pre-request Script, Tests, Settings.
- Query Params:** None listed.
- Body:** None listed.

The "Send" button is highlighted in blue. Below the request details, the response status is shown as "Status: 200 OK" with a time of "5 ms" and a size of "260 B". There's also a "Save as Example" button.

The response body is displayed in JSON format:

```
1: {  
2:   "message": "Login User sucessful"  
3: }
```

Figure 4.7: Login route in Postman

To check the GET request, we can use a browser. So, in a browser go to <http://localhost:8000/api/users/current>, and we will get the correct JSON data.

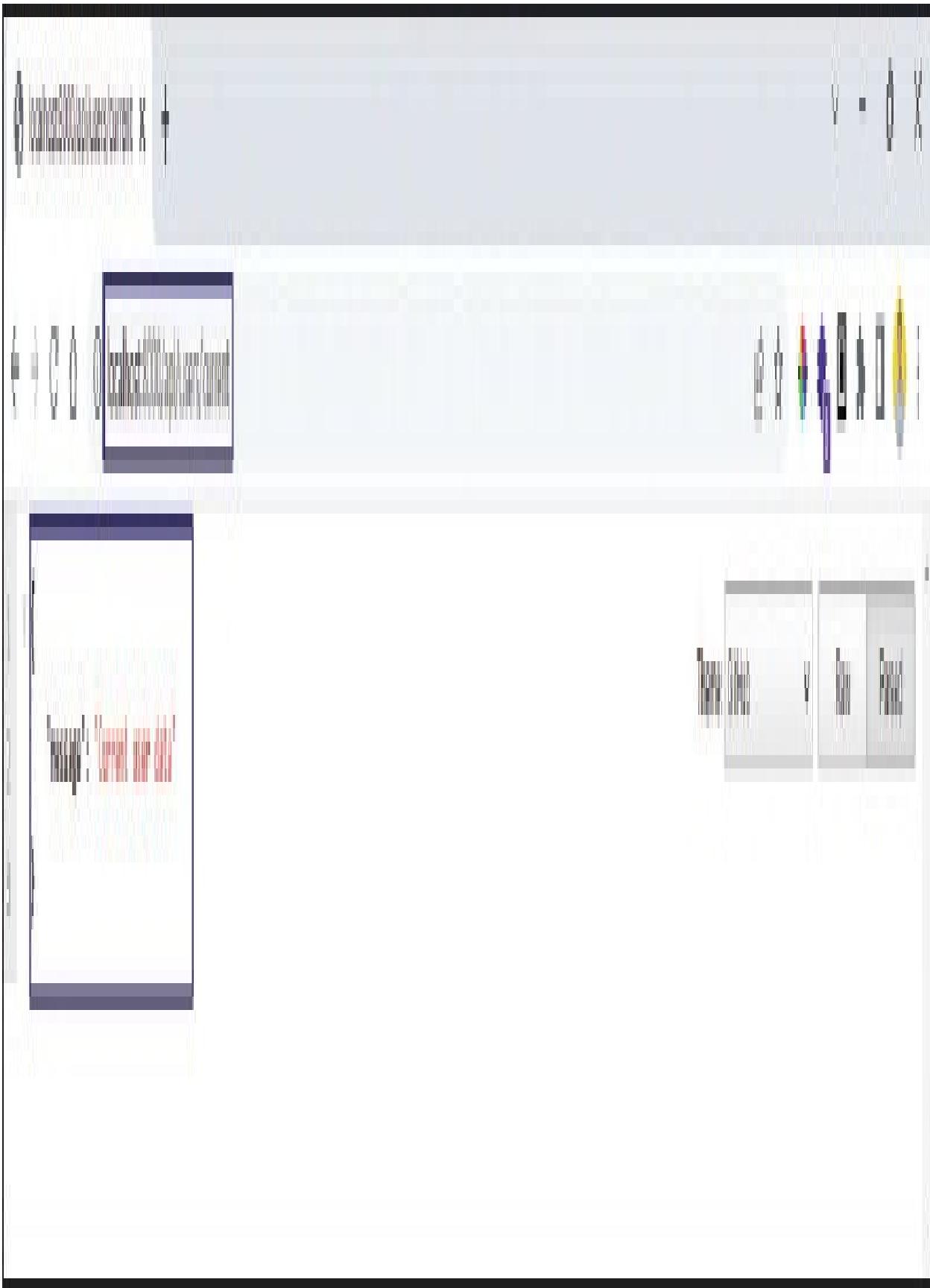


Figure 4.8: Get users in browser

Register User with Hashed Password

Before updating the register user function, we will install the package of bcryptjs and jsonwebtoken. These packages are used for password hashing and creating JWT respectively.

```
npm i bcryptjs jsonwebtoken
```

We are also importing jwt, bcrypt, and userModel in our userController.js file.

```
const jwt = require('jsonwebtoken')
const bcrypt = require('bcryptjs')
const User = require('../models/userModel')
```

A screenshot of the Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, userController.js, backend, and Visual Studio Code. The left sidebar shows files: server.js (selected), userController.js (highlighted), and openController.js. The main code editor area displays the following code:

```
1 const asyncHandler = require('express-async-handler')
2 const jwt = require('jsonwebtoken')
3 const bcrypt = require('bcryptjs')
4 const User = require('../models/userModel')

5
6 const registerUser = asyncHandler(async (req, res) => {
7     res.json({ message: 'Register User sucessful' })
8 }
9
10 const loginUser = asyncHandler(async (req, res) => {
```

The imports for jwt, bcrypt, and User are highlighted with a green rectangular selection.

The bottom terminal window shows the output of an npm audit command:

```
naben@haben: ~Desktop/Task-MERN/backend
$ npm i bcryptjs jsonwebtoken
added 11 packages, and audited 129 packages in 4s

12 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
```

The status bar at the bottom shows line 5, column 1, spaces 4, VIM-N, ORG, JS, Go Live, and a refresh icon.

Figure 4.9: Adding bcryptjs and jsonwebtoken

Now, in the userController.js file, we will update the registerUser(). Add the following code in the function:

```
const { name, email, password } = req.body

if (!name || !email || !password) {
  res.status(400)
  throw new Error('All fields are mandatory')
}

const userExists = await User.findOne({ email })

if (userExists) {
  res.status(400)
  throw new Error('User Exists')
}
```

Here, we are first destructuring the name, email, and password from the req.body. After that, we are checking if any field is missing and throwing an error, if that is the case.

We are using the `findOne()` on the `User` model to get the user data. Here, we are giving the email. Now, if the user already exists, we are throwing an error.

File Edit Selection View Go Run Terminal Help UserController.js · backend · Visual Studio Code

server.js UserController.js X

controllers > UserController.js > registerUser > asyncHandler() callback

```
4 const User = require('../models/userModel')
5
6 const registerUser = asyncHandler(async (req, res) => [
7   const { name, email, password } = req.body
8
9   if (!name || !email || !password) {
10     res.status(400)
11     throw new Error('All fields are mandatory')
12   }
13
14   const userExists = await User.findOne({ email })
15   if (userExists) {
16     res.status(400)
17     throw new Error('User Exists')
18   }
19
20   res.json({ message: 'Register User sucessful'})
21 ])
22
23 const loginUser = asyncHandler(async (req, res) => {
24   res.json({ message: 'Login User sucessful'})
25 })
26
27 const getCurrentUser = asyncHandler(async (req, res) => {
```

In 19, Col 1 Spaces: 4 UTF-8 CRLF {} JavaScript ⚡ Go live ⌂ ⌂

Figure 4.10: Updating Register User

In Postman, if we pass the name, email, and password fields, we will get back Status 200 and the correct JSON back.

Home Workspaces API Network Explore

Search Postman

Invite Upgrade

No Environment

MERN-Task / Register User

POST http://localhost:4000/api/users

Send

Params Authorization Headers Body Pre-request Script Tools Settings Cookies

Body

Key Value Description Edit

name	Narendu	
email	narendu@gmail.com	
password	narendu1234	

Key Value Description

Body Cookies Headers Test Results

Status 200 OK Time 19 ms Date 2023-08-10T10:45:23.000Z

Pretty Raw Preview Visualize JSON

```
1 {  
2   "message": "Register user successful"  
3 }
```

Copy Find and Replace

Cookies Capture request Headers Tools

Figure 4.11: Checking Register User in Postman

Now, we don't want to save a plain password in our database for the user. Because if someone hacks the database, they get all username and passwords. So, we are going to use bcrypt now in registerUser() in the userController.js file. Update the file with the following code:

```
const salt = await bcrypt.genSalt(10)
const hashedPassword = await bcrypt.hash(password, salt)

const user = await User.create({ name, email, password: hashedPassword })

if (user) {
  res.status(201).json({ _id: user.id, name: user.name, email: user.email })
} else {
  res.status(400)
  throw new Error('Invalid user data')
}
```

Here, we are first creating a random variable with the genSalt() method. After that, we are using the hash() method to generate a hash password. Here, we are passing the password and the salt value.

Now, we are using the mongoose create method on the User model to add the name, email, and hashedPassword of the user in the database.

If the user was saved successfully, we are sending back the id, name, and email. And if there is some error, we are throwing it back with an Invalid user data message.

File Edit Selection View Go Run Terminal Help userController.js-backend - Visual Studio Code

server.js userController.js

```
controllers > userController.js > registerUser
```

```
13
14     const userExists = await User.findOne({ email })
15     if (userExists) {
16         res.status(400)
17         throw new Error('User Exists')
18     }
19
20     const salt = await bcrypt.genSalt(10)
21     const hashedPassword = await bcrypt.hash(password, salt)
22
23     const user = await User.create({ name, email, password: hashedPassword })
24
25     if (user) {
26         res.status(201).json({ _id: user.id, name: user.name, email: user.email })
27     } else {
28         res.status(400)
29         throw new Error('Invalid user data')
30     }
31 }
32
33 const loginUser = asyncHandler(async (req, res) => {
34     res.json({ message: 'Login User sucessful' })
35 })
```

Ln 31, Col 3 Spaces:4 UTF-8: CRLF {} JavaScript Go Live ⚡

Figure 4.12: Adding hashed password

Back in Postman if we give all data again and hit the Send button, we will get a Status 201. And the correct JSON will also be returned back.

Home Workspaces API Network Explore

Search Postman

Invite Upgrade X

Register User Login User + No Environment

MERN-Tech / RegisterUser

POST http://localhost:3001/api/users

Send

Params Authorization Headers Body Pre-request Script Tests Settings Create

History

Key Value Description Bulk Edit

name Nahendu

email nahendu@gmail.com

password nahendu12345

Key Value Description

Body Cookies Headers Test Results

Status: 201 Created Time: 234 ms Size: 19 B Save as Example

Pretty Raw Preview Viewsize JSON

```
1 {
2   "id": "4474079ef1997e07c1f471",
3   "name": "Nahendu",
4   "email": "nahendu@gmail.com"
5 }
```

Figure 4.13: Registering user with Postman

We can also confirm the same in the MongoDB cloud database. Here, in the users the new user with the hashed password will be saved.

Cloud Manager

https://cloud.mongodb.com/v2/5477e3d31046d1302a525520#metrics/replSet/6477e4c5a42a4367ef23a9/e

Atlas MongoDB AccessManager Billing All Clusters Get Help Narendra

MERN-Task-App Data Services Charts

DEPLOYMENT DATABASES | COLLECTIONS

+ Create Database

taskApp.users

SEARCH NAMESPACEES

taskApp

task

users

Storage Size: 20KB Logical Data Size: 17B Total Documents: 1 Indexes Total Size: 40KB

Find Indexes Schema Anti-Patterns Aggregation Search Indexes

Filter type a query: { field: 'value' } Reset Apply More Options

QUERY RESULTS: 1 document

```
_id: ObjectId('4478478af15bc7ed7cbf5671')
name: 'Narendra'
email: 'narendra@gmail.com'
password: '$2y$10$3MhJHf2JNhTqfhrASzvAnqz0gf68lyrkaOJRCJ3UmTaFH3ck'
createdAt: 2013-06-01T11:54:50.148+00:00
updatedAt: 2013-06-01T11:54:50.148+00:00
__v: 0
```

Figure 4.14: Saved user in database

Login User

Now, we will update the Login user functionality. So, in the loginUser function of the userController.js file, add the following code:

```
const { email, password } = req.body  
  
const user = await User.findOne({ email })  
  
if (user && (await bcrypt.compare(password, user.password))) {  
  res.json({ _id: user.id, name: user.name, email: user.email })  
}  
else {  
  res.status(400)  
  throw new Error('Invalid data')  
}
```

Here, we are first destructuring the email and password from the req.body. We are using the findOne() on the User model to get the user data. Here, we are giving the email.

Now, if the user exists and password matched, we are sending back the id, name, and email. Notice that since the password is hashed in the database, we are using bcrypt.compare() to compare the passwords.

If the user doesn't exist or the password doesn't match, we are throwing an error with 'Invalid data' message.

File Edit Selection View Go Run Terminal Help userController.js - Visual Studio Code

JS server.js JS userController.js X

controllers > JS userController.js > [o] loginUser > ↗ asyncHandler() callback

```
27     } else {
28         res.status(400)
29         throw new Error('Invalid user data')
30     }
31 }
32
33 const loginUser = asyncHandler(async (req, res) => {
34     const { email, password } = req.body
35     const user = await User.findOne({ email })
36
37     if (user && (await bcrypt.compare(password, user.password))) {
38         res.json({ _id: user.id, name: user.name, email: user.email })
39     } else {
40         res.status(400)
41         throw new Error('Invalid data')
42     }
43 }
44
45 const getCurrentUser = asyncHandler(async (req, res) => {
46     res.json({ message: 'Current user data' })
47 }
48
49 module.exports = { registerUser, loginUser, getCurrentUser }
```

Ln 38, Col 67 Spaces: 4 UTF-8 CRLF () JavaScript ⚡ Go live ⚡ ⚡

Figure 4.15: Updating Login User

Back in Postman, a POST request to `http://localhost:8000/api/users/login` with no data, will give us an error JSON back.

☰ ← → Home Workspaces API Network Explore

Search Postman

Invite Upgrade - X

Register User Login User + ⚙

No Environment

MERN-Task / Login User

Save ↻ 🔍 🗑

POST http://localhost:8000/api/users/login

Send ↻ 📁

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies ↻

History none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description	...	Bulk Edit
Key	Value	Description

Body Cookies Headers (7) Test Results

Status: 400 Bad Request Time: 56 ms Size: 270 B Save as Example

Pretty Raw Preview Visualize JSON ↻

```
1 {  
2   "message": "Invalid data"  
3 }
```

Figure 4.16: Login User with no data

Even a POST request to `http://localhost:8000/api/users/login` with the wrong password, will give us an error JSON back.

Home Workspaces API Network Explore

Search Postman

Invite 🔍 🔔 🔕 Upgrade

HTTP Register User Login User No Environment

MERN-Task / Login User

Save 📁 🚮 🗑️

POST http://localhost:3000/api/users/login Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

History none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description	Bulk Edit
email	nabendu@gmail.com		
password	12345		

Body Cookies Headers (7) Test Results

Status: 400 Bad Request Time: 292 ms Size: 270 B Save as Example

Pretty Raw Preview Visualize JSON

```
1: { "message": "Invalid Data" }
```

Figure 4.17: Login User with wrong password

Only a POST request to `http://localhost:8000/api/users/login` with all correct data, will give us Status 200 and id, name and email back.

Home Workspaces API Network Explore

Search Postman

Invite Upgrade No Environment

Register User Login User

MERN-Task / Login User

POST http://localhost:8000/api/users/login

Send

Params Authorization Headers Body Pre-request Script Tests Settings Cookies

Body

none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description	Bulk Edit
<input checked="" type="checkbox"/> email	nebendu@gmail.com		
<input checked="" type="checkbox"/> password	nebendu12345		

Key Value Description Bulk Edit

Body Cookies Headers [7] Test Results

Pretty Raw Preview Visualize JSON

Status: 200 OK Time: 124 ms Size: 314 B Save as Example

```
{  
  "id": "647887041f59c2ed7ca16621",  
  "name": "Nebendu",  
  "email": "nebendu@gmail.com"  
}
```

Figure 4.18: Login User successfully

Understanding JWT

JWT(JSON Web Token) is often used to add authentication in web apps. The official site for JWT is <https://jwt.io/> and as per it, the JWT token consists of three parts. The first part is the Header, the second is the Payload, and the third is the Verify Signature.

Out of these, the second is the most important, because it contains the details of the payload. But the third one also contains the signature without which the token cannot be verified.

The screenshot shows the jwt.io website interface. At the top, there's a navigation bar with links for Debugger, Libraries, Introduction, Ask, and a Crafted by auth0 logo. Below the header, there are two main sections: "Encoded" on the left and "Decoded" on the right.

Encoded PASTE A TOKEN HERE

`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c`

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239822  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
) □ secret base64 encoded
```

Figure 4.19: The JWT Token

Using JWT

To use JWT, we will first add JWT_SECRET in the .env file. The secret can be anything and we have kept it thewebdev.

A screenshot of the Visual Studio Code interface. The top bar shows the menu: File, Edit, Selection, View, Go, Run, Terminal, Help, and the status bar indicates "ctrl + F5 - Reload - Visual Studio Code". Below the menu is a toolbar with icons for file operations. The main workspace shows several tabs: "server.js", "UserController.js", "env", and "X". The "env" tab is active and displays the following environment variables:

```
1 MODE_ENV = development
2 PORT = 8080
3 MONGO_URI = mongodb+srv://admin:admin@narendracluster.amp07ug.mongodb.net/taskApp
4 JWT_SECRET = thekeybey
```

Figure 4.20: The JWT Secret

Then, in the userController.js file, we will add the following function:

```
const generateJWTtoken = id => jwt.sign({ id }, process.env.JWT_SECRET, {  
  expiresIn: '5d' })
```

Now, the function generateJWTtoken is taking an id. And it is using the jwt.sign() method. Here, it is taking the id and secret from the environment file as the first two parameters.

In the third parameter, we are giving the expiresIn parameter. Here, we can specify after how much time our JWT will expire, and we need to generate a new one. We have kept it for five days.

A screenshot of the Visual Studio Code interface. The top bar shows the menu: File, Edit, Selection, View, Go, Run, Terminal, Help, and the workspace name: userController.js - backend. The title bar also displays "userController.js" and "Visual Studio Code". The main editor area shows the file "userController.js" with the following code:

```
44
45 const getCurrentUser = asyncHandler(async (req, res) => {
46   res.json({ message: 'Current user data' })
47 }
48
49 const generateJWTtoken = id => jwt.sign({ id }, process.env.JWT_SECRET, { expiresIn: '5d' })
50
51 module.exports = { registerUser, loginUser, getCurrentUser }
```

Figure 4.21: The generate token method

Then, in the registerUser and loginUser function in the userController.js file, we are also passing back the token when the request is successful. The token is generated by passing the id in the generateToken() function.

File Edit Selection View Go Run Terminal Help userController.js - backend - Visual Studio Code

server.js userController.js

```
controllers > userController.js > loginUser > asyncHandler() callback > token
```

```
13
14     const userExists = await User.findOne({ email })
15     if (userExists) {
16         res.status(400)
17         throw new Error('User Exists')
18     }
19
20     const salt = await bcrypt.genSalt(10)
21     const hashedPassword = await bcrypt.hash(password, salt)
22
23     const user = await User.create({ name, email, password: hashedPassword })
24
25     if (user) {
26         res.status(201).json([ { _id: user.id, name: user.name, email: user.email, token: generateJWTtoken(user._id) } ])
27     } else {
28         res.status(400)
29         throw new Error('Invalid user data')
30     }
31 })
32
33 const loginUser = asyncHandler(async (req, res) => {
34     const { email, password } = req.body
35     const user = await User.findOne({ email })
36
37     if (user && (await bcrypt.compare(password, user.password))) {
38         res.json({ _id: user.id, name: user.name, email: user.email, token: generateJWTtoken(user._id) })
39     } else {
40         res.status(400)
41         throw new Error('Invalid data')
```

Figure 4.22: Using generate token method

Now, in Postman the login route `http://localhost:8000/api/users/login` with all correct data, will give us Status 200. It will also return back id, name, email, and JWT token.

The screenshot shows the Postman application interface. At the top, there are navigation links for Home, Workspaces, API Network, and Explore. A search bar says "Search Postman". On the right, there are icons for Invite, Upgrade, and Help. The main header shows "No Environment" and a "+" button. Below the header, there are two items: "POST Register User" and "POST Login User". The "POST Login User" item is selected. To its right are "Save", "Cancel", and "Send" buttons. The left sidebar has sections for Collections (MERN-Task / Login User), Environments, and History. The main content area shows a POST request to "http://localhost:8000/api/users/login". The "Body" tab is selected, showing the following parameters:

Key	Value	Description
email	nabendu@gmail.com	
password	nabendu12345	

Below the body, there are tabs for Body, Cookies, Headers, and Test Results. The "Test Results" tab is selected, showing a status of 200 OK with a response time of 100 ms and a size of 490 B. The response body is displayed as JSON:

```
1 {  
2   "id": "4678670a1f39c7e07cb4471",  
3   "name": "Nabendu",  
4   "email": "nabendu@gmail.com",  
5   "token": "eyJhbGciOiJIUzI1NiJ9.RBcc16JkpxVC39.eyJpczI6IjY0Nzg4NzBhZmY1MDM3ZWQ3Y2JmRTY3MSIsInIiOjE1MTY4NTYyNTY2MzIzZXhwIjoxNjg2NDUzNjMyfQ.  
av1LSEIdz8WUtg1-6V5J1nZ2P_-7gyr4a511cvQ0icn"  
6 }
```

Figure 4.23: Login route getting JWT Token

Similarly, the register route `http://localhost:8000/api/users` with all correct data, will give us Status 200. It will also return back id, name, email, and JWT token.

Home Workspaces API Network Explore

Search Postman

Invite Upgrade

No Environment

MERN-Task / Register User

POST <http://localhost:3000/api/users>

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

History none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description	Bulk Edit
<input checked="" type="checkbox"/> name	Shikha		
<input checked="" type="checkbox"/> email	shikha@gmail.com		
<input checked="" type="checkbox"/> password	shikha12345		

Key	Value	Description
-----	-------	-------------

Status: 201 Created Time: 176 ms Size: 501 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": "64700bc635f4ace641e66815",
3   "name": "Shikha",
4   "email": "shikha@gmail.com",
5   "token": "eyJhbGciOiJIUzI1NiInRicCI6IkpxYVJ9.eyJpZCI6Ijy0Nzg4YmM2NzYwNGF5YTtYONGE2IjAxNzIzInIhcG8tVTV4NTYyMTcmtH1wIzXnITjowNjg2NDUzNzAyIQ.
6   58jxrlavgq99qTEFAygf5RqQ0o7a3LahexL_dn0ta-o"
```

Figure 4.24: Register route getting JWT Token

But, if we try to Register with the same details, we will get back the error of User Exists in the Register route:

Home Workspaces API Network Explore

Search Postman

Invite 🚧 Upgrade - X

No Environment

POST Register User POST Login User +

MERN-Task / Register User

Save 🛡️ 🗑️ 📁

POST http://localhost:3000/api/users Send

Params Authorization Headers (3) Body Pre-request Script Tests Settings Cookies

History none form-data x-www-form-urlencoded raw binary GraphQL

Key	Value	Description	... Bulk Edit
name	Shikha		
email	shikha@gmail.com		
password	shikha12345		

Key	Value	Description

Body Cookies Headers (7) Test Results

Status: 400 Bad Request Time: 32 ms Size: 269 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {  
2   "message": "User Exists"  
3 }
```

Figure 4.25: Register route getting error

Conclusion

In this chapter, we have learned about JWT tokens and hashing passwords. Here, we have first created a user model and controller. Next, we have created the register user functionality. Here, while saving the password, we used the bcrypt package to hash it.

Next, we created the login functionality. Here, to compare the passed password to the saved password, we had to use bcrypt again. Then, we learned about JWT tokens and implemented them in our project.

In the next chapter, we will use JWT to protect routes.

Points to Remember

We have created the user model with name, email, and password as String. We have also added basic validation in it.

We have used the library of bcryptjs to hash the password, before saving it to the database. JWT(JSON Web Token) is used to authenticate web-apps. We are using the library of jsonwebtoken in our project.

We have generated a JWT token using the method of jwt.sign() in which we can also give the days after which this token will expire.

CHAPTER 5

Auth Middleware and Protecting Routes

Introduction

This chapter talks about Auth middleware and Protecting routes. Here, we will first create an Auth middleware using the JWT created in the last chapter. After that, we will protect some of the routes using this Auth middleware, which require the JWT token to be verified.

Structure

In this chapter, we are going to discuss the following topics:

Creating Auth middleware

Understanding Protect Routes

Protecting Task Routes

Creating Auth middleware

We will create a file authMiddleware.js in the middleware folder and add the following content in authMiddleware.js file:

```
const jwt = require('jsonwebtoken')
const asyncHandler = require('express-async-handler')
const User = require('../models/userModel')

const protect = asyncHandler(async (req, res, next) => {
  let token;

  if (req.headers.authorization && req.headers.authorization.startsWith('Bearer'))
  {
    try {
      token = req.headers.authorization.split(' ')[1]
      const decoded = jwt.verify(token, process.env.JWT_SECRET)
      req.user = await User.findById(decoded.id).select('-password')
      next()
    } catch (error) {
      console.log(error)
    }
  }
})
```

```
res.status(401)

throw new Error('You are not authorized')

}

}

})

module.exports = { protect }
```

Here, we are importing the required imports of jwt, asyncHandler, and User model. After that, we wrote a function called protect. Inside the function, we are checking if the authorization header is present with an if statement.

Inside the if statement, we have a try..catch block. In the try, we are first splitting the token with the split command because it will be like “Bearer XXXXYYYYMM”. Next, we are using the verify method from jwt and passing the token and Secret. After that, we are getting the user details back with findById(). Notice that we are omitting the password here. After that, the required next() of the middleware is there.

In the catch part, we are returning a 401 and also throwing an error You are not authorized.

File Edit Selection View Go Run Terminal Help authMiddleware.js - backend - Visual Studio Code

EXPLORER ... JS server.js JS userController.js JS authMiddleware.js X

BACKEND middleware > JS authMiddleware.js > protect > ↗ asyncHandler() callback

> connect
controllers
JS taskController.js
JS userController.js
middleware
JS authMiddleware.js
JS errorMiddleware.js
models
JS taskModel.js
JS userModel.js
> node modules
routes
JS taskRoutes.js
JS userRoutes.js
env
gitignore
{} package-lock.json
{} package.json
JS server.js

```
1 const jwt = require('jsonwebtoken')
2 const asyncHandler = require('express-async-handler')
3 const User = require('../models/userModel')
4
5 const protect = asyncHandler(async (req, res, next) => {
6   let token;
7
8   if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
9     try {
10       token = req.headers.authorization.split(' ')[1]
11       const decoded = jwt.verify(token, process.env.JWT_SECRET)
12       req.user = await User.findById(decoded.id).select('-password')
13       next()
14     } catch (error) {
15       console.log(error)
16       res.status(401)
17       throw new Error('You are not authorized')
18     }
19   }
20 })
21
22 module.exports = { protect }
23
```

OUTLINE TIMELINE

ln 19, Col 4 Spaces: 4 UTF-8 CRLF () JavaScript Go Live

Figure 5.1: Creating authMiddleware.js file

Next, we will also add the following code in authMiddleware.js file:

```
if (!token) {  
  res.status(401)  
  throw new Error('Not authorized, no token')  
}
```

Here, we are checking if the token is not available. Then we are sending a 401 and also an error Not authorized, no token.

A screenshot of the Visual Studio Code interface showing a file named `authMiddleware.js`. The code implements a middleware function named `protect` that checks for a JWT token in the authorization header and verifies it using a secret key. If successful, it retrieves the user from the database and continues the request. If there is no token or it fails verification, it returns a 401 status and an error message.

```
File Edit Selection View Go Run Terminal Help authMiddleware.js backend - Visual Studio Code

JS server.js JS UserController.js JS authMiddleware.js X

middleware > JS authMiddleware.js > (1) protect > (1) asyncHandler() callback

try {
  token = req.headers.authorization.split(' ')[1]
  const decoded = jwt.verify(token, process.env.JWT_SECRET)
  req.user = await User.findById(decoded.id).select('-password')
  next()
} catch (error) {
  console.log(error)
  res.status(401)
  throw new Error('You are not authorized')
}

if (!token) {
  res.status(401)
  throw new Error('Not authorized, no token')
}

module.exports = { protect }
```

Figure 5.2: Updating authMiddleware.js file

Understanding Protect Routes

To protect a route, we need to add the protect function in the route. So, we will go to the userRoutes.js file and import the protect function. After that, in the get route for /current, we will add the protect function.

```
const { protect } = require('../middleware/authMiddleware')
```

```
File Edit Selection View Go Run Terminal Help userRoutes.js - backend - Visual Studio Code

src | userController.js | authMiddleware.js | userRoutes.js X

1 const express = require('express')
2 const router = express.Router()
3 const { registerUser, loginUser, getCurrentUser } = require('../controllers/userController')
4 const { protect } = require('../middleware/authMiddleware')
5
6 router.post('/', registerUser)
7 router.post('/login', loginUser)
8 router.get('/current', protect, getCurrentUser)
9
10 module.exports = router
11
```

Figure 5.3: Adding protect in userRoutes.js

Now, in Postman go to `http://localhost:8000/api/users/current` and we will get the Not authorized, no token message. This message is from the `authMiddleware.js` file because we didn't provide a token that was required.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar is present, along with a 'Logout' button and an 'Upgrade' dropdown.

In the main workspace, there is a list of collections: 'Protected Route' is selected. Below it, a specific request is displayed:

- Method:** GET
- URL:** http://localhost:8000/api/users/current
- Send** button (blue)

The 'Params' tab is active, showing two parameters:

Key	Value	Description	Bulk Edit
key	value	Description	(empty)

At the bottom, the response section shows:

- Status: 401 Unauthorized
- Time: 14 ms
- Size: 283 B
- Save as Example

The response body is displayed in JSON format:

```
1: { "message": "Not authorized, no token"}
```

Figure 5.4: Current user route with error

Now, in Postman to add a token, we will first go to the Authorization tab. Here, in the Type select Bearer Token. And after that, in the Token field add the token, which we have got from the Login User POST request in the previous chapter.

After that click the Send button and we will get a status 200 with successful message of Current user data.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar says "Search Postman". On the right, there are buttons for "Invite", "Upgrade", and a close button. Below the header, there's a toolbar with icons for "Register User", "Login User", "Protected Route", and others. A dropdown menu says "No Environment".

The main area shows a collection named "MERN-Task / Protected Route". On the left, there are sections for "Collections", "Environments", and "History".

A specific request is selected:

- Method: GET
- URL: <http://localhost:8000/api/users/current>
- Authorization tab is selected.
- Type: Bearer Token
- Token: eyJhbGciOiJIUzI1NiJ9.R5cO8kpXVCJ9.eyJy...

The "Body" tab is visible at the bottom, showing JSON response data:

```
1: {  
2:   "message": "Current user data"  
3: }
```

At the bottom right, status information is shown: Status: 200 OK, Time: 36 ms, Size: 266 B, Save as Example, and other icons.

Figure 5.5: Current user route with token

Now, in the userController.js file and inside the getCurrentUser() add the following code:

```
const { _id, name, email } = await User.findById(req.user.id)  
res.status(200).json({ id: _id, name, email })
```

Here, we are using the findById() on the User and then getting back the id, name, and email. We are sending this back as a success.

```
File Edit Selection View Go Run Terminal Help userController.js (backend) Visual Studio Code

C:\Users\user\Documents\GitHub\Node.js-Auth\backend>cd backend
C:\Users\user\Documents\GitHub\Node.js-Auth\backend>code .\src\controller\userController.js

controllers > userController.js

45 const getCurrentUser = asyncHandler(async (req, res) => {
46   const { _id, name, email } = await User.findById(req.user.id)
47   res.status(200).json({ id: _id, name, email })
48 }
49
50 const generateJWTToken = id => jwt.sign({ id }, process.env.JWT_SECRET, { expiresIn: '5d' })
51
52 module.exports = [ registerUser, loginUser, getCurrentUser ]
```

Figure 5.6: Updating getCurrentUser

Again, in Postman go to `http://localhost:8000/api/users/current`, and we get the current user data back.

Home Workspaces API Network Explore Search Postman [A Help](#) [\\$](#) [U](#) [Upgrade](#) [-](#) [X](#)

Register User Login API Protected Route No Environment [+](#) [-](#)

HTTP Test / Protected Route Save [/](#) [-](#) [X](#)

Method: GET URL: http://localhost:8080/api/user/current Send

Headers: Authorization: Bearer Token

Type: Bearer Token (Note: These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about [variables](#))

The authorization header will be automatically generated when you send the request. Learn more about [authorization](#).

Token: eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vYmVzdGFuZC5jb20iLCJhdWQiOiJodHRwOi8vYmVzdGFuZC5jb20iLCJleHAiOjE2MjQyNjUxNzAsImF1ZCI6ImRvZ2VuZC11c2VycyIsInN1YiI6IjIwMjAxMDIwMDAwMDAwMCIsInNjb3BlcyI6WyJhcGlfcmVhZCJdfQ.

Body Cookies Headers Test Results

Preview Raw Preview Variables JSON [-](#)

```
1 {  
2   "id": "5a78770d19e29e07e0f1871",  
3   "name": "Robert",  
4   "email": "robert@ppp1.com"  
5 }
```

6 0

Figure 5.7: Getting User data back

Protecting Task Routes

Now, we will protect all of our task routes that we created earlier. Here, again we will follow the same process as in the earlier section.

GET and POST Route

We will first protect the GET and POST route in Task. First, in the taskRoutes.js file, we will protect the GET route. Here, again we are importing the protect function and adding it to the get route.

```
const { protect } = require('../middleware/authMiddleware')
```

A screenshot of the Visual Studio Code interface showing the file `taskRoutes.js`. The code defines an Express router for tasks, including routes for getting, setting, updating, and deleting tasks. It also includes a middleware protection step. The code is as follows:

```
File Edit Selection View Go Run Terminal Help taskRoutes.js backend - Visual Studio Code

server.js taskRoutes.js ✘ userRoutes.js taskController.js

routes > taskRoutes.js ...

1 const express = require('express');
2 const router = express.Router();
3 const { getTasks, setTask, updateTask, deleteTask } = require('../controllers/taskController');
4
5 const { protect } = require('../middleware/authMiddleware')
6
7 router.get('/', protect, getTasks)
8 router.post('/', setTask)
9 router.put('/:id', updateTask)
10 router.delete('/:id', deleteTask)
11
12 module.exports = router;
```

Figure 5.8: Protecting GET route for task

Now, in Postman we will go to the GET route `http://localhost:8000/api/tasks`, and hit the Send button. We will get an error since we have not given any token.

The screenshot shows the Postman application interface. At the top, there are navigation links for Home, Workspaces, API Network, and Explore. A search bar is followed by a toolbar with icons for Help, Import, Export, and Upgrade. The main workspace shows a collection named "MERN-Task / Protected Task Route". A specific request is selected, which is a GET request to the URL `http://localhost:3001/api/tasks`. The "Authorization" tab is active, showing the value `undefined`. Other tabs include Headers, Body, Pre-request Script, Tests, and Settings. Below the request details, the "Body" tab is selected, displaying a JSON response object with a single key-value pair: `"error": "no authorization, no token"`.

Figure 5.9: GET route for task error in Postman

Now, we will also add the protect function to all the routes in taskRoutes.js file.

taskRoutes.js

```
const express = require('express');
const router = express.Router();
const { getTasks, setTask, updateTask, deleteTask } = require('../controllers/taskController');

const [ protect ] = require('../middleware/authMiddleware')

router.get('/', protect, getTasks)
router.post('/', protect, setTask)
router.put('/id', protect, updateTask)
router.delete('/id', protect, deleteTask)

module.exports = router;
```

Figure 5.10: Protecting all Task routes

We will modify the getTask and setTask functions in taskController.js file. Here, we are finding by the user's id and also saving the user's id in through the POST request.

File Edit Selection View Go Run ... taskController.js - backend - Visual Studio Code

controllers > taskController.js ..

```
1 const asyncHandler = require('express-async-handler')
2 const Task = require('../models/taskModel')
3
4 const getTasks = asyncHandler(async (req, res) => {
5     const tasks = await Task.find({ user: req.user.id })
6     res.status(200).json(tasks);
7 })
8
9 const setTask = asyncHandler(async (req, res) => {
10     if(!req.body.text){
11         res.status(400)
12         throw new Error('Please enter a task')
13     }
14
15     const task = await Task.create({ text: req.body.text, user: req.user.id })
16     res.status(200).json(task);
17 })
18
```

Figure 5.11: Updating getTasks function

Back, in Postman we will go to the POST route `http://localhost:8000/api/tasks`, add a task and hit the Send button. We will get an error since we have not given any token.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar is followed by an 'Invite' button and other account-related icons. The main workspace shows a collection named 'MERN-Task / Post Task'. Below it, a POST request is defined with the URL `http://localhost:3000/api/tasks`. The 'Body' tab is selected, showing a key-value pair where the key is 'text' and the value is 'Learn Blockchain'. Other tabs include Params, Authorization, Headers, Pre-request Script, Tests, Settings, Cookies, and a Bulk Edit section. At the bottom, the response status is shown as 401 Unauthorized with a message: 'message': 'Not authorized, no token'.

POST `http://localhost:3000/api/tasks`

Body

Key	Value	Description	Bulk Edit
<input checked="" type="checkbox"/> text	Learn Blockchain		<input type="button" value="..."/>

Status: 401 Unauthorized Time: 25 ms Size: 263 B Save as Example

Body

```
1: "message": "Not authorized, no token"
```

Figure 5.12: POSTMAN with POST route without token

Now in the Authorization tab, we will add a Bearer token and we will be able to successfully add the task.

The screenshot shows the Postman application interface. At the top, there are navigation links for Home, Workspaces, API Network, and Explore, along with a search bar and user account options. Below the header, a collection named "MERN-Task / Post Task" is selected. A specific POST request is highlighted, targeting the URL `http://localhost:8000/api/tasks`. The "Authorization" tab is active, showing a dropdown set to "Bearer Token". A note below the dropdown states: "These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables." To the right of the token input field, there is a status indicator showing "Status: 200 OK Time: 96 ms Size: 416 B". The "Body" tab is also visible at the bottom.

POST `http://localhost:8000/api/tasks`

Authorization

Type: Bearer Token

Token: eyJhbGciOiJIUzI1NiJ9.R5cC18kpXVCJ9.eyJ...

Body

```
1: {text: "Learn Blockchain", 2: user: "6478870aff59c7e07chf5671", 3: _id: "64798c6d389011439d8b460a", 4: createdAt: "2023-06-02T06:29:56.974Z", 5: updatedAt: "2023-06-02T06:29:56.974Z", 6: __v: 0}
```

Figure 5.13: POSTMAN with POST route with token

We have added one more task for the same user. Now, in Postman, go to the GET route `http://localhost:8000/api/tasks`, and add the token of the user in the Authorization tab. Now, on clicking the Send button, we will get the tasks of this user.

Home Workspaces API Network Explore

Search Postman Invite Upgrade

Register User Login User Post Task Protected Tasks Route

MERN-Task / Protected Tasks Route

GET http://localhost:8000/api/tasks

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about [authorization](#).

Token eyJhbGciOiJUzI1NiIsInR5cCI6IkpXVCJ9.eyJy...

Status: 200 OK Time: 80 ms Size: 587 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": "64798c643890114390834601",
3   "text": "Learn blockchain",
4   "user": "64798878aff59c7e07cbf5d71",
5   "createdAt": "2023-06-02T06:29:56.978Z",
6   "updatedAt": "2023-06-02T06:29:56.978Z",
7   "__v": 0
8 },
9 {
10   "id": "64798c9f3890114390834600",
11   "text": "Learn ChatGPT",
12   "user": "64798878aff59c7e07cbf5d71",
13   "createdAt": "2023-06-02T06:30:55.772Z",
14   "updatedAt": "2023-06-02T06:30:55.772Z",
15   "__v": 0
16 }
17 ]
18 }
```

Online Find and Replace Console Cookies Capture requests Runner Trash

Figure 5.14: POSTMAN with GET route with token

We have added two more tasks for a different user through the POST route. Now in the GET, when we use this user's token, we will get back this user's tasks.

Home Workspaces API Network Explore

Search Postman Invite Upgrade

Protected Tasks Route

MERN-Task / Protected Tasks Route

GET http://localhost:8000/api/tasks

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about [authorization](#).

Token eyJhbGciOiJuZfNishnR5cOISlkpxVCJ9eyJpZC8jYONGdymM2MzVnNOFYTYONGE2N

Status: 200 OK Time: 82 ms Size: 595 B Save as Example

Pretty Row Preview Visualize JSON

```
1 {
2   "id": "64790d42309011439d0b4615",
3   "text": "Learn IELTS",
4   "userId": "647838c635f3ac6dd66015",
5   "createdAt": "2023-06-02T06:33:58.583Z",
6   "updatedAt": "2023-06-02T06:33:58.583Z",
7   "__v": 0
8 },
9 {
10   "id": "64790d5a309011439d0b4616",
11   "text": "Learn US English",
12   "userId": "647838c635f3ac6dd66015",
13   "createdAt": "2023-06-02T06:34:02.629Z",
14   "updatedAt": "2023-06-02T06:34:02.629Z",
15   "__v": 0
16 }
17 ]
18 }
```

Online Find and Replace Console Cookies Capture requests Runner Trash

Figure 5.15: Added more tasks for Users

PUT and DELETE Route

First, we are going to update the PUT route. This is done in the updateTask function in the taskController.js file. Here, we are first finding the user with findById() method, by passing the user id.

Next, if we do not find any user, we are throwing an error. We are also throwing an error if a wrong token is used to update the task.

```
const user = await User.findById(req.user.id)
```

```
if(!user){  
  res.status(401)  
  throw new Error('No such user found')  
}
```

```
if (task.user.toString() !== user.id) {  
  res.status(401)  
  throw new Error('User is not authorized to update')  
}
```

File Edit Selection View Go Run Terminal Help taskController.js - Visual Studio Code

server.js taskRoutes.js taskController.js X

controllers > taskController.js > ...

```
19 const User = require('../models/userModel')
20
21 const updateTask = asyncHandler(async (req, res) => {
22     const task = await Task.findById(req.params.id)
23
24     if (!task) {
25         res.status(400)
26         throw new Error('Task not found')
27     }
28
29     const user = await User.findById(req.user.id)
30
31     if (!user) {
32         res.status(401)
33         throw new Error('No such user found')
34     }
35
36     if (task.user.toString() !== user.id) {
37         res.status(401)
38         throw new Error('User is not authorized to update')
39     }
40
41     const updatedTask = await Task.findByIdAndUpdate(req.params.id, req.body, { new: true })
```

Ln 20, Col 1 Spaces: 4 UTF-8 CR/LF () JavaScript Go live ⚡

Figure 5.16: Updating updateTask function

Now we are going to update the DELETE route. This is done in the deleteTask function in the taskController.js file. Here, we are again using the same logic as the updateTask function.

```
const user = await User.findById(req.user.id)

if(!user){
  res.status(401)
  throw new Error('No such user found')
}

if (task.user.toString() !== user.id) {
  res.status(401)
  throw new Error('User is not authorized to update')
}
```

File Edit Selection View Go Run Terminal Help taskController.js · backend · Visual Studio Code

JS server.js JS taskRoutes.js JS taskController.js X

controllers > JS taskController.js > ...

```
44 const deleteTask = asyncHandler(async (req, res) => {
45     const task = await Task.findById(req.params.id)
46
47     if (!task) {
48         res.status(400)
49         throw new Error('Task not found')
50     }
51
52     const user = await User.findById(req.user.id)
53
54     if(!user){
55         res.status(401)
56         throw new Error('No such user found')
57     }
58
59     if (task.user.toString() !== user.id) {
60         res.status(401)
61         throw new Error('User is not authorized to delete')
62     }
63
64     await Task.findByIdAndUpdateAndDelete(req.params.id)
65
66     res.status(200).json({ id: req.params.id })
67 })
```

Ln 4, Col 1 Spaces: 4 UTF-8 CRLF {} JavaScript Go Live ⚡ 🌐

Figure 5.17: Updating deleteTask function

Now, in Postman, we will put a DELETE request with a url like <http://localhost:8000/api/tasks/64798d42389011439d8b4613>. Here, take any task id of a user. Now, when we hit the Send button, we will get an error, since we have not provided any token.

The screenshot shows the Postman interface for testing an API endpoint. The top navigation bar includes Home, Workspaces, API Network, and Explore. A search bar for 'Search Postman' is present, along with user icons for Invite, Upgrade, and Logout.

The main workspace displays a collection named 'MERN-Task / Delete Task'. It lists three items: a POST request for 'Login User', a GET request for 'Protected Tasks Route', and a DELETE request for 'Delete Task'. The DELETE request is highlighted with a blue border.

The 'DELETE' request details are shown in the center:

- Method: DELETE
- URL: <http://localhost:8000/api/tasks/6478ed41389011430d8b4613>
- Send button is visible

The 'Params' tab is selected in the request settings. The 'Query Params' section contains two rows:

Key	Value	Description	...	Bulk Edit
Key	Value	Description	...	⋮

The 'Body' tab is active at the bottom, showing a JSON response:

```
1 {  
2   "message": "Not authorized, no token"  
3 }
```

The status bar at the bottom right indicates: Status: 401 Unauthorized, Time: 11 ms, Size: 263 B, Save as Example, and a copy icon.

Figure 5.18: POSTMAN with DELETE route without token

Now, we will add a token in the Authorization tab. But this token will be of a different user. Now, when we hit Send, we will get the User is not authorized to delete error.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. On the right side of the header are buttons for Invite, Upgrade, and a close button. Below the header, there's a search bar labeled "Search Postman".

The main workspace displays a collection named "MERN-Task / Delete Task". It contains a single request:

- Method:** DELETE
- URL:** http://localhost:8000/api/tasks/64798d42389011439d8b4613
- Authorization Type:** Bearer Token

A tooltip for the Authorization section states: "Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables." Below this, it says: "The authorization header will be automatically generated when you send the request. Learn more about authorization."

The response section shows a status of 401 Unauthorized with a message: "Status: 401 Unauthorized Time: 65 ms Size: 281 B". The response body is displayed in JSON format:

```
1: {  
2:   "message": "User is not authorized to delete"  
3: }
```

Figure 5.19: POSTMAN with DELETE route with wrong token

We will give the correct token of the user, who has this task. Now, we are able to delete the task and get the correct message back with Status 200.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar is followed by several icons. On the right side of the header, there are buttons for 'Initiate', 'Upgrade', and a close button.

In the main workspace, there is a collection named 'MERN-Task / Delete Task'. Below it, a specific request is defined:

- Method:** DELETE
- URL:** http://localhost:3001/api/tasks/64798d42389011439d8b4113
- Authorization:** Bearer Token

The 'Authorization' tab is selected. A note states: "Headers will be automatically generated when you send the request. Learn more about authorization." Below this, a 'Token' field contains a long string of characters: eyJhbGciOiJIaWNrMSczIjkpVClJ... .

At the bottom of the request panel, there are tabs for Body, Cookies, Headers, and Test Results. The Body tab is selected, showing a JSON response:

```
{"id": "64798d42389011439d8b4113"}
```

The status bar at the bottom right indicates: Status: 200 OK, Time: 143 ms, Size: 168 B, Save as Example, and a refresh icon.

Figure 5.20: POSTMAN with DELETE route with right token

Also, we are confirming the same by going to <http://localhost:8000/api/tasks> and giving this user's token. Here, we can see only one task, with the other being deleted.

Home Workspaces API Network Explore

Search Postman

Invite Upgrade

No Environment

MERN-Task / Protected Tasks Route

GET http://localhost:8000/api/tasks

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about [authorization](#).

Token eyJhbGciOiJFUzI1NiJ9R5cCMkpkXVC9eyjPZD8jYONsg4YmM2MzVmNGFYTYYONGE2NjAxNSlsmhCQ8MTY4NTY4NzU4OCw2XhwjoxNjg2MTE3Ntg4IQ.aOLWcd82DMyt8soR8rI4zZrauceRFj5KuW08U1baQI

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "_id": "6479005a309011439d0b441b",
3   "text": "Learn US English",
4   "userId": "647900cc63514ec464a864015",
5   "createdAt": "2023-06-02T06:34:02.629Z",
6   "updatedAt": "2023-06-02T06:34:02.629Z",
7   "__v": 0
8 }
```

Status 200 OK Time 56 ms Size 416 B Save as Example

Figure 5.21: POSTMAN with GET route with right token

Now, in Postman we will do a PUT request with a url like <http://localhost:8000/api/tasks/64798d5a389011439d8b4616>. Here, take any task id of a user. Now, when we hit the Send button, we will get an error, since we have not provided any token.

The screenshot shows the Postman application interface. At the top, there are navigation links: Home, Workspaces, API Network, and Explore. A search bar is followed by an 'invite' button, a gear icon, and an 'Upgrade' dropdown. On the right, there are icons for settings, a refresh, and a close button.

In the main area, there's a list of requests: POST Login User, GET Protected Tasks Route, and PUT Put Task. The PUT Put Task request is selected. To its right, there are buttons for 'Save' and 'Edit'.

The request details panel shows the method as PUT, the URL as <http://localhost:3000/api/tasks/64798d5a389011439d8b4616>, and a 'Send' button. Below this, tabs for Params, Authorization, Headers, Body (selected), Pre-request Script, Tests, and Settings are visible. The Body tab shows the content type as 'x-www-form-urlencoded'. There is one key-value pair: 'text' with the value 'Learn United States English'.

The response panel at the bottom shows a status of 401 Unauthorized. The response body is a JSON object with a single key 'message': "Not authorized, no token".

Figure 5.22: POSTMAN with PUT route with wrong token

This time we will give the correct token for the user and the task will be updated.

The screenshot shows the Postman application interface. At the top, there are navigation links for Home, Workspaces, API Network, and Explore. A search bar is followed by buttons for Invite, Upgrade, and other account options. Below the header, a collection named "MERN-Task / PUT Task" is selected. The main area displays a PUT request to the URL `http://localhost:3000/api/tasks/6479865a38901143908b4616`. The "Authorization" tab in the request settings is active, showing a "Type" dropdown set to "Bearer Token" and a "Token" input field containing a long JWT token. The "Body" tab is selected in the response preview, showing a JSON object with the following structure:

```
1 {  
2   "_id": "6479865a38901143908b4616",  
3   "text": "Learn United States English",  
4   "user": "64700bc838f4aca644a66015",  
5   "createdAt": "2023-06-02T06:34:02.629Z",  
6   "updatedAt": "2023-06-02T06:53:14.855Z",  
7   "__v": 0  
8 }
```

The response preview shows a status of 200 OK with a time of 119 ms and a size of 427 B. There are buttons for Save as Example and more options.

Figure 5.23: POSTMAN with PUT route with right token

Also, we are confirming the same by going to <http://localhost:8000/api/tasks> and giving this user's token. Here, we can see that the task has been updated.

Home Workspaces API Network Explore

Search Postman

Invite

Upgrade

No Environment

MEDIA-Task / Protected Tasks Route

GET http://localhost:8000/api/tasks

Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about [authorization](#).

Token

```
eyJhbGciOiJUzI1NiJ9.R0cI0ikpXVCJ9eyJpZD1jYONza4VmV2Mz1mNGFjTY0AGE2NjAeNgIiimhdC0MTY4NTY4NzU4OGwiZkhwIiorNjg2MTESNVtg4Q.oJLWcf620MV8soR9rI4zZrauceRFfcBKuW0$UTBqQI
```

Body Cookies Headers (7) Test Results

Status 200 OK Time: 10 ms Size: 429 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {
2   "_id": "64798d5a38001189d8b616",
3   "text": "Learn United States English",
4   "userId": "647880c635c40ca644056013",
5   "createdAt": "2023-06-02T06:34:02.629Z",
6   "updatedAt": "2023-06-02T06:53:14.855Z",
7   "__v": 0
8 }
```

Figure 5.24: POSTMAN with GET route with right token

Conclusion

In this chapter, we have learned about creating Auth middleware. Here, we used the JWT token created in the previous chapter to do authentication. Next, we learned to protect routes using this Auth middleware.

Here, we first protected the current route. After that, we protected all Task routes, which include GET, POST, PUT, and DELETE routes.

In the next chapter, we will start with the frontend with ReactJS. We are also going to do React Router setup and create Register and Login pages.

Points to remember

We have created auth middleware using json web token. Here, we have verified it using the passed token and the JWT secret.

We have protected various routes using the auth middleware. Now, routes can be accessed only after providing a valid token.

We generated the token in Postman and used the token through Postman in protected routes.

Multiple Choice Questions

Authorization tokens start with the Keyword.

Header

Creator

Bearer

Berer

To protect routes we add _____ to it.

Middle Layer

Middleware

Tailware

Headware

In Postman, we add Token in which tab?

Params

Headers

Authorization

Test

In PUT route, we add both _____ and _____ in Postman.

body, params

headers, tests

params, Authorization

body, tests

In DELETE route, we add only ____ in Postman.

body

headers

params

tests

Answers

c

b

c

a

c

CHAPTER 6

Creating Frontend and React Router

Introduction

This chapter talks about the creation of frontend using ReactJS. We will also cover the setup for React Router version 6 in the project, which is used for navigation in React. We will also create the Header, Register, and Login components in the frontend.

Structure

In this chapter, we are going to discuss the following topics:

Creating Frontend with ReactJS

Basic Project setup

React Router setup

Creating Components and Pages

[Creating Frontend with ReactJS](#)

The browser that shows the HTML, CSS, and JavaScript site doesn't understand ReactJS. So, we installed React with the famous create-react-app from Facebook. Here, we are also adding the template to add the basic structure of Redux in our application.

Note that we have to run the following command inside the app:

```
npx create-react-app frontend --template redux
```

```
npm install react react-dom n X + v
naben@Nabendu MINGW64 ~ /Desktop/Task-KERN
$ npx create-react-app frontend --template redux
Creating a new React app in C:\Users\naben\Desktop\Task-KERN\frontend.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template-redux...
[redacted] / idealTree:webpack-dev-server: 3.11.1 fetch manifest emojis-list@3.0.0
```

Figure 6.1: Creating React app

If everything is perfect, we will get the message, Happy hacking!. After that, we also opened the whole app in VS Code by giving the command code.

```
MINGW64/c/Users/naben/D... + v
```

Created git commit.

Success! Created frontend at C:\Users\naben\Desktop\Task-MERN\frontend
Inside that directory, you can run several commands:

`npm start`

Starts the development server.

`npm run build`

Bundles the app into static files for production.

`npm test`

Starts the test runner.

`npm run eject`

Removes this tool and copies build dependencies, configuration files
and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

```
cd frontend
```

```
npm start
```

Happy hacking!

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN
```

```
$ code .
```

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN
```

```
$ |
```

Figure 6.2: Successful React app

Now, as per the instructions, we will change to the new folder of frontend with the cd command. Also, run npm start to start our frontend app.

```
naben@Nabendu: ~[NG64] ->/Desktop/Task-HEN
```

```
$ cd Frontend/
```

```
naben@Nabendu: ~[NG64] ->/Desktop/Task-HEN/foreground (master)
```

```
$ open start
```

```
> Frontend@1.0 start
```

```
> react-scripts start
```

Figure 6.3: Starting React app

After some statements, we will get a message of frontend been started on <http://localhost:3000/>.

```
Windows PowerShell X + v
Compiled successfully)

You can now view frontend in the browser.

Local: http://localhost:3000
On Your Network: http://192.168.29.252:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

Figure 6.4: React App started

Open any browser and go to `http://localhost:3000/` url and we will see the default content for a React app with Redux template.



- 0 +

2 Add Amount Add Async Add If Odd

Edit `src/App.js` and save to reload.

Learn [React](#), [Redux](#), [Redux Toolkit](#), and [React Redux](#)

Figure 6.5: React Redux app on browser

Basic Project Setup

First, we will remove this basic boilerplate code to start with our project setup.
So, in VS Code delete the following files in the screenshot.

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File Edit Selection View Go Run ... reportWebVitals.js - Task-MERN - Visual Studio Code
- Toolbar:** Standard icons for file operations.
- Explorer View:** Shows the project structure under "TASK-MERN".
 - backend
 - frontend (selected)
 - node_modules
 - public
 - src
 - app
 - features
 - # App.css (highlighted)
 - JS App.js
 - JS App.test.js
 - # index.css
 - JS index.js
 - logo.svg (highlighted)
 - JS reportWebVitals.js (highlighted)
 - JS setupTests.js
 - gitignore
 - { package-lock.json
 - { package.json
 - README.md
- OUTLINE
- TIMELINE

- Context Menu (open over reportWebVitals.js):**
- Open to the Side (Ctrl+Enter)
- Open With...
- Reveal in File Explorer (Shift+Alt+R)
- Open in Integrated Terminal
- Select for Compare
- Open Timeline
- Cut (Ctrl+X)
- Copy (Ctrl+C)
- Copy Path (Shift+Alt+C)
- Copy Relative Path (Ctrl+K Ctrl+Shift+C)
- Rename... (F2)
- Delete (highlighted)
- Status Bar:** master ① 0 Δ 0 Ln 1, Col 1 Spaces:2 UTF-8 LF {} JavaScript Go Live

Figure 6.6: Removing boilerplate files

Next, we will remove some code referring to the preceding deleted items and update our index.js file as follows:

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import { Provider } from 'react-redux';
import { store } from './app/store';
import App from './App';
import './index.css';

const container = document.getElementById('root');
const root = createRoot(container);

root.render(
  <>
    <Provider store={store}>
      <App />
    </Provider>
  </>
)
```

</>

);

The screenshot shows a Visual Studio Code interface with a dark theme. The title bar reads "index.js - Task-MFRN - Visual Studio Code". The left sidebar has icons for file operations like Open, Save, and Close, and a search icon. The main editor area contains the following code:

```
index.js M X

frontend > src > index.js > ...

1 import React from 'react';
2 import { createRoot } from 'react-dom/client';
3 import { Provider } from 'react-redux';
4 import { store } from './app/store';
5 import App from './App';
6 import './index.css';
7
8 const container = document.getElementById('root');
9 const root = createRoot(container);
10
11 root.render(
12   <>
13     <Provider store={store}>
14       <App />
15     </Provider>
16   </>
17 );
18
```

Figure 6.7: Basic index.js file

We will also delete all code from the App.js file and only keep the following code. It will update our app running in the browser on <http://localhost:3000/> as shown in the screenshot.

```
function App() {  
  return (  
    <div>  
      <h1>React App</h1>  
    </div>  
  );  
}  
  
export default App;
```

A screenshot of the Visual Studio Code interface. The title bar shows "React App" and "File Edit Selection View Go Run Terminal Apps Task Manager Visual Studio Code". The left sidebar shows a file tree with "React App" selected. The main editor area displays the following code:

```
function App() {
  return (
    <div>
      <h1>React App</h1>
    </div>
  );
}

export default App;
```

The code editor uses a dark theme with syntax highlighting. The entire code block is highlighted with a green rectangular selection.

Figure 6.8: Basic App.js file

We will also remove the earlier content from the store.js file inside the app folder and add the following content. This is the basic code for store of reducer in our app.

```
import { configureStore } from '@reduxjs/toolkit';
```

```
export const store = configureStore({  
  reducer: {  
    },  
});
```

A screenshot of the Visual Studio Code interface. The top bar shows the menu: File, Edit, Selection, View, Go, Run, Terminal, Help, store.js · Task-MERN · Visual Studio Code. The title bar indicates the current file is store.js. The left sidebar is the Explorer view, showing the project structure: TASK-MERN (backend, frontend, node_modules, public, src, app, store.js, features). The store.js file is selected and highlighted with a green border. The main editor area shows the following code:

```
1 import { configureStore } from '@reduxjs/toolkit';
2
3 export const store = configureStore({
4   reducer: [
5     ,
6   ],
7});
```

Figure 6.9: Basic store.js file

Also, delete the whole content of the index.css file and add the following content to it. This will contain all of our styles for the project.

```
@import url('https://fonts.googleapis.com/css2?  
family=Karla:wght@300;400;600;700&display=swap');
```

```
* {  
margin: ·;  
padding: ·;  
box-sizing: border-box;  
}
```

```
body {  
font-family: <Karla>, sans-serif;  
height: 100vh;  
}
```

```
a {  
text-decoration: none;
```

```
color: #ccc;
```

```
}
```

```
p {
```

```
line-height: 1.4;
```

```
}
```

```
ul {
```

```
list-style: none;
```

```
}
```

```
li {
```

```
line-height: 1.4;
```

```
}
```

```
h1,
```

```
h2,
```

```
h3 {
```

```
font-weight: 100;
```

```
margin-bottom: 10px;
```

```
}
```

```
.container {
```

```
width: 100%;
```

```
max-width: 960px;
```

```
margin: auto;
```

```
padding: 20px;
```

```
text-align: center;
```

```
}
```

```
.header {
```

```
display: flex;
```

```
justify-content: space-between;
```

```
align-items: center;
```

```
padding: 20px 0;
```

```
border-bottom: 1px solid #e6e6e6;
```

```
margin-bottom: 10px;
```

```
}
```

```
.header ul {
```

```
display: flex;
```

```
align-items: center;  
justify-content: space-between;  
}  
  
  

```

```
.header ul li {  
margin-left: 10px;  
}  
  
  

```

```
.header ul li a {  
display: flex;  
align-items: center;  
}  
  
  

```

```
.header ul li a:hover {  
color: #FFF;  
}  
  
  

```

```
.header ul li a svg {  
margin-right: 0px;  
}
```

```
.heading {  
  font-size: 1rem;  
  font-weight: 400;  
  margin-bottom: 0px;  
  padding: 10px;  
}  
  
.
```

```
.heading p {  
  color: #888888;  
}  
  
.
```

```
.goals {  
  display: grid;  
  grid-template-columns: repeat(1, 1fr);  
  gap: 10px;  
}  
  
.
```

```
.goal {  
  background-color: #f8f8f8;  
  margin: 10px 0;  
  padding: 10px 10px 0px;  
}
```

```
position: relative;  
}  
  
}
```

```
.goal .close {  
position: absolute;  
top: 1 · px;  
right: 10px;  
cursor: pointer;  
border: none;  
background: none;  
}
```

```
.form,  
.content {  
width: 1 · %;  
margin: 0 auto;  
}
```

```
.form-group {  
margin-bottom: 1 · px;  
}
```

```
.form-group input,  
.form-group textarea,  
.form-group select {  
width: 100%;  
padding: 10px;  
border: 1px solid #e6e6e6;  
border-radius: 0px;  
margin-bottom: 10px;  
font-family: inherit;  
}  
  
 
```

```
.form-group label {  
text-align: left;  
display: block;  
margin: 0 0 10px 0;  
}  
  
 
```

```
.btn {  
padding: 10px 20px;  
border: 1px solid #ccc;  
border-radius: 0px;  
}
```

```
background: #fff;  
color: #fff;  
font-size: 16px;  
font-weight: normal;  
cursor: pointer;  
text-align: center;  
appearance: button;  
display: flex;  
align-items: center;  
justify-content: center;  
}
```

```
.btn svg {  
margin-right: 10px;  
}
```

```
.btn-reverse {  
background: #fff;  
color: #fff;  
}
```

```
.btn-block {  
    width: 100%;  
    margin-bottom: 10px;  
}  
  
}
```

```
.btn:hover {  
    transform: scale(1.1);  
}  
  
}
```

```
.loadingSpinnerContainer {  
    position: fixed;  
    top: 50%;  
    right: 50%;  
    bottom: 50%;  
    left: 50%;  
    background-color: rgba(0, 0, 0, 0.5);  
    z-index: 1000;  
    display: flex;  
    justify-content: center;  
    align-items: center;  
}  
  
}
```

```
.loadingSpinner {  
    width: 75px;  
    height: 75px;  
    border: 1px solid;  
    border-color: #ccc transparent #000 transparent;  
    border-radius: 50%;  
    animation: spin 1.5s linear infinite;  
}  
  
@keyframes spin {  
    0% {  
        transform: rotate(0deg);  
    }  
    100% {  
        transform: rotate(360deg);  
    }  
}
```

```
@media (max-width: 600px) {  
    .form {
```

```
width: 9%;
```

```
}
```

```
.heading h1 {
```

```
font-size: 1rem;
```

```
}
```

```
.heading p {
```

```
font-size: 1.0rem;
```

```
}
```

```
}
```

A screenshot of the Visual Studio Code interface showing the file `index.css`. The code defines styles for an `a` element, the `body`, and `p` elements. It includes a font import rule and various styling rules like margin, padding, and line-height.

```
frontend > src > # index.css > a
1 @import url('https://fonts.googleapis.com/css2?family=Karla:wght@300,400,600,700&
2 display=swap');
3 *
4   margin: 0;
5   padding: 0;
6   box-sizing: border-box;
7 }
8
9 body {
10   font-family: 'Karla', sans-serif;
11   height: 100vh;
12 }
13
14 a {
15   text-decoration: none;
16   color: #000;
17 }
18
19 p {
20   line-height: 1.7;
21 }
22
23 ul {
```

Figure 6.10: Styles in index.css file

We will also delete the counter folder, which contains most of our default Redux code.

A screenshot of the Visual Studio Code interface. The title bar shows "store.js - Task-MERN - Visual Studio Code". The Explorer sidebar on the left lists project files and folders, including "backend", "frontend", "node_modules", "public", "src", "app", and "features". A context menu is open over the "store.js" file in the "app" folder. The menu items are:

- New File...
- New Folder...
- Reveal in File Explorer Shift+Alt+R
- Open in Integrated Terminal
- Find in Folder... Shift+Alt+F
- Cut Ctrl+X
- Copy Ctrl+C
- Paste Ctrl+V
- Copy Path Shift+Alt+C
- Copy Relative Path Ctrl+K Ctrl+Shift+C
- Rename... F2
- Delete

The "Delete" option at the bottom of the menu is highlighted with a green border.

Bottom status bar: master* 0 △ 0 Ln 5, Col 1 Spaces:2 UTF-8 LF {} JavaScript Go Live R Q

Figure 6.11: Removing counter folder

Now, we will create some of the components which we will use in our project. So, first create a components folder inside src and add a Dashboard.js file in it. Add the following basic content to it:

```
import React from 'react'
```

```
const Dashboard = () => {
```

```
    return (
```

```
        <div>Dashboard</div>
```

```
    )
```

```
}
```

```
export default Dashboard
```

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** On the left, it displays the project structure of a "TASK-MERN" application. The "components" folder under "src/app" is highlighted with a green border.
- Code Editor:** The main area shows the content of the "Dashboard.js" file. The code is as follows:

```
1 import React from 'react'
2
3 const Dashboard = () => {
4   return (
5     <div>Dashboard</div>
6   )
7 }
8
9 export default Dashboard
```

The code editor has syntax highlighting for JavaScript and React components. The "Dashboard" component is currently selected in the code editor.

Figure 6.12: Creating Dashboard.js file

Next, create a Login.js file inside the components folder and add the basic code to it.

```
import React from 'react'
```

```
const Login = () => {
```

```
    return (
```

```
        <div>Login</div>
```

```
    )
```

```
}
```

```
export default Login
```

The screenshot shows the Visual Studio Code interface with a dark theme. The title bar reads "Login.js - Task-MERN - Visual Studio Code". The left sidebar is the Explorer view, showing the project structure of "TASK-MERN". The "src" folder contains "app", "components", "Dashboard.js", and "Login.js". "Login.js" is currently selected and highlighted with a green border. The main editor area displays the following code:

```
1 import React from 'react'
2
3 const Login = () => {
4   return (
5     <div>Login</div>
6   )
7 }
8
9 export default Login
```

Figure 6.13: Creating Login.js file

Lastly, create a Register.js file inside the components folder and add the basic code to it.

```
import React from 'react'
```

```
const Register = () => {
```

```
    return (
```

```
        <div>Register</div>
```

```
    )
```

```
}
```

```
export default Register
```

The screenshot shows a dark-themed interface of Visual Studio Code. At the top, the menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and Register.js - Task-MERN - Visual Studio Code. The title bar also displays "Register.js - Task-MERN - Visual Studio Code". The left sidebar is the Explorer view, showing a file tree for a project named "TASK-MERN". The "frontend" folder contains "backend", "public", "src" (which has "app", "components" with "Dashboard.js", "Login.js", and "Register.js", and "features"), "App.js", "App.test.js", "#index.css", "index.js", and "setupTests.js". It also includes ".gitignore", "package-lock.json", "package.json", and "README.md". The "Register.js" file is currently selected and open in the main editor area. The code is as follows:

```
1 import React from 'react'
2
3 const Register = () => {
4   return (
5     <div>Register</div>
6   )
7 }
8
9 export default Register
```

Figure 6.14: Creating Register.js file

React Router Setup

Now, we will set up a react router in our project. React router is an external package that needs to be installed in the project, because react doesn't come in-built with routing functionalities.

So, in the frontend folder in the terminal install react router using the following command:

```
npm i react-router-dom
```

Also, we will update our App.js file as follows. Here, we are first importing BrowserRouter, Routes and Route from react-router-dom. After that, we are using it inside the return statement as shown here:

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

```
function App() {
  return (
    <Router>
      <div className=>container>
        <Routes>
```

```
</Routes>

</div>

</Router>

);

}

export default App;
```

A screenshot of the Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, Apps, Task MERN, and Visual Studio Code. The title bar shows the file path: frontend > src > App.js > App. The code editor displays the following code:

```
1 import { BrowserRouter as Router, Routes, Route } from 'react-router-dom'  
2  
3 function App() {  
4   return (  
5     <Router>  
6       <div className='container'>  
7         <Routes>  
8           </Routes>  
9         </div>  
10      </Router>  
11    );  
12  }  
13 }
```

The code editor has a green selection box highlighting the Router component and its child components. Below the code editor is a navigation bar with links: PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, showing a terminal session:

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN  
$ cd frontend/  
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/frontend (master)  
$ npm i react-router-dom
```

The terminal shows the command to change directory to the frontend folder and the command to install the react-router-dom package. A message "Run 'npm audit' for details." is displayed. The bottom status bar shows the file name master*, line 8, column 1, and other status indicators.

Figure 6.15: Adding react router

Next, we will import the three components of Dashboard, Login, and Register in App.js. After that, inside the <Routes>, we will add the three components with different paths.

```
import Dashboard from './components/Dashboard';
import Login from './components/Login';
import Register from './components/Register';
...
...
<Route path='/' element={<Dashboard />} />
<Route path='/login' element={<Login />} />
<Route path='/register' element={<Register />} />
```

A screenshot of a Visual Studio Code editor window. The title bar shows the file path: 'frontend > src > App.js > App'. The code editor displays the following React component definition:

```
1 import { BrowserRouter as Router, Routes, Route } from 'react-router-dom'
2 import Dashboard from './components/Dashboard';
3 import Login from './components/Login';
4 import Register from './components/Register';
5
6 function App() {
7   return (
8     <Router>
9       <div className='container'>
10         <Routes>
11           <Route path='/' element={<Dashboard />} />
12           <Route path='/login' element={<Login />} />
13           <Route path='/register' element={<Register />} />
14         </Routes>
15       </div>
16     </Router>
17   );
18 }
19
20 export default App;
```

The code uses the `BrowserRouter` from `react-router-dom` to handle routing. It defines a `Routes` component with three `Route` components: one for the root path ('/') mapping to `Dashboard`, one for the '/login' path mapping to `Login`, and one for the '/register' path mapping to `Register`. The `Dashboard`, `Login`, and `Register` components are imported from their respective files in the `./components` directory.

Figure 6.16: Basic routes

Now, in `http://localhost:3000/` we will see our basic Dashboard component.

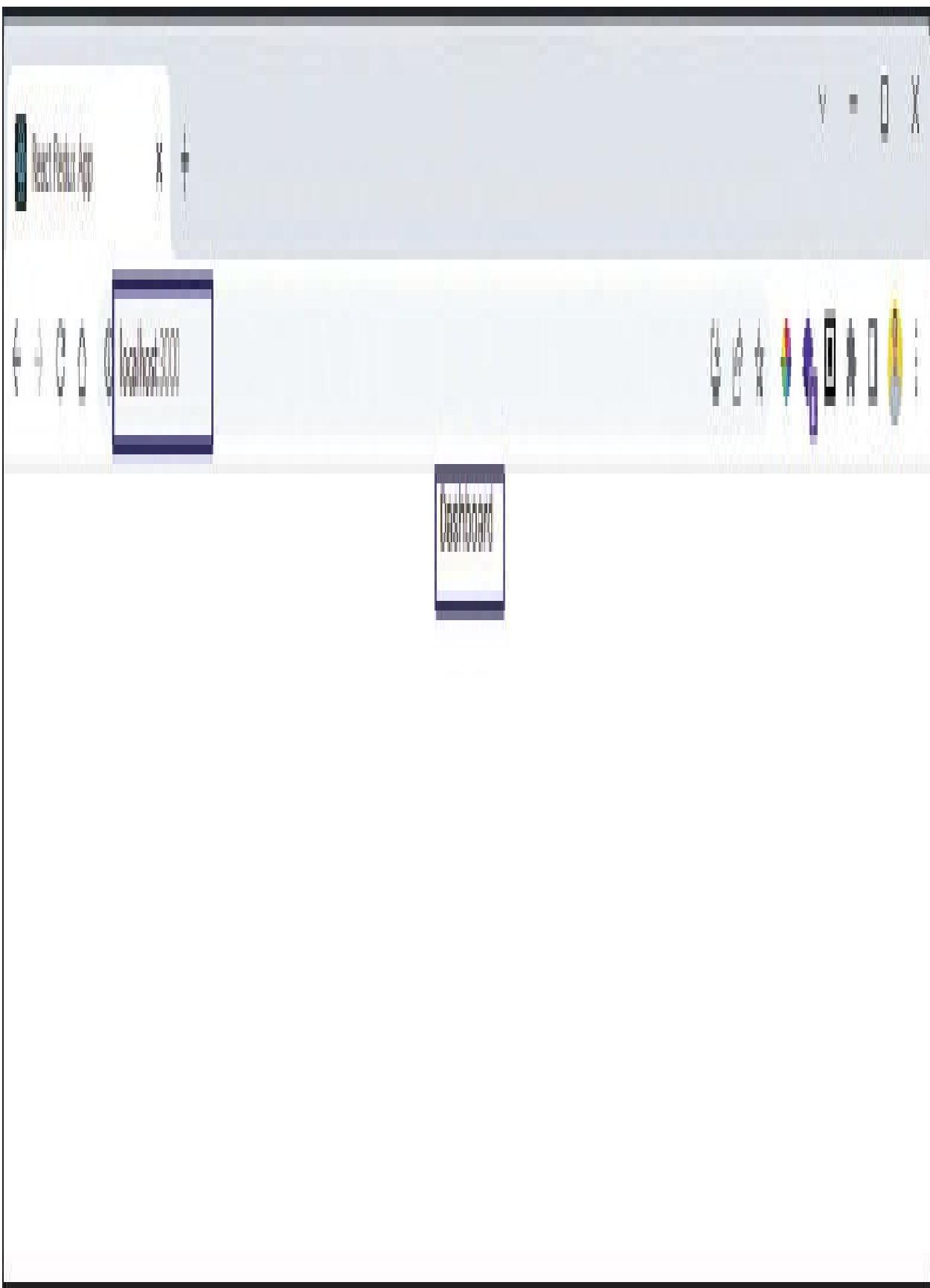


Figure 6.17: Dashboard route

In `http://localhost:3000/login`, we will see our Login component code.



Figure 6.18: Login route

And in `http://localhost:3000/register` we will see our Register component code.

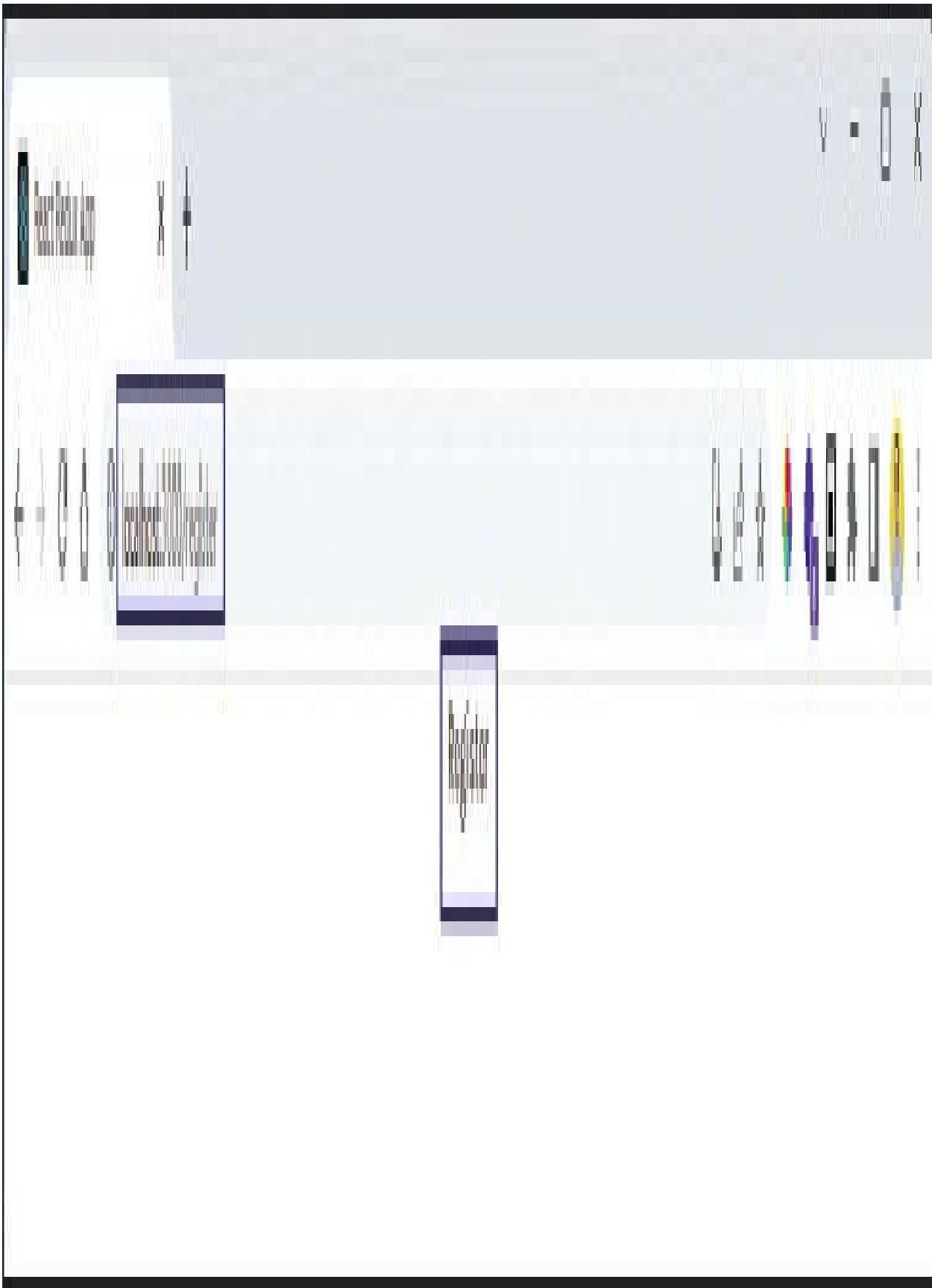


Figure 6.19: Register route

Creating Components and Pages

Now, we will create some of the components in our project. First, we will create the common component of the Header, which will be shown on all pages. Following that, we will create the pages for Register and Login.

Header Component

We will first create the common Header component. So, create a file called Header.js inside the components folder. Here, we have first opened the integrated terminal and installed the icon package of react-icons, using the following command.

```
npm i react-icons
```

Next, we have created the basic structure of the Header component. Here, first, we are importing some icons from react-icons. And we are also importing Link from react-router-dom. Inside the return statement, we have the basic header with a Link showing Task Creator to the home route.

```
import { FaSignInAlt, FaSignOutAlt, FaUser } from 'react-icons/fa'
```

```
import { Link } from 'react-router-dom'
```

```
const Header = () => {
```

```
  return (
```

```
    <header className=>header>
```

```
    <div className=>logo</div>
```

```
    <Link to=>/>Task Creator</Link>
```

```
  </div>
```

```
</header>
```

```
)
```

```
}
```

```
export default Header
```

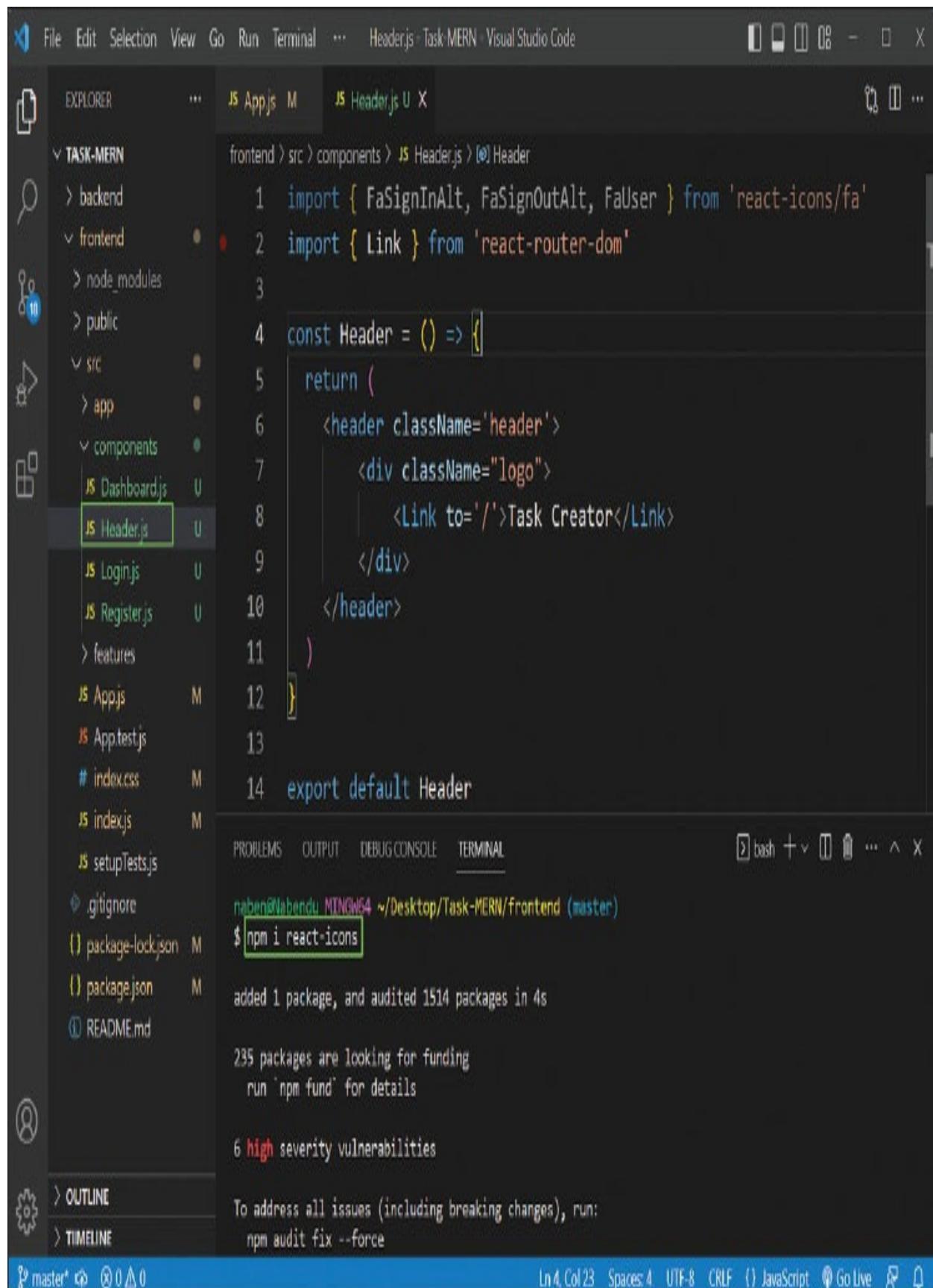


Figure 6.20: Creating header component

Next, in the Header.js file, we will also add the other Links for /login and /register route, inside an ul tag. Here, we are also using the icons imported earlier. Note that on clicking these links, we will be taken to the respective route, showing that component.

```
<ul>  
  <li>  
    <Link to=>/login>>  
    <FaSignInAlt /> Login  
  </Link>  
  </li>  
  <li>  
    <Link to=>/register>>  
    <FaUser /> Register  
  </Link>  
  </li>  
</ul>
```

File Edit Selection View Go Run Terminal Help Header.js - Visual Studio Code

frontend > src > components > Header.js

```
1 import { FaSignInAlt, FaSignOutAlt, FaUser } from 'react-icons/fa'
2 import { Link } from 'react-router-dom'
3
4 const Header = () => {
5   return (
6     <header className='header'>
7       <div className="logo">
8         <Link to="/">Task Creator</Link>
9       </div>
10      <ul>
11        <li>
12          <Link to='/login'>
13            <FaSignInAlt /> Login
14          </Link>
15        </li>
16        <li>
17          <Link to='/register'>
18            <FaUser /> Register
19          </Link>
20        </li>
21      </ul>
22    </header>
23  )
24}
```

Ln 10, Col 15 Spaces: 4 UTF-8 () JavaScript Go Live ⌂ ⌂

Figure 6.21: Adding routes in header component

We will finally add the Header component inside our App.js file. But notice here, we are adding it outside the <Routes>. It means that it will be shown on all pages.

The screenshot shows a Visual Studio Code interface with the following details:

- File Bar:** File Edit Selection View Go Run Terminal Help Apps - Task MERN - Visual Studio Code
- Editor:** The main editor area displays `App.js` content. The code imports `BrowserRouter`, `Dashboard`, `Login`, `Register`, and `Header` components.
- Code Content:**

```
1 import { BrowserRouter as Router, Routes, Route } from 'react-router-dom'
2 import Dashboard from './components/Dashboard';
3 import Login from './components/Login';
4 import Register from './components/Register';
5 import Header from './components/Header';

6
7 function App() {
8   return (
9     <Router>
10       <div className='container'>
11         <Header />
12         <Routes>
13           <Route path='/' element={<Dashboard />} />
14           <Route path='/login' element={<Login />} />
15           <Route path='/register' element={<Register />} />
16         </Routes>
17       </div>
18     </Router>
19   );
20 }
21
22 export default App;
```
- Left Sidebar:** Shows file navigation with `App.js` currently selected.
- Bottom Status Bar:** In 11, Col 19 | Spaces: 2 | UTF-8 | LF | {} JavaScript | Go Live |

Figure 6.22: Adding header component in App.js

Now, go to `http://localhost:3000/` and we will see a nice-looking header with logo, and two icons for Login and Register.

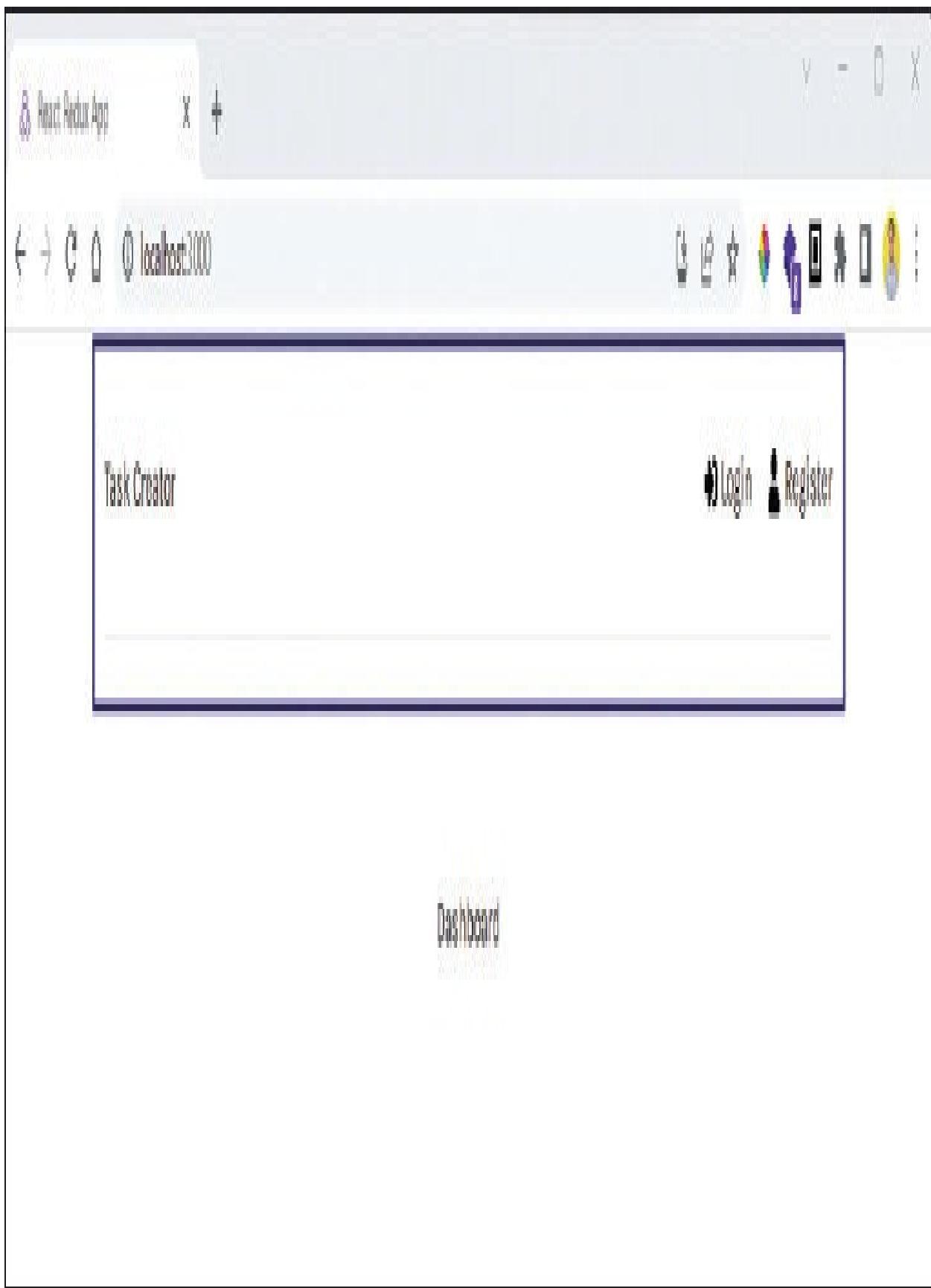


Figure 6.23: Header component in browser

Register Page

Now, we will create our Register page. We can go to this page by clicking the Register logo in the dashboard.

First, replace the basic code with the following code in Register.js file. Here, we are first importing useState and an icon. Next, we are first creating a state variable of formData which will be equal to an object containing name, email, password and password2.

We are also destructuring the formData. Inside the return statement, we have a section showing a heading and a paragraph.

Inside the second section, we have a form that we will complete next.

```
import { useState } from “react”
import { FaUser } from ‘react-icons/fa’

const Register = () => {
  const [formData, setFormData] = useState({ name: ‘’, email: ‘’, password: ‘’, password2: ‘’ })
  const { name, email, password, password2 } = formData
  const onChange = () => {}
```

```
return (
```

```
<>
```

```
<section className=>heading>
```

```
<h1><FaUser /> Register</h1>
```

```
<p>Please create an account</p>
```

```
</section>
```

```
<section className=>form>
```

```
<form onSubmit={onSubmit}>
```

```
</form>
```

```
</section>
```

```
</>
```

```
)
```

```
}
```

```
export default Register
```

A screenshot of Visual Studio Code displaying a React component named `Register`. The code uses functional components and hooks like `useState` and `useEffect`. It includes imports for `useState` from `react` and `faUser` from `'react-icons/fa'`. The component structure features a heading section with a logo and a form section containing a form with an onSubmit handler. The code ends with an export statement.

```
File Edit Selection View Go Run Terminal Help Register - 100% - 0 X
B App.js M B Register.js X B Header.js V
frontend>src>components>B Register.js > Register
1 import { useState } from "react"
2 import { FaUser } from 'react-icons/fa'
3
4 const Register = () => {
5   const [formData, setFormData] = useState({ name: '', email: '', password: '', password2: '' })
6   const { name, email, password, password2 } = formData
7   const onChange = () => []
8
9   return (
10     <>
11       <section className='heading'>
12         <h1><FaUser /> Register</h1>
13         <p>Please create an account</p>
14       </section>
15       <section className='form'>
16         <Form onSubmit={onSubmit}>
17           </Form>
18       </section>
19     </>
20   )
21 }
22
23 export default Register
```

Figure 6.24: Creating register component

Inside the form in Register.js file, we will have three input statements for the name, email, and password. These inputs are like what we have in React for input. Also, notice that we have a name tag in all inputs.

```
<div className=>form-group>

<input type=>text> className=>form-control> id=>name> name=>name> value={name} placeholder=>Enter your name> onChange={onChange}

/>

</div>

<div className=>form-group>

<input type=>email> className=>form-control> id=>email> name=>email> value={email} placeholder=>Enter your email> onChange={onChange}

/>

</div>

<div className=>form-group>

<input type=>password> className=>form-control> id=>password>
name=>password> value={password} placeholder=>Enter password> onChange={onChange}

/>

</div>
```

File Edit Selection View Go Run Terminal Help Register.js - Task MERN - Visual Studio Code

```
App.js Register.js Header.js
frontend > src > components > Register.js > Register
  14  </section>
  15  <section className='form'>
  16    <form onSubmit={onSubmit}>
  17      <div className='form-group'>
  18        <input type='text' className='form-control' id='name' name='name' value={name} placeholder='Enter
  19          your name' onChange={onChange}>
  20        </div>
  21        <div className='form-group'>
  22          <input type='email' className='form-control' id='email' name='email' value={email} placeholder='Enter
  23            your email' onChange={onChange}>
  24          </div>
  25        <div className='form-group'>
  26          <input type='password' className='form-control' id='password' name='password' value={password}
  27            placeholder='Enter password' onChange={onChange}>
  28          </div>
  29        </form>
  30      </section>
  31    </>
  32  )
  33 }
```

In 28 Col 17 Spaces: 2 CRLF {} JavaScript Go live

Figure 6.25: Adding fields in register component

We will also add the input for password2, which is our confirm password field. Also, we added our Submit button in Register.js file.

```
<div className=>form-group>

<input type=>password> className=>form-control> id=>password2>
name=>password2> value={password2} placeholder=>Confirm password>
onChange={onChange}

/>

</div>

<div className=>form-group>

<button type=>submit> className=>btn btn-block>Submit</button>

</div>
```

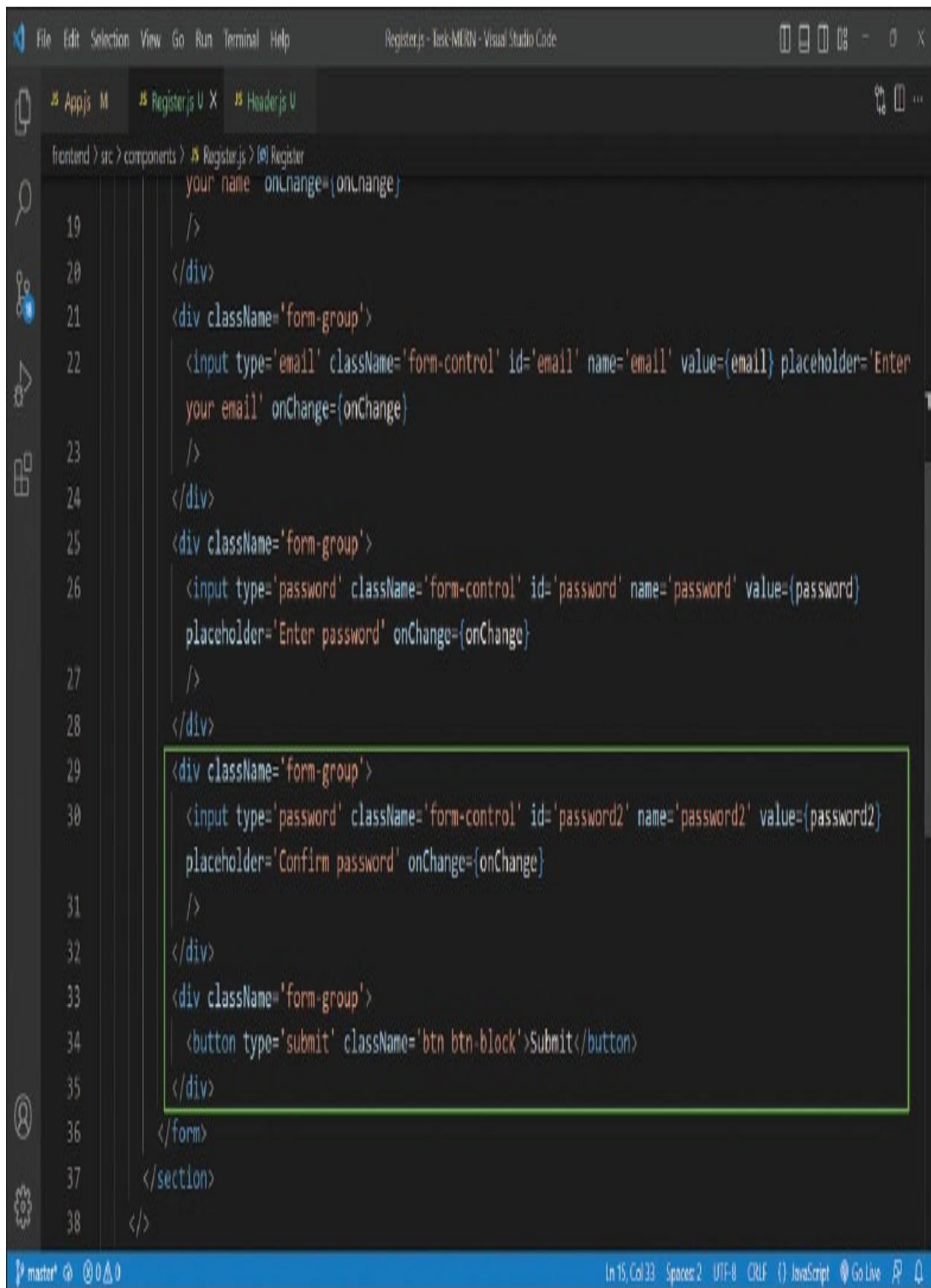


Figure 6.26: Adding more fields in register component

Next, we will complete our onChange function in the Register.js file. Here, we are using e.target.name to get all input dynamically. Or else, we would have needed to create different onChanges for every input field. The onSubmit is now empty, and logic will be added in later chapters.

```
const onChange = e => {  
  setFormData(prevState => ({  
    ...prevState,  
    [e.target.name]: e.target.value  
  }))  
}  
  
const onSubmit = e => {  
  e.preventDefault()  
}
```

```
File Edit Selection View Go Run Terminal Help
Register.js (148 lines, 10.3 KB)
App.js Register.js Header.js
Backend > src > components > Register.js (148 lines, 10.3 KB)
1 import { useState } from "react"
2 import { FaUser } from 'react-icons/fa'
3
4 const Register = () => {
5   const [formData, setFormData] = useState({ name: '', email: '', password: '', password2: '' })
6   const { name, email, password, password2 } = formData
7
8   const onChange = e => {
9     setFormData(prevState => [
10       ...prevState,
11       [e.target.name]: e.target.value
12     ])
13   }
14
15   const onSubmit = e => {
16     e.preventDefault()
17   }
18
19   return (

```

Figure 6.27: Adding functions in register component

On clicking on the register logo, we will be taken to <http://localhost:3000/register>. And the nice-looking Register component will be shown.

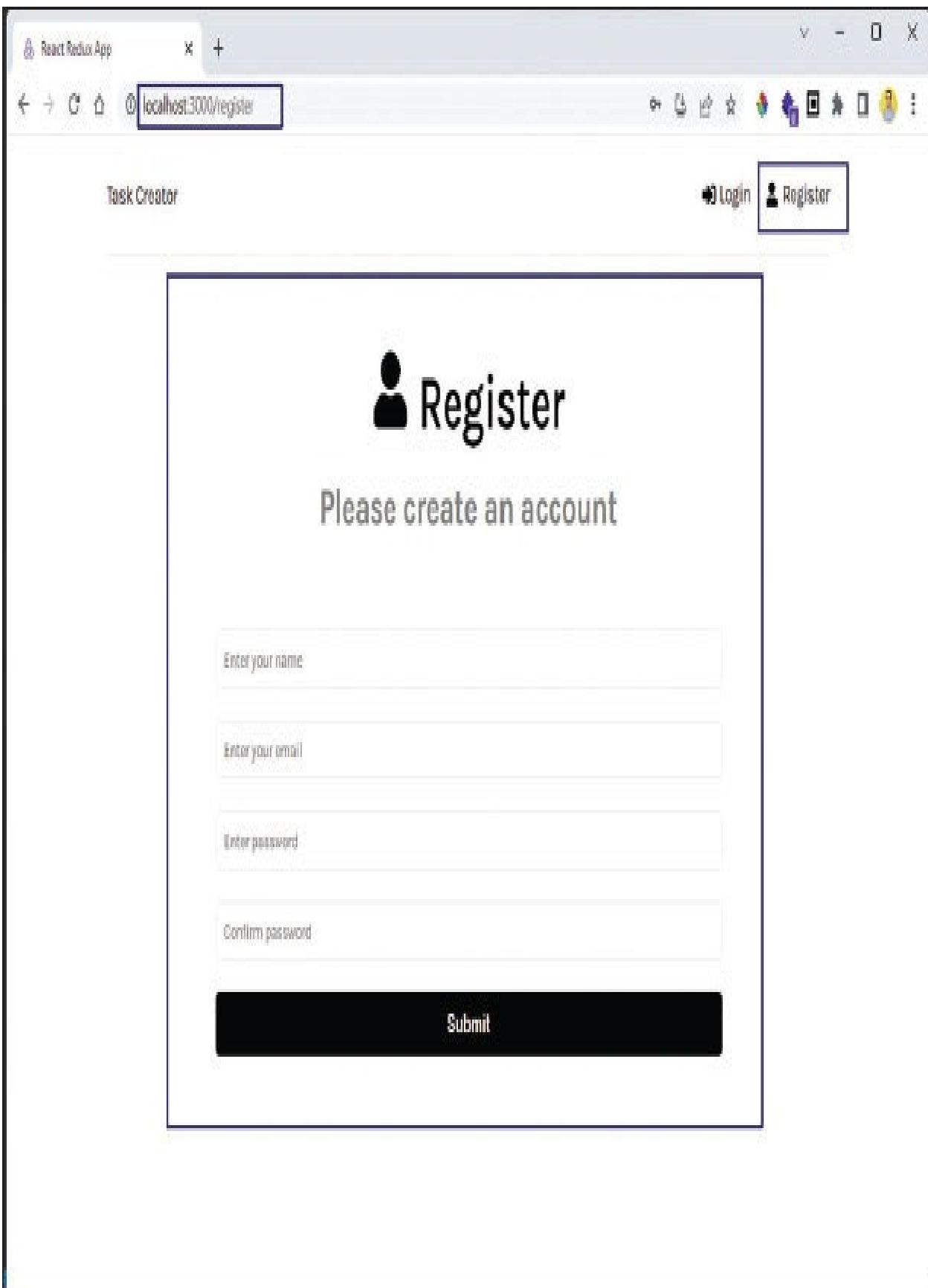


Figure 6.28: Register page in browser

Login Page

Now, we will create the Login page, which is quite similar to the Register page created in the earlier section. First, create a file Login.js in the components folder and add the following content to it.

Here, we again have the formData, but with fewer fields of email and password. The onChange and onSubmit are also similar to the Register page.

```
import { FaSignInAlt } from 'react-icons/fa'

import { useState } from "react"

const Login = () => {

  const [formData, setFormData] = useState({ email: '', password: '' })

  const { email, password } = formData

  const onChange = e => {

    setFormData(prevState => ({
      ...prevState,
      [e.target.name]: e.target.value
    }))
  }

  const onSubmit = e => {
```

```
e.preventDefault()
```

```
}
```

```
return (
```

```
<div>Login</div>
```

```
)
```

```
}
```

```
export default Login
```

File Edit Selection View Go Run Terminal Help Login.js - Task MERN - Visual Studio Code

App.js M Register.js U Login.js X

frontend>src>components>Login.js>onSubmit

```
1 import { FaSignInAlt } from 'react-icons/fa'
2 import { useState } from "react"
3
4 const Login = () => {
5   const [formData, setFormData] = useState({ email: '', password: '' })
6   const { email, password } = formData
7   const onChange = e => {
8     setFormData(prevState => ({
9       ...prevState,
10      [e.target.name]: e.target.value
11    }))
12  }
13
14  const onSubmit = e => [
15    e.preventDefault()
16  ]
17
18  return (
19    <div>Login</div>
20  )
21}
22
23 export default Login
```

Ln 16 Col 4 Spaces:2 UTF-8 CRLF ↗ JavaScript ⚡ Go live ⌂ ⌂

Figure 6.29: Creating login component

Now, inside the return statement, we will have a section with a header and another with a form. The form section will only contain an email, password, and button.

```
<><

<section className=>heading>

<h1><FaSignInAlt /> Login</h1>

<p>Login and start creating tasks</p>

</section>

<section className=>form>

<form onSubmit={onSubmit}>

<div className=>form-group>

<input type=>email> className=>form-control> id=>email> name=>email> value={email} placeholder=>Enter your email> onChange={onChange}

/>

</div>

<div className=>form-group>

<input type=>password> className=>form-control> id=>password> name=>password> value={password} placeholder=>Enter password> onChange={onChange}
```

```
/>

</div>

<div className=>form-group>

<button type=>submit> className=>btn btn-block>Submit</button>

</div>

</form>

</section>

</>
```

File Edit Selection View Go Run Terminal Help Login.js - Task MERN - Visual Studio Code

App.js M Register.js U Login.js X

frontend > src > components > Login.js (0) Login

```
19 <>
20   <section className='heading'>
21     <h1>FaSignInAlt /> Login</h1>
22     <p>Login and start creating tasks</p>
23   </section>
24   <section className='form'>
25     <form onSubmit={onSubmit}>
26       <div className='form-group'>
27         <input type='email' className='form-control' id='email' name='email' value={email} placeholder='Enter your email' onChange={onChange}>
28       </div>
29     </div>
30     <div className='form-group'>
31       <input type='password' className='form-control' id='password' name='password' value={password} placeholder='Enter password' onChange={onChange}>
32     </div>
33   </div>
34   <div className='form-group'>
35     <button type='submit' className='btn btn-block'>Submit</button>
36   </div>
37 </form>
38 </section>
39 </>
```

master* @ ① 0 ▲ 0 Ln 29 Col 17 Spaces:2 UTF-8 CRLF ⓘ JavaScript ⓘ Go live ⌂ ⌂

Figure 6.30: Adding fields in Login Component

On clicking on the Login logo, we will be taken to <http://localhost:3000/login>. And the nice-looking login component will be shown.



Figure 6.31: Login Component in Browser

Conclusion

In this chapter, we have learned about creating frontends with ReactJS. Here, we had first created a frontend app with the `create-react-app` command. Next, we deleted some boiler plate code to do the basic setup in the app.

After that, we did the React router setup in our project. This library is required for routing in our React app. Lastly, we have created the components of Header, Register, and Login in this chapter.

In the next chapter, we will start with the redux setup using slice. Then we are going to complete the User registration along with the Register form.

Points to remember

We have created a react app with a redux template using the create-react-app command.

Removing unnecessary files when starting with React app, such as lovo.svg and reportWebVitals.js

Setting up the react router in the project and showing different components on different paths.

Creating components using react icons and different form elements of input and button.

CHAPTER 7

Redux Setup with Slice

Introduction

This chapter covers the implementation of Redux in the project. We will use the latest Toolkit method in Redux to implement a global state in our project. In this chapter, we are also going to set up a reducer for Registration form. Finally, we are also going to test the user registration workflow.

Structure

In this chapter, we are going to discuss the following topics:

Redux setup with a toolkit

Auth service and slice

Registration form hook up

Testing user registration

Redux setup with a toolkit

To start with Redux with a toolkit, we need to have a slice. Now, traditionally, redux has a separate action creator and reducer. But with the latest toolkit, they have been merged to form the slice. Now, we have the reducer and action creator logic both in slice.

So, create a folder auth inside the features folder. Inside it, create an authSlice.js file and add the following code in it.

Here, we are first importing createSlice and createAsyncThunk from the redux toolkit. The createSlice will be used to create the slice used for reducer and action. And the createAsyncThunk will be used for asynchronous actions, which is an API call in our case later on.

After that, we added a variable called localuser, which uses local storage to get a user variable. We are going to set this variable later.

Next, we have the initialState of our slice, which is an object. Here, we have the user key first, which is equal to the localuser initially or null if localuser is not present. We also have values for isError, isSuccess, and isLoading, which are set as false. These will be used when we do the API call. Lastly, we have a message as an empty string.

Next, we are exporting the authSlice, which is created using the createSlice method. We will complete this method next.

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
```

```
const localuser = JSON.parse(localStorage.getItem('user'))
```

```
const initialState = {  
  user: localuser ? localuser : null,  
  isError: false,  
  isSuccess: false,  
  isLoading: false,  
  message: <> }  
}
```

```
export const authSlice = createSlice({  
})
```

A screenshot of the Visual Studio Code interface showing the file structure and content of the `authSlice.js` file.

The file path is `frontend/src/features/auth/authSlice.js`.

```
1 import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
2
3 const localuser = JSON.parse(localStorage.getItem('user'))
4
5 const initialState = {
6   user: localuser ? localuser : null,
7   lsError: false,
8   lsSuccess: false,
9   isLoading: false,
10  message: ''
11 }
12
13 export const authSlice = createSlice({
14
15 })
```

Figure 7.1: Creating authSlice.js file

Now, we will write code inside the createSlice method in the authSlice.js file. Here, we are first giving the name of the slice as auth. Then we are taking the initialState. After that, we have a reducers value, which has a reset. This reset can be used to make the state back to the initial state.

Next, we have an extraReducers value, which we are going to complete soon. In the end, we are exporting the reset action and also the reducer.

```
export const authSlice = createSlice({  
  name: <auth>,  
  initialState,  
  reducers: {  
    reset: state => {  
      state.isLoading = false  
      state.isSuccess = false  
      state.isError = false  
      state.message = <>  
    },  
  },  
  extraReducers: () => {}  
}
```

)

```
export const { reset } = authSlice.actions
```

```
export default authSlice.reducer
```

File Edit Selection View Go Run Terminal Help authSlice - Task MERN - Visual Studio Code

App.js authSlice.js

frontend > gc > features > auth > authSlice.js [0] default

```
13 export const authSlice = createSlice({
14   name: 'auth',
15   initialState,
16   reducers: {
17     reset: state => {
18       state.isLoading = false
19       state.isSuccess = false
20       state.isError = false
21       state.message = ''
22     },
23   },
24   extraReducers: () => {}
25 })
26
27 export const { reset } = authSlice.actions
28 export default authSlice.reducer
29
```

Figure 7.2: Adding reducers in the authSlice.js file

Now, we will import the reducer in the store.js file. Because of the store in a Redux app, we are able to use the global state in different components.

```
import authReducer from './features/auth/authSlice'
```

```
export const store = configureStore({  
  reducer: {  
    auth: authReducer  
  }  
});
```

A screenshot of the Visual Studio Code interface. The title bar shows "store.js - Task-NTM - Visual Studio Code". The left sidebar has three tabs: "App.js M", "authSlice.js U", and "store.js M X". The main editor area displays the following code:

```
1 import { configureStore } from '@reduxjs/toolkit';
2 import authReducer from '../features/auth/authSlice'
3
4 export const store = configureStore({
5   reducer: {
6     auth: authReducer
7   }
8 });
9
```

The code uses syntax highlighting with colors for different language constructs like keywords, comments, and strings. The file path "frontend/src/app/store.js" is shown above the editor. The status bar at the bottom indicates "1 file, 100%".

Figure 7.3: Adding authReducer in store.js file

Now, we will be using the famous Chrome/Firefox extension of Redux Devtools in our project. Just go to the link and install it in the Chrome/Firefox browser. Since we have already added it, we are getting the Remove from Chrome button. Or else we will get Add to Chrome button if the extension is not installed.

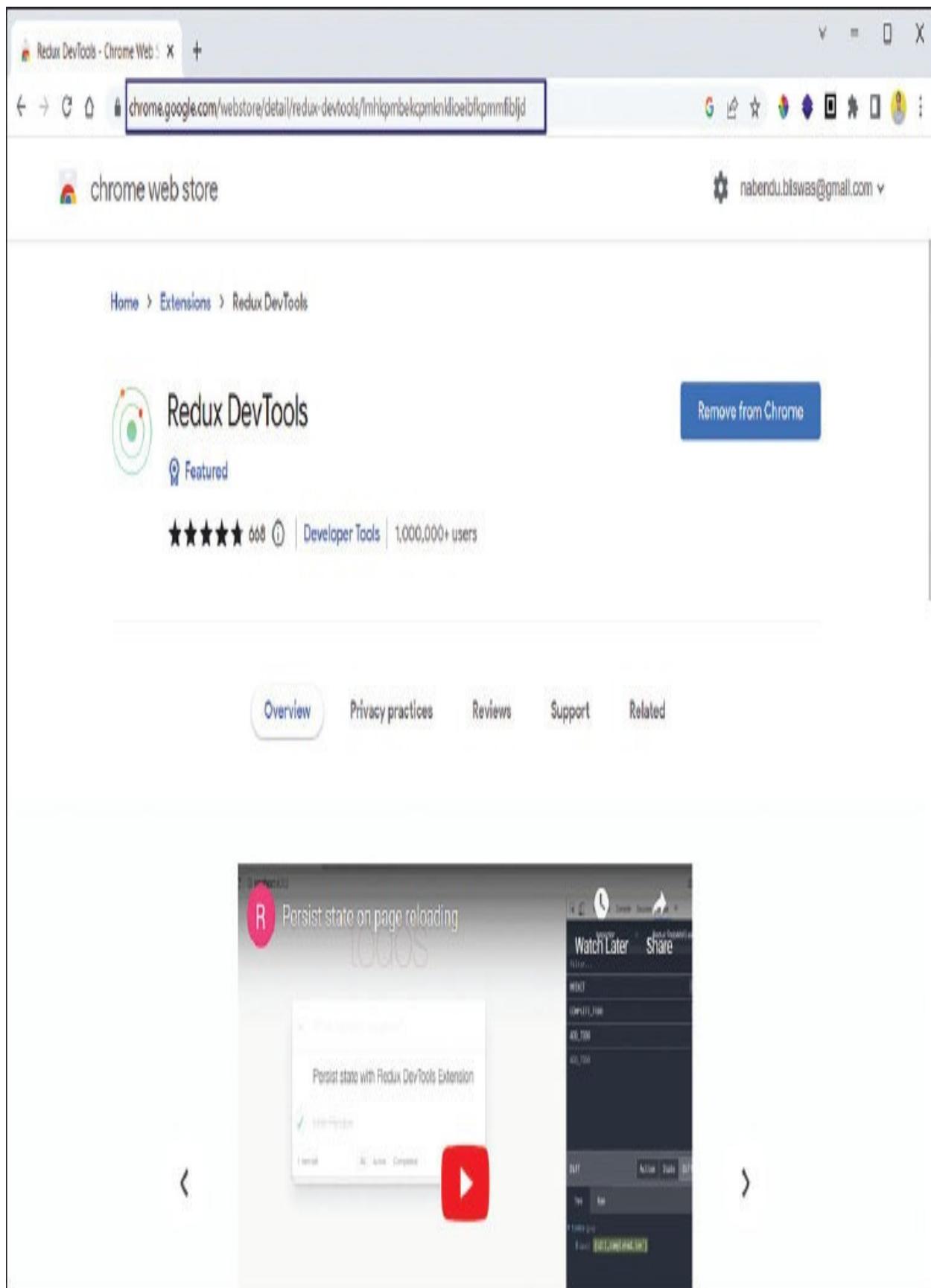


Figure 7.4: Adding Redux Devtools in Browser

In the Redux toolkit, we don't need to do any special configurations for this extension. Earlier in redux projects, we needed to install packages and add code for this extension to work. But the redux toolkit comes in-built with it, and no special configuration is required.

So, open the developer tool, and we will see a Redux tab. Here, we will see all of our states for auth. This extension is used in almost all production projects since the Redux states can sometimes be difficult to track.



Figure 7.5: Redux Devtools in Browser

Auth service and slice

We will create an auth service in this part and also will complete the auth slice started in the previous part. But first, we will install the package of axios and react-toastify in our front end.

The package of axios will be used to do API calls and react-toastify will be used to show toast messages in case of error.

So, we will add both by opening the integrated terminal and using the following npm command.

```
npm i axios react-toastify
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
```

The screenshot shows a terminal window with a dark background and light-colored text. At the top, there are tabs labeled 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. On the right side of the terminal window, there are icons for 'Search', 'File', 'Edit', 'Terminal', and 'Help'. The main area of the terminal contains the following text:

```
[user@localhost ~] % cd /Desktop/Task-15/react-project
$ npm i axios react-toastify
```

added 5 packages, and audited 159 packages in 7s

25 packages are looking for funding
run 'npm fund' for details

6 high severity vulnerabilities

To address all issues (including breaking changes), run:

Figure 7.6: Adding axios and toastify

Next, we will create the auth service. This service will be used in doing the API call and also saving the result in local storage. So, create a file authService.js inside the auth folder.

Here, we are first importing axios and also creating a variable called API_URL which is set to /api/users/. Next, we have created a register function which is taking the userData. Inside the function, we are doing a POST call to the API_URL with the userData.

The result of the api call, which we get in response.data, is set in the local storage variable user using localStorage.setItem() function. We are also returning the response from the function.

Lastly, we are exporting the register function.

```
import axios from 'axios'

const API_URL = '/api/users/'

const register = async (userData) => {
  const response = await axios.post(API_URL, userData)
  if (response.data) {
    localStorage.setItem('user', JSON.stringify(response.data))
  }
}

export { register }
```

```
}
```

```
return response.data
```

```
}
```

```
const authService = { register }
```

```
export default authService
```

File Edit Selection View Go Run Terminal Help authService.js - Task: MERN - Visual Studio Code

EXPLORER ⌂ App.js M authSlice.js U authService.js U X ⌂ ...

TASK-MERN frontend > src > features > auth > authService.js (0) authService

backend 1 import axios from 'axios'

frontend 2 const API_URL = '/api/users/'

node_modules 3

public 4 const register = async (userData) => {

SRC 5 const response = await axios.post(API_URL, userData)

app 6 if (response.data) {

storejs 7 localStorage.setItem('user', JSON.stringify(response.data))

components 8 }

features/auth 9 return response.data

authService.js 10 }

App.js 11

App.test.js 12 const authService = { register }

index.css 13

index.js 14 export default authService

setupTests.js

gitignore

package-lock.json

package.json

README.md

Figure 7.7: Creating authService.js file

Back in the authSlice.js file, we will first import the authService. After that, we export a variable register. Here, we are using the createAsyncThunk function. Again Redux toolkit comes built with a redux thunk, which is a must to do api calls through redux. In the earlier redux versions without a toolkit, we have to manually install the package and do the installation.

Here, the first parameter is the auth/register, which means from the auth folder, take the register function. Next, the parameter is an async arrow function which takes the user and thunkAPI as a parameter.

Inside the function, we have a try...catch block. From the try block, we are calling the register function from the authService.js file with the user and returning it. In the catch part, we are simply catching different kinds of errors and returning them.

```
import authService from './authService'

...
export const register = createAsyncThunk(
  'auth/register',
  async (user, thunkAPI) => {
    try {
```

```
return await authService.register(user)

} catch (error) {

const message = (error.response && error.response.data &&
error.response.data.message) ||

error.message ||

error.toString()

return thunkAPI.rejectWithValue(message)

}

}

)
```

A screenshot of the Visual Studio Code interface showing the file `authSlice.js`. The code implements a Redux Toolkit slice for user registration. It imports `createSlice` and `createAsyncThunk` from `@reduxjs/toolkit`, and `authService` from the local module. It defines an initial state object and a `register` thunk action that uses `authService.register` to handle the user's registration. If an error occurs, it extracts a message from the error object or its `toString` representation and rejects the thunk API with that message.

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
import authService from './authService'

const localUser = JSON.parse(localStorage.getItem('user'))

const initialState = {
    user: localUser,
    status: 'idle',
    error: null
}

export const register = createAsyncThunk(
    'auth/register',
    async (user, thunkAPI) => {
        try {
            return await authService.register(user)
        } catch (error) {
            const message = (error.response && error.response.data && error.response.data.message) ||
                error.message ||
                error.toString()
            return thunkAPI.rejectWithValue(message)
        }
    }
)

export const authSlice = createSlice({
```

Figure 7.8: Adding thunk in authSlice.js file

The api endpoint from the file authService.js should go to the backend endpoint of `http://localhost:8000` and not to the frontend endpoint of `http://localhost:3000`. So, we need to add a proxy variable in the package.json file in the frontend folder.

`“proxy”: “http://localhost:8000”,`

So, now when we call `axios.post()` inside the register function in the authService.js file, we will go to `http://localhost:8000/api/users`

File Edit Selection View Go Run Terminal Help package.json - Visual Studio Code

package.json M X

frontend > package.json > dependencies

```
1 {
  2   "name": "frontend",
  3   "version": "0.1.0",
  4   "proxy": "http://localhost:8000",
  5   "private": true,
  6   "dependencies": [
  7     "@reduxjs/toolkit": "^1.9.5",
  8     "@testing-library/jest-dom": "^5.16.5",
  9     "@testing-library/react": "^13.4.0",
  10    "@testing-library/user-event": "^14.4.3",
  11    "axios": "^1.4.0",
  12    "react": "^18.2.0",
  13    "react-dom": "^18.2.0",
  14    "react-icons": "^4.9.0",
  15    "react-redux": "^8.1.0",
  16    "react-router-dom": "^6.13.0",
  17    "react-scripts": "5.0.1",
  18    "react-toastify": "^9.1.3",
  19    "web-vitals": "^2.1.4"
```

Figure 7.9: Adding proxy in the package.json file

Inside our authSlice.js file in the createSlice function, we will add extraReducers. We generally put the reducer action, which changes the state in these extraReducers. In the earlier redux versions, we used to do these through the switch statement cases. You can read details for the same in the official redux documentation [here](#).

In our code, we have three cases for pending, fulfilled and rejected cases. Since we are doing an API call to `http://localhost:8000/api/users` through our register function, we have three states in it. When the API call starts, we have a pending state since the API calls generally take 1–2 seconds. We are making the `isLoading` state true and will also show a spinner in this state in the front end.

The next state is the fulfilled case, which will happen if our API call is successful. Here, we are making the `isLoading` as false and `isSuccess` as true and storing the payload which we receive back in the user state.

If, for some reason, the API call is not successful, we have a rejected case. Here, we are making the `isLoading` as false and `isError` as true. We are also storing the error in the message state, which we will receive in the payload—also making the user state variable null.

```
extraReducers: builder => {
  builder
    .addCase(register.pending, (state) => {
```

```
state.isLoading = true  
})  
.addCase(register.fulfilled, (state, action) => {  
state.isLoading = false  
state.isSuccess = true  
state.user = action.payload  
})  
.addCase(register.rejected, (state, action) => {  
state.isLoading = false  
state.isError = true  
state.message = action.payload  
state.user = null  
})  
}
```

File Edit Selection View Go Run Terminal Help authSlice.js Task MERN Visual Studio Code

frontend > src > features > auth > authSlice.js > authSlice > extraReducers

```
35     state.isSuccess = false
36     state.isError = false
37     state.message = ''
38   },
39 },
40 extraReducers: builder => {
41   builder
42     .addCase(register.pending, (state) => {
43       state.isLoading = true
44     })
45     .addCase(register.fulfilled, (state, action) => {
46       state.isLoading = false
47       state.isSuccess = true
48       state.user = action.payload
49     })
50     .addCase(register.rejected, (state, action) => {
51       state.isLoading = false
52       state.isError = true
53       state.message = action.payload
54       state.user = null
55     })
56   }
57 })
58 }
```

master ① 0 ▾ 0

Ln 40, Col 32 Sources: 4 UTF-8 CRLF () JavaScript Go Live

Figure 7.10: Adding extraReducers in the authSlice.js file

Registration form hook up

We will now use the slice in our Register component to register a new user. So, head over to the Register.js file and add the following imports in it.

Here, we are first importing the custom hooks of useSelector and useDispatch from react-redux. These hooks are used to get the global state of redux and also to dispatch the action creator inside a slice.

After that, we import useNavigate from react-router-dom. This custom hook is used to navigate easily to different routes. Next, we are importing toast from react-toastify, which will be used to show the toast message.

Lastly, we are importing the register and reset from authSlice, which will be used to call the register and reset action creators.

```
import { useSelector, useDispatch } from 'react-redux'  
import { useNavigate } from 'react-router-dom'  
import { toast } from 'react-toastify'  
import { register, reset } from '../features/auth/authSlice'
```

File Edit Selection View Go Run Terminal Help Register.js - Task MERN - Visual Studio Code

App.js M authSlice.js U Register.js X

frontend > src > components > Register.js > (0) Register > (0) onsubmit

```
1 import { useState } from "react"
2 import { FaUser } from 'react-icons/fa'
3 import { useSelector, useDispatch } from 'react-redux'
4 import { useNavigate } from 'react-router-dom'
5 import { toast } from 'react-toastify'
6 import { register, reset } from '../features/auth/authSlice'
7
8 const Register = () => {
9   const [formData, setFormData] = useState({ name: '', email: '', password: '', password2: '' })
10  const { name, email, password, password2 } = formData
11
12  const onChange = e => {
13    setFormData(prevState => ({
14      ...prevState,
15      [e.target.name]: e.target.value
16    }))
17  }
18}
```

Figure 7.11: Adding imports in the Register.js file

Now, to show the toast message using react-toastify, we need to do one more configuration. Here, we will first import ToastContainer from react-toastify. And will also import a corresponding css file.

We now need to add the ToastContainer component after the Routing is done, which is after the <Router> tag.

```
import { ToastContainer } from 'react-toastify'  
import 'react-toastify/dist/ReactToastify.css'  
...  
<ToastContainer />
```

File Edit Selection View Go Run Terminal Help App.js Task MERN - Visual Studio Code

frontend > src > App.js > App

```
4 import Register from './components/Register';
5 import Header from './components/Header';
6 import { ToastContainer } from 'react-toastify'
7 import 'react-toastify/dist/ReactToastify.css'
8
9 function App() {
10   return (
11     <>
12       <Router>
13         <div className='container'>
14           <Header />
15           <Routes>
16             <Route path='/' element={<Dashboard />} />
17             <Route path='/login' element={<Login />} />
18             <Route path='/register' element={<Register />} />
19           </Routes>
20         </div>
21       </Router>
22       <ToastContainer />
23     </>
24   );
25 }
26
27 export default App;
```

master 0 0 ▲ 0

Ln 23, Col 1 Spaces: 2 LF: () JavaScript Go Live

Figure 7.12: Configuring toastify in App.js file

Back in the Register.js file, we will create variables for navigate and dispatch using the useNavigate and useDispatch hooks.

Next, we will get the global state of the user, isLoading, isError, isSuccess, message using the useSelector hook. Lastly, inside the onSubmit() function, we are first checking if the password and the confirm password are different. And showing a toast message of Passwords are different in that case.

Else we are taking the name, email and password entered by the user and putting it in a userData variable and, after that dispatching it with the dispatch to the register function in the authSlice.js file.

```
const navigate = useNavigate()  
const dispatch = useDispatch()  
  
const { user, isLoading, isError, isSuccess, message } = useSelector(state =>  
state.auth)  
  
...  
  
...  
  
const onSubmit = e => {  
  e.preventDefault()
```

```
if (password !== password2) {  
    toast.error('Passwords are different')  
} else {  
    const userData = { name, email, password }  
    dispatch(register(userData))  
}  
}
```

File Edit Selection View Go Run Terminal Help Register.js · Task MERN · Visual Studio Code

frontend > src > components > Register.js > (1) Register > (1) onsubmit

```
8 const Register = () => {
9   const [formData, setFormData] = useState({ name: '', email: '', password: '', password2: '' })
10  const { name, email, password, password2 } = formData
11
12  const navigate = useNavigate()
13  const dispatch = useDispatch()
14
15  const { user, isLoading, isError, isSuccess, message } = useSelector(state => state.auth)
16
17  const onChange = e => {
18    setFormData(prevState => ({
19      ...prevState,
20      [e.target.name]: e.target.value
21    }))
22  }
23
24  const onSubmit = e => {
25    e.preventDefault()
26    if (password !== password2) {
27      toast.error('Passwords are different')
28    } else {
29      const userData = { name, email, password }
30      dispatch(register(userData))
31    }
32  }
33}
```

Figure 7.13: Using hooks in the Register.js file

Now, in the Register.js file, we will add an useEffect(). This useEffect will run when this component loads or when any of the user, isError, isSuccess, message, navigate, dispatch changes.

Inside it, we first check if the state of error is there and show a toast with the error message for that case. Next, if the state is of success or we got the user data, then navigate back to the home route.

If none of the cases is there, we are dispatching the reset to setback everything to initial state values.

```
import { useEffect, useState } from “react”
...
useEffect(() => {
  if (isError) toast.error(message)
  if (isSuccess || user) navigate(‘/’)
  dispatch(reset())
}, [user, isError, isSuccess, message, navigate, dispatch])
```

```
File Edit Selection View Go Run Terminal Help Register.js - Visual Studio Code
B App.js M B authSlice.js X B Register.js D Register
from frontend/src/components/Register.js
1 import { useEffect, useState } from "react"
2 import { FaUser } from 'react-icons/fa'
3 import { useSelector, useDispatch } from 'react-redux'
4 import { useNavigate } from 'react-router-dom'
5 import { toast } from 'react-toastify'
6 import { register, reset } from '../features/auth/authSlice'
7 import Spinner from "./Spinner"
8
9 const Register = () => {
10   const [formData, setFormData] = useState({ name: '', email: '', password: '', password2: '' })
11   const { name, email, password, password2 } = formData
12
13   const navigate = useNavigate()
14   const dispatch = useDispatch()
15
16   const { user, isLoading, isError, isSuccess, message } = useSelector(state => state.auth)
17
18   useEffect(() => {
19     if (isError) toast.error(message)
20     if (isSuccess || user) navigate('/')
21     dispatch(reset())
22   }, [user, isError, isSuccess, message, navigate, dispatch])
23
24   const onChange = e =>
```

Figure 7.14: useEffect in Register.js file

We will also show a loading spinner on our Register page when the user clicks on the Submit button. For this, we will create a Spinner component. So, create a Spinner.js file inside the components folder and add the following content to it.

Here, we are just using the already defined CSS in our index.css file, which shows the spinner icon.

```
const Spinner = () => {  
  return (  
    <div className='loadingSpinnerContainer'>  
      <div className='loadingSpinner' />  
    </div>  
  )  
}  
  
export default Spinner
```

The screenshot shows a dark-themed interface of Visual Studio Code. The top bar includes the standard menu options: File, Edit, Selection, View, Go, Run, Terminal, Help, and the current workspace name: Spinner.js - task-MRN - Visual Studio Code. On the far right of the top bar are icons for closing, minimizing, maximizing, and exiting the window.

The left side features a file tree (Explorer) with the following structure:

- > backend
- < frontend
 - > src
 - > components
 - Spinner.js
 - > features
 - auth
 - > node_modules
 - > public
 - > src
 - > app
 - > storejs
 - > components
 - Dashboard.js
 - Header.js
 - Login.js
 - Register.js
 - Spinner.js
 - > features/auth

Figure 7.15: Creating Spinner.js file

Now, we will first import the Spinner in our Register.js file.

```
import Spinner from “./Spinner”
```

```
File Edit Selection View Go Run Terminal Help Register.js Task MERN Visual Studio Code

frontend/src/components/Register.js

1 import { useEffect, useState } from 'react'
2 import { FaUser } from 'react-icons/fa'
3 import { useSelector, useDispatch } from 'react-redux'
4 import { useNavigate } from 'react-router-dom'
5 import { toast } from 'react-toastify'
6 import { register, reset } from '../features/auth/authSlice'
7 Import Spinner from './Spinner'

8
9 const Register = () => {
```

Figure 7.16: Importing Spinner Component

Now, we will update the return statement in the Register.js file. Here, we are first checking if the isLoading state is available and showing the Spinner component if that is the case. Or else we are showing the whole earlier code.

```
isLoading ? <Spinner /> : (  
<>  
<section className='heading'>  
...  
</section>  
<section className='form'>  
...  
</section>  
</>  
)
```

A screenshot of the Visual Studio Code interface showing the `Register.js` file. The file contains the following code:

```
File Edit Selection View Go Run Terminal Help Register.js - Task MERN - Visual Studio Code
App.js M authSlice.js U Register.js X Spinner.js U
frontend > src > components > Register.js > Register
38     }
39   }
40
41   return (
42     isLoading ? <Spinner /> : [
43       <>
44         <section className='heading'>...
45         </section>
46         <section className='form'>...
47         </section>
48       </>
49     ]
50   )
51 }
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71   </>
72   )
73   )
74 }
75
76 export default Register
```

The code is a functional component named `Register`. It includes logic for loading state (`isLoading`) and renders different sections based on whether it's loading or not. The code editor shows syntax highlighting and line numbers from 38 to 76.

Figure 7.17: Using Spinner Component

Testing user registration

Click on the Register button, and you will go to <http://localhost:3000/register>. We have also opened the Redux dev tools, and here we can notice that reset is called and the user state is set to the initial value.

Here, we have entered all the required fields.

The screenshot shows a web browser window with a registration form on the left and a Redux DevTools sidebar on the right.

Left Side (Registration Form):

- Page URL: `localhost:3001/register`
- Form fields:
 - First Name: `Hiday`
 - Email: `hiday@gmail.com`
 - Password (Visible): `*****`
 - Confirm Password (Visible): `*****`
- Submit button: `Submit`

Right Side (Redux DevTools):

- Toolbar: Inspector, Console, Redux, Actions, Settings.
- Actions:
 - `auth/reset`
- State Table:

Path	Type	Action	State	Diff	Trace	Test
<code>auth</code>	Object	<code>auth/reset</code>	<code>{ user: null, isError: false, isSuccess: false, isLoading: false, message: "" }</code>			
- State Preview:

```
auth (path)
  user (print null)
  isError (print) false
  isSuccess (print) false
  isLoading (print) false
  message (print) ""
```
- Panel Buttons: `auth/reset (1)`, `<`, `>`, `Live`, `1x`, `2x`.
- Bottom Toolbar: Inspector, Log monitor.

Figure 7.18: Initial states in the register

We have intentionally given the confirmed password as wrong. Here, a toast message of Passwords are different is shown, which automatically disappears after 5 seconds.

localhost:3000/register

Task Creator

Passwords are different

Actions Settings

Reset Revert Swoop Commit Reset... ▾

Diff...

auth/reset 129:41:38

auth/reset 48:23:76

State Action State Diff Trace Test

Tree Chart Raw

▼ auth (pr)

user (pr) null

isError (pr) false

isSuccess (pr) false

isPending (pr) false

message (pr) "

auth/reset ()

Submit

Inspector Log monitor

Figure 7.19: Toast message in register

We have, this time, given the same passwords in both fields and hit the Submit button. Here, we have been taken to the home page of the Dashboard. But also notice that different actions have been done in the Redux devtools.

Here, first, the pending action ran from the extraReducers in the authSlice.js file. Here, the isLoading was set to true, and the rest states were kept as it was earlier.

The screenshot shows a web browser window with the URL `localhost:3000`. The page title is "Task Creator". On the left, there's a sidebar with a "Dashboard" section. The main content area displays a list of tasks:

Task	Time
auth/reset	1:29:41.38
auth/register/pending	<0:11.49
auth/register/fullfilled	<0:00.38
auth/reset	<0:00.38
auth/reset	<0:00.38

Below the table, there are tabs for "State", "Action", "State", "Diff", "Trace", and "Test". The "State" tab is selected, showing the current state of the "auth" reducer:

```
auth (pending)
  user (null)
  failure (false)
  isSuccess (false)
  isLoading (true)
  message ()
```

The "Action" tab shows the history of actions:

```
auth/reset ()
```

At the bottom, there are navigation icons for the Redux DevTools, including arrows for step operations and a search bar.

Figure 7.20: Pending state in the register

Since the API call to the backend was successful, the fulfilled action had run. This has made the isLoading false and isSuccess as true. And the user state was set to the payload received from the backend. This payload consists of _id, name, email, and token.

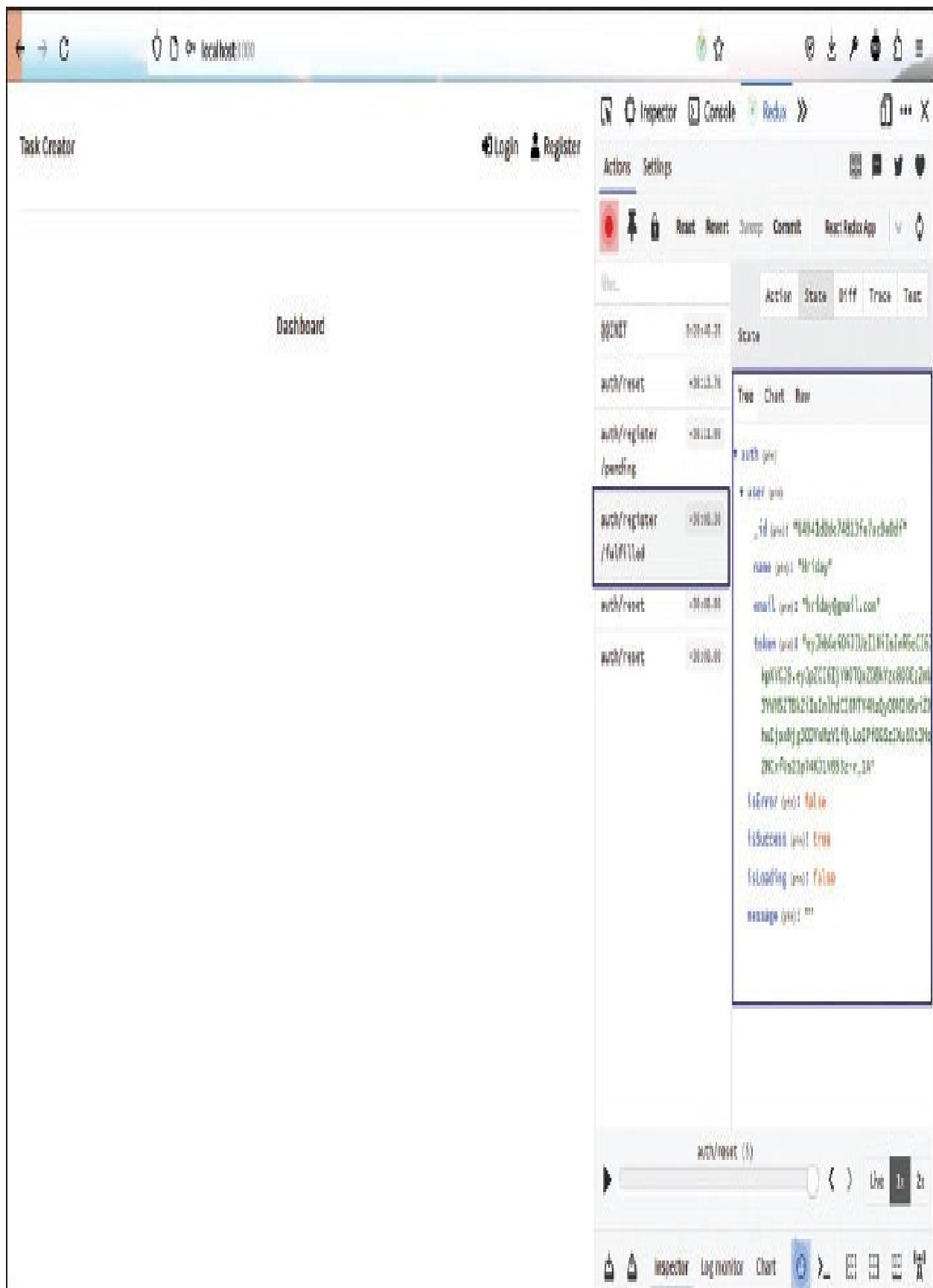


Figure 7.21: Fulfilled state in the register

The reset was run again, and it set everything back to an earlier state, but the user was kept intact.



Figure 7.22: Reset state in the register

The user data was also saved in the Local Storage of the browser, containing all the fields of the user object.

The screenshot shows a browser window with a developer tools overlay. The title bar reads "localhost:4000". The main content area says "Task Creator" and has "Login" and "Register" buttons. Below that is a "Dashboard" section. The developer tools panel is open at the bottom, specifically the "Storage" tab.

The Storage panel lists the following items:

- Cache Storage
- Cookies
- IndexedDB
- Local Storage**
- Session Storage
- http://localhost:3000

Under Local Storage, there are two entries:

- ally-app... (Key) Value: "userAgent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.140 Safari/4193" (Value)
- pusher... (Key) Value: {"timestamp":1573912808927,"transport":"xhr_polling","latency":250} (Value)

Under Session Storage, there is one entry:

- user (Key) Value: [Object object] (Value)

The "user" value is expanded to show:

- + userObject: [Object object]
- + name: "Troy"
- + email: "troy@gmail.com"
- + token: "ehtGOCUtg1HicJjQ2p74K7W652cJAJ"
- + __proto__: Object

Figure 7.23: Local Storage in Browser

Finally, we checked the value for users in the connected MongoDB database, and a new field was added to it.

System Status: All Good

©2023 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Figure 7.24: Record in Database

Conclusion

In this chapter, we have learned about implementing Redux in our project through the toolkit. Here, we first created a slice for auth, which is a combination of reducer and action creator. We had also hooked it up with auth service to help do api calls to the backend endpoint.

Finally, we have completed our Register user functionality from the front end using the slice. And we were able to Register a user by submitting a form from the front end.

In the next chapter, we will implement the login and logout functionalities.

Points to remember

Implementing the Redux toolkit in a project using slice and services.

Using a slice in the latest Redux toolkit, we combine the functionalities of reducer and action creator.

Getting global state through redux in a component. And dispatching action creators from a component.

CHAPTER 8

Login and Logout Functionalities

Introduction

This chapter covers the implementation of Logout and Login functionalities in the project. We will be going to use logic similar to the previous chapter to set both of them. In this chapter, we are also going to set up a reducer for the Login form. Finally, we are also going to test the user login workflow.

Structure

In this chapter, we are going to discuss the following topics:

Implementing Logout

Implementing Login

Login form hook up

Testing user login

Implementing Logout

In the previous chapter, we have implemented the Register functionality. Now, the logout functionality will be much simpler than it. Here, on click of a new Logout button, we have removed all tokens.

So, first, in the authSlice.js file, create a new function called logout. Similar to the register function, it uses the createAsyncThunk function. But it is much simpler than the register function of the previous chapter.

Here, the first parameter is the auth/login, which means from the auth folder, take the login function. We are going to create this function next in the authService.js file. Next, the parameter is an async arrow function calling the logout() from the authService.js file.

```
export const logout = createAsyncThunk('auth/logout', async () => await authService.logout())
```

A screenshot of the Visual Studio Code interface showing the file `authSlice.js`. The code defines two thunk functions: `register` and `logout`, and a slice object `authSlice`.

```
File Edit Selection View Go Run Terminal Help authSlice.js Task MERN Visual Studio Code

authSlice.js X

frontend > src > features > auth > authSlice.js > authSlice

12 }
13
14 export const register = createAsyncThunk(
15   'auth/register',
16   async (user, thunkAPI) => {
17     try {
18       return await authService.register(user)
19     } catch (error) {
20       const message = (error.response && error.response.data && error.response.data.message) ||
21                     error.message ||
22                     error.toString()
23       return thunkAPI.rejectWithValue(message)
24     }
25   }
26 )
27
28 export const logout = createAsyncThunk('auth/logout', async () => await authService.logout())
29
30 export const authSlice = createSlice([

]
```

Figure 8.1: Creating logout in authSlice.js file

Now, in the authService.js file, we will create our logout function. Again notice that here it's much simpler than the register function created in the earlier chapter.

Here, we are just removing the user from the local storage using the localStorage.removeItem() method. After that, we are also exporting the logout function.

```
const logout = () => localStorage.removeItem('user')
```


Figure 8.2: Creating logout in authService.js file

We don't have a Logout button in our project. This Logout button will be created in the Header component. So, head over to the Header.js file and import useNavigate from react-router-dom.

We are also importing useSelector and useDispatch from react-redux. Next, we import logout and reset from the authSlice file.

Inside the Header, we have created a navigate variable using the useNavigate hook. Then we created a dispatch variable using the useDispatch hook. And we are also accessing the global state of the user using the useSelector hook.

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

import { useSelector, useDispatch } from 'react-redux'

import { logout, reset } from '../features/auth/authSlice'

...

const navigate = useNavigate()

const dispatch = useDispatch()

const { user } = useSelector(state => state.auth)
```

File Edit Selection View Go Run Terminal Help Header.js - Task MERN - Visual Studio Code

autShops.js autServices.js Header.js X

frontend > src > components > Header.js [0] Header

```
1 import { FaSignInAlt, FaSignOutAlt, FaUser } from 'react-icons/fa'
2 import { Link, useNavigate } from 'react-router-dom'
3 import { useSelector, useDispatch } from 'react-redux'
4 import { logout, reset } from '../features/auth/authSlice'

5
6 const Header = () => {
7   const navigate = useNavigate()
8   const dispatch = useDispatch()
9   const { user } = useSelector(state => state.auth)

10
11   return (
12     <header className='header'>
13       <div className="logo">
14         <Link to='/Task Creator'>Task Creator</Link>
15       </div>
16     </header>
17   )
18 }

19 export default Header
```

Figure 8.3: Adding imports in the Header.js file

We will update our logic to show buttons in the Header.js file. Earlier, we were just showing the Login and Register buttons inside the through two

But now we have updated the logic and are using a ternary operator. Earlier we were only showing the pair or Login and Register button. But now if user is available, we are showing the Logout button. Or else the Login and Register buttons.

```
{user ? (  
  <li>  
    <button className=>btn>  
      <FaSignOutAlt /> Logout  
    </button>  
  </li>  
) : (  
  <>  
  <li>  
    <Link to=>/login><FaSignInAlt /> Login</Link>  
  </li>  
  <li>
```

<Link to=>/register><FaUser /> Register</Link>

</>

)}

File Edit Selection View Go Run ... Header.js - Task-MERN - Visual Studio Code

authSlice.js U authService.js U Header.js X

frontend > src > components > Header.js > Header

```
12 <header className='header'>
13   <div className="logo">
14     <Link to='/'>Task Creator</Link>
15   </div>
16   <ul>
17     {user ? (
18       <li>
19         <button className='btn'>
20           <FaSignOutAlt /> Logout
21         </button>
22       </li>
23     ) : (
24       <>
25         <li>
26           <Link to='/login'><FaSignInAlt /> Login</Link>
27         </li>
28         <li>
29           <Link to='/register'><FaUser /> Register</Link>
30         </li>
31       </>
32     )}
33   </ul>
34 </header>
35 )
```

Figure 8.4: Adding Logout button in Header.js file

Now, we will add an onClick handler to the Logout button in the Header.js file. The onClick handler is calling a function logoutFn.

We have created the logoutFn function, which is first dispatching the logout() from the authSlice.js file. Next, it dispatches the reset() from the authSlice.js file. Afterwards, we navigate back to the home route using navigate().

```
const logoutFn = () => {
  dispatch(logout())
  dispatch(reset())
  navigate(</>)
}

...
...

{user ? (
<li>
  <button className=>btn</button> onClick={logoutFn}>
    <FaSignOutAlt /> Logout
  </button>
)
```


) : (

<>

...

</>

)}

Header.js · task-MERN · Visual Studio Code

authSlice.js U authService.js U Header.js U X

frontend > src > components > Header.js > Header > logoutFn

```
const navigate = useNavigate()
const dispatch = useDispatch()
const { user } = useSelector(state => state.auth)

const logoutFn = () => {
    dispatch(logout())
    dispatch(reset())
    navigate('/')
}

return (
    <header className='header'>
        <div className="logo">
            <Link to='/'>Task Creator</Link>
        </div>
        <ul>
            {user ? (
                <li>
                    <button className='btn' onClick={logoutFn}>
                        <FaSignOutAlt /> Logout
                    </button>
                </li>
            ) : (
                <></gt>
            )}
        </ul>
    </header>
)
```

Figure 8.5: Adding Logout function in Header.js file

As in the case of register, we need to add a case in the extraReducers in the authSlice.js file. In the register, we had three cases because we were doing the API call.

But here, we have only one case of fulfilled in which we are making the user state null.

```
.addCase(logout.fulfilled, (state) => {  
  state.user = null  
})
```

File Edit Selection View Go Run Terminal Help authSlice Link MERN Visual Studio Code

authSlice.js X authService.js U Headers U

frontend > src > features > auth > **authSlice.js** > authSlice > extraReducers

```
38     state.message = '';
39   },
40 },
41 extraReducers: builder => {
42   builder
43     .addCase(register.pending, (state) => {
44       state.isLoading = true
45     })
46     .addCase(register.fulfilled, (state, action) => {
47       state.isLoading = false
48       state.isSuccess = true
49       state.user = action.payload
50     })
51     .addCase(register.rejected, (state, action) => {
52       state.isLoading = false
53       state.isError = true
54       state.message = action.payload
55       state.user = null
56     })
57     .addCase(logout.fulfilled, (state) => {
58       state.user = null
59     })
60   }
61 }
```

In 50, Col 15 Spaces 4 UTF-8 CRLF {} JavaScript ⚡ Go live ⌂ ⌂

Figure 8.6: Adding logout case function in authSlice.js file

Now, go to <http://localhost:3000/> from the Chrome/Firefox browser and open the redux devtools. Log in to the app if not already done.

If logged in, we will see the Logout button. And also, in Redux Devtools, we will see the INIT state with the user and other states.

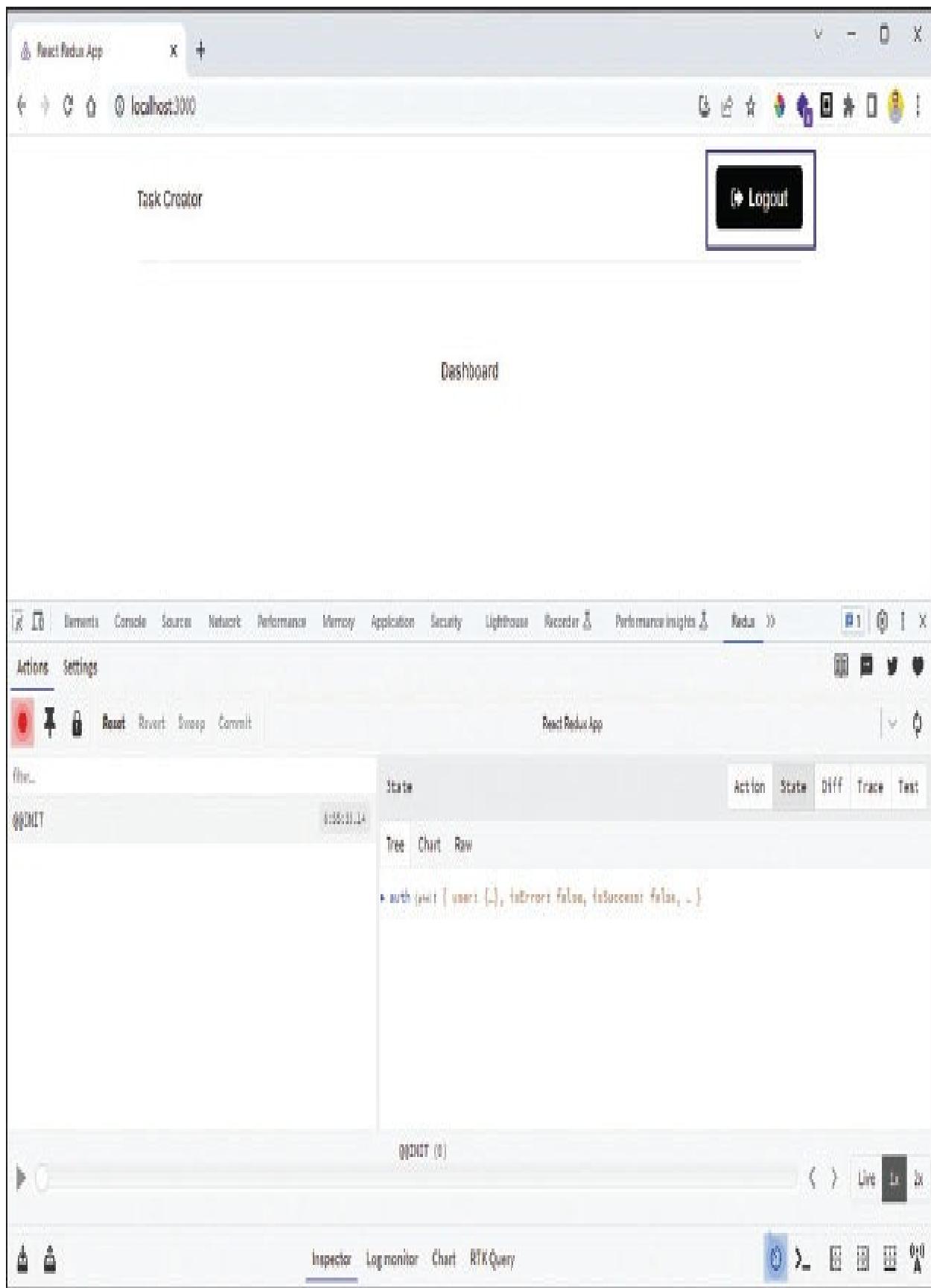


Figure 8.7: Logout button in localhost

Once we click on the Logout button in the preceding screenshot, we see three states have been changed. We have a logout/pending, reset and finally, the logout/fulfilled states.

Since, in the Header.js in logoutFn() we are also calling the reset() from the authSlice.js file, these two additional states are being called.

We can see in the fulfilled state that the user is set to null.

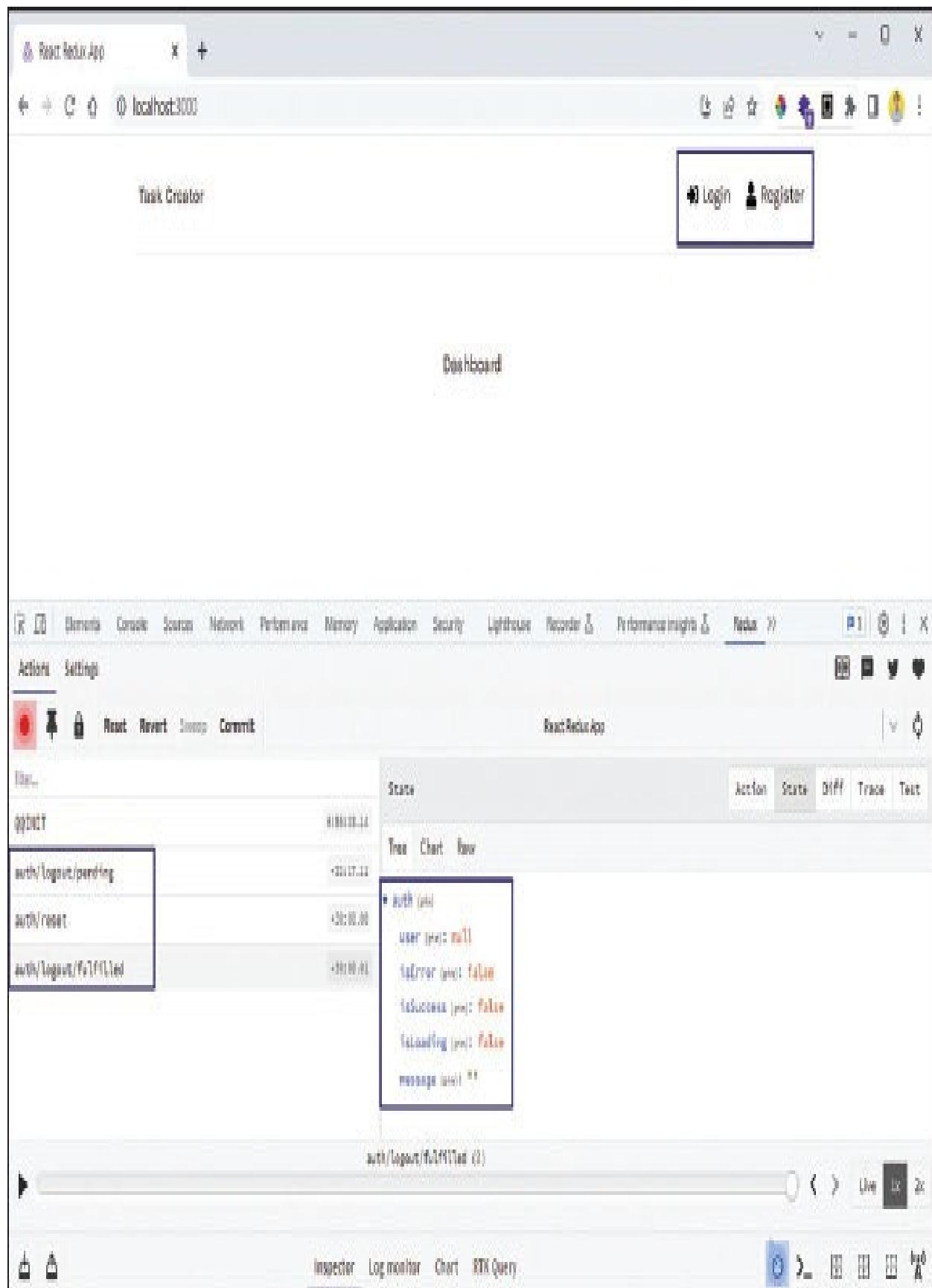


Figure 8.8: On clicking the Logout button in localhost

When they Logout, the user's data is also deleted from the local storage. To check the same, Login again and go to the Application tab in Chrome or Storage tab in Firefox.

Here, click on Local Storage and then <http://localhost:3000/>. Here, we will see the user data being stored in local storage.

The screenshot shows a browser window with a tab labeled "React Native App". The address bar indicates the page is at "localhost:1000". The main content area displays a "Task Creator" interface with a "Logout" button. Below this, a "Dashboard" section is visible. At the bottom of the screen, the developer tools are open, specifically the Application tab. The Application tab shows storage details for the URL "http://localhost:1000". It lists a single item with the following data:

```
[{"id": "60701afff9c7ed3cbf8271", "name": "Abend", "mail": "abend@gmail.com", "role": "Abend", "token": "eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ0ZWxkYmFzZS5jb20iLCJ0eXAiOiJKV1QiLCJ4bWxuczI6Imh0dHA6Ly9leGFtcGxlLmNvbS9yZWxkYmFzZS5jb20iLCJ4bWxucyI6Imh0dHA6Ly9leGFtcGxlLmNvbS9yZWxkYmFzZS5jb20iLCJpYXQiOjE2MjQwOTUyNjMsImV4cCI6MTYyNDUwOTI2M30.", "url": "http://localhost:1000"}]
```

Figure 8.9: Login in localhost showing the user in local storage

Now, click on the Logout button, and we will see that the user data has been deleted from the Local Storage.

This happens because when we click on the LocalStorage button, then the function `localStorage.removeItem('user')` in `authService.js` is executed. And this removes the user from local storage.

The screenshot shows a browser window with a developer tools overlay. The title bar of the browser says "React Redux App". The developer tools toolbar includes icons for back, forward, search, and refresh, along with a color palette and other developer options.

The main content area displays a "Task Creator" page with a "Dashboard" section. At the top right, there are "Login" and "Register" buttons. Below the dashboard, there is a large, empty space.

The bottom portion of the image shows the "Application" tab of the developer tools Network panel. The sidebar lists "Market", "Service Workers", and "Storage". Under "Storage", "LocalStorage" is selected, and a list of items is shown, with one item highlighted: "http://localhost:3001". Other items listed include "Session Storage", "IndexedDB", "Web SQL", "Cookies", "Private State Tokens", "InMemory", "SharedStorage", and "Cache Storage". A message "Select a value to preview" is displayed at the bottom of the storage list.

Figure 8.10: Logout in localhost removing user in local storage

Implementing Login

Now, we will implement the Login functionality. This is quite similar to the Register functionality in the previous chapter.

Back in the authSlice.js file, we are exporting a variable login. Here, we are using the createAsyncThunk function.

Here, the first parameter is the auth/login, which means from the auth folder, take the login function. Next, the parameter is an async arrow function which takes the user and thunkAPI as a parameter.

Inside the function, we have a try...catch block. From the try block, we are calling the login function from the authService.js file with the user and returning it. In the catch part, we are simply catching different kinds of errors and returning them.

```
export const login = createAsyncThunk('auth/login', async (user, thunkAPI) =>
{
  try {
    return await authService.login(user)
  } catch (error) {
    const message =
      (error.response && error.response.data && error.response.data.message) ||
```

```
error.message ||  
error.toString()  
return thunkAPI.rejectWithValue(message)  
}  
})
```

A screenshot of the Visual Studio Code interface. The top bar shows the menu (File, Edit, Selection, View, Go, Run, Terminal, Help) and the workspace name (authSlice.js · Task-MERN · Visual Studio Code). The status bar on the right includes icons for document, search, and refresh, along with a percentage (100%) and a close button.

The left sidebar displays a file tree with the following structure:

```
frontend/sic/features/auth/authSlice.js
```

The main code editor area contains the following TypeScript code:

```
28 export const login = createAsyncThunk('auth/login', async (user, thunkAPI) => {
29   try {
30     return await authService.login(user)
31   } catch (error) {
32     const message =
33       (error.response || error.response.data || error.response.message) ||
34       error.message ||
35       error.toString()
36     return thunkAPI.rejectWithValue(message)
37   }
38 })
39
40 export const logout = createAsyncThunk('auth/logout', async () => await authService.logout())
41
42 export const authSlice = createSlice({
43   name: 'auth'
```

Figure 8.11: Adding thunk in authSlice.js file

Next, we have created a login function in the authService.js file, which takes the userData. Inside the function, we are doing a POST call to the API_URL plus login with the userData.

The result of the api call, which we get in response.data, is set in the local storage variable user using localStorage.setItem() function. We are also returning the response from the function.

Lastly, we are exporting the login function.

```
const login = async (userData) => {  
  const response = await axios.post(API_URL + 'login', userData)  
  if (response.data) {  
    localStorage.setItem('user', JSON.stringify(response.data))  
  }  
  return response.data  
}
```

File Edit Selection View Go Run Terminal Help authService.js - Task-Monitor - Visual Studio Code

authSlice.js U authService.js U Header.js U

fronted > src > features > auth > authService.js > register

```
1 import axios from 'axios'
2 const API_URL = '/api/users/'
3
4 const register = async (userData) => {
5   const response = await axios.post(API_URL, userData)
6   if (response.data) {
7     localStorage.setItem('user', JSON.stringify(response.data))
8   }
9   return response.data
10 }
11
12 const login = async (userData) => {
13   const response = await axios.post(API_URL + 'login', userData)
14   if (response.data) {
15     localStorage.setItem('user', JSON.stringify(response.data))
16   }
17   return response.data
18 }
19
20
21 const logout = () => localStorage.removeItem('user')
22
23 const authService = { register, logout, login }
```

In 8, Col 4 Spaces: 2 UTF-8 CR LF {} JavaScript Go Live

Figure 8.12: Adding login functionalities in the authService.js file

Next, similar to the register, we will add three cases in the authSlice.js file. In our code, we have three cases for pending, fulfilled and rejected cases. Since we are doing an API call to `http://localhost:8000/api/users/login` through our login function, we have three states in it. When the API call starts, we have a pending state since the API calls generally take 1–2 seconds. Again, we are making the `isLoading` state to true and will also show a spinner in this state in the front end.

The next state is the fulfilled case, which will happen if our API call is successful. Here, we are making the `isLoading` as false and `isSuccess` as true and storing the payload which we receive back in the user state.

If, for some reason, the API call is not successful, we have a rejected case. Here, we are making the `isLoading` as false and `isError` as true. We are also storing the error in the message state, which we will receive in the payload—also making the user state variable null.

```
.addCase(login.pending, (state) => {  
  state.isLoading = true  
}  
  
.addCase(login.fulfilled, (state, action) => {  
  state.isLoading = false  
  state.isSuccess = true  
  state.user = action.payload
```

})

.addCase(login.rejected, (state, action) => {

state.isLoading = false

state.isError = true

state.message = action.payload

state.user = null

})

File Edit Selection View Go Run Terminal Help authSlice.js - Task MTBN - Visual Studio Code

authSlice.js U authService.js U Headers.js U

frontend > src > features > auth > authSlice.js (W authSlice) (D extraReducers) (G addCase) callback

```
63     .addCase(register.rejected, (state, action) => {
64         state.isLoading = false
65         state.isError = true
66         state.message = action.payload
67         state.user = null
68     })
69     .addCase(login.pending, (state) => {
70         state.isLoading = true
71     })
72     .addCase(login.fulfilled, (state, action) => {
73         state.isLoading = false
74         state.isSuccess = true
75         state.user = action.payload
76     })
77     .addCase(login.rejected, (state, action) => {
78         state.isLoading = false
79         state.isError = true
80         state.message = action.payload
81         state.user = null
82     })
83     .addCase(logout.fulfilled, (state) => {
84         state.user = null
85     })
86
```

In 69 (Col 49) Source | UTF-8 | DNU | () JavaScript | Go To Line | R | Q

Figure 8.13: Adding cases for login in the authSlice.js file

Login form hook up

We will now use the Slice in our Login component to log in a user. So, head over to the Login.js file and add the following imports in it.

Here, we are first importing the useEffect hook custom, which will be used to dispatch the reset function soon. Next, we are importing hooks of useSelector and useDispatch from react-redux. These hooks are used to get the global state of redux and also to dispatch the action creator inside a slice.

After that, we import useNavigate from react-router-dom. This custom hook is used to navigate easily to different routes. Next, we are importing toast from react-toastify, which will be used to show the toast message.

Next, we are importing login and reset from authSlice, which will be used to call the login and reset action creators. Lastly, we are importing the Spinner component, which will be used to show the Spinner when the API call starts.

Next, we will create variables for navigate and dispatch using the useNavigate and useDispatch hooks.

Next, we will get the global state of the user, isLoading, isError, isSuccess, message using the useSelector hook.

```
import { useState, useEffect } from “react”
```

```
import { useSelector, useDispatch } from 'react-redux'
import { useNavigate } from 'react-router-dom'
import { toast } from 'react-toastify'
import { login, reset } from '../features/auth/authSlice'
import Spinner from "./Spinner"
...
const navigate = useNavigate()
const dispatch = useDispatch()

const { user, isLoading, isError, isSuccess, message } = useSelector(state =>
state.auth)
```

```
File Edit Selection View Go Run Terminal Help Login.js - Task-MERN - Visual Studio Code
authSlice.js U authService.js U Login.js X Register.js U
frontend > src > components > Login.js > Login
1 import { FaSignInAlt } from 'react-icons/fa'
2 import { useState, useEffect } from "react"
3 import { useSelector, useDispatch } from 'react-redux'
4 import { useNavigate } from 'react-router-dom'
5 import { toast } from 'react-toastify'
6 import { login, reset } from '../features/auth/authSlice'
7 import Spinner from "./Spinner"
8
9 const Login = () => [
10   const [formData, setFormData] = useState({ email: '', password: '' })
11   const { email, password } = formData
12   const navigate = useNavigate()
13   const dispatch = useDispatch()
14
15   const { user, isLoading, isError, isSuccess, message } = useSelector(state => state.auth)
16
17   const onChange = e => {
18     setFormData(prevState => ({
19       ...prevState,
20       [e.target.name]: e.target.value
21     }))
22 }
```

Figure 8.14: Adding imports in the Login.js file

Now, in the Login.js file, we will add an useEffect(). This useEffect will run when this component loads or when any of the user, isError, isSuccess, message, navigate, dispatch changes.

Inside it, we are first checking if the state of error is there and show a toast with the error message for that case. Next, if the state is of success or we got the user data, then navigate back to the home route.

If none of the cases is there, we are dispatching the reset to setback everything to initial state values.

```
File Edit Selection View Go Run Terminal Help Login.js - Test-MERN - Visual Studio Code
autSlice.js authService.js Login.js Register.js

frontend > src > components > Login.js > onLogin > onChange > setFormData callback

14
15 const { user, isLoading, isError, isSuccess, message } = useSelector(state => state.auth)
16
17 useEffect(() => {
18   if (isError) toast.error(message)
19   if (isSuccess || user) navigate('/')
20   dispatch(reset())
21 }, [user, isError, isSuccess, message, navigate, dispatch])
22
23 const onChange = e => [
24   setFormData(prevState => [
25     ...prevState,
26     [e.target.name]: e.target.value
27   ])
28 ]
29
30 const onSubmit = e => [
31   e.preventDefault()
32 ]
```

Figure 8.15: Adding useEffect in Login.js file

Now in the Login.js file, inside the onSubmit() function, we are taking the email and password entered by the user and putting it in a userData variable and, after that dispatching it with the dispatch to the login function in the authSlice.js file.

```
const onSubmit = e => {  
  e.preventDefault()  
  
  const userData = { email, password }  
  
  dispatch(login(userData))  
}
```

File Edit Selection View Go Run Terminal Help Login Task MERN Visual Studio Code

authSlice.js | authService.js | Login.js X Register.js

frontend/src/components/Login.js (18/18) onSubmit

```
23 const onChange = e => {
24   setFormData(prevState => ({
25     ...prevState,
26     [e.target.name]: e.target.value
27   }));
28 }
29
30 const onSubmit = e => {
31   e.preventDefault()
32   const userData = { email, password }
33   dispatch(login(userData))
34 }
35
36 return (
37   <>
38   <section className='heading'>
39     <h1>Login</h1>
```

Figure 8.16: Adding submit logic in the Login.js file

Now, we will update the return statement in the Login.js file. Here, we are first checking if the isLoading state is available and showing the Spinner component if that is the case. Or else we are showing the whole earlier code.

```
isLoading ? <Spinner /> : (  
<>  
<section className='heading'>  
...  
</section>  
<section className='form'>  
...  
</section>  
</>  
)
```

File Edit Selection View Go Run Terminal Help Login.js - Task-MERN - Visual Studio Code

JS authSlice.js U JS authService.js U JS Login.js U JS Register.js U

frontend > src > components > JS Login.js [0] Login

```
36 return (
37   isLoading ? <Spinner /> : (
38     <>
39       <section className='heading'>
40         <h1><FaSignInAlt /> Login</h1>
41         <p>Login and start creating tasks</p>
42       </section>
43       <section className='form'>
44         <form onSubmit={onSubmit}>
45           <div className='form-group'>
46             <input type='email' className='form-control' id='email' name='email' value={email} placeholder='Enter your email' onChange={onChange} />
47           </div>
48           <div className='form-group'>
49             <input type='password' className='form-control' id='password' name='password' value={password} placeholder='Enter password' onChange={onChange} />
50           </div>
51           <div className='form-group'>
52             <button type='submit' className='btn btn-block'>Submit</button>
53           </div>
54         </form>
55       </section>
56     </>
57   )
)
```

1657 Col 1 Source 2 INT-8 CRLF {} JavaScript ⌂ Go live ⌂ ⌂

Figure 8.17: Adding Spinner in Login.js file

Testing user login

Click on the Login button, and you will go to. We have also opened the Redux dev tools, and here we can notice that reset is called and the user state is set to the initial value.

Here, we have entered all the required fields.

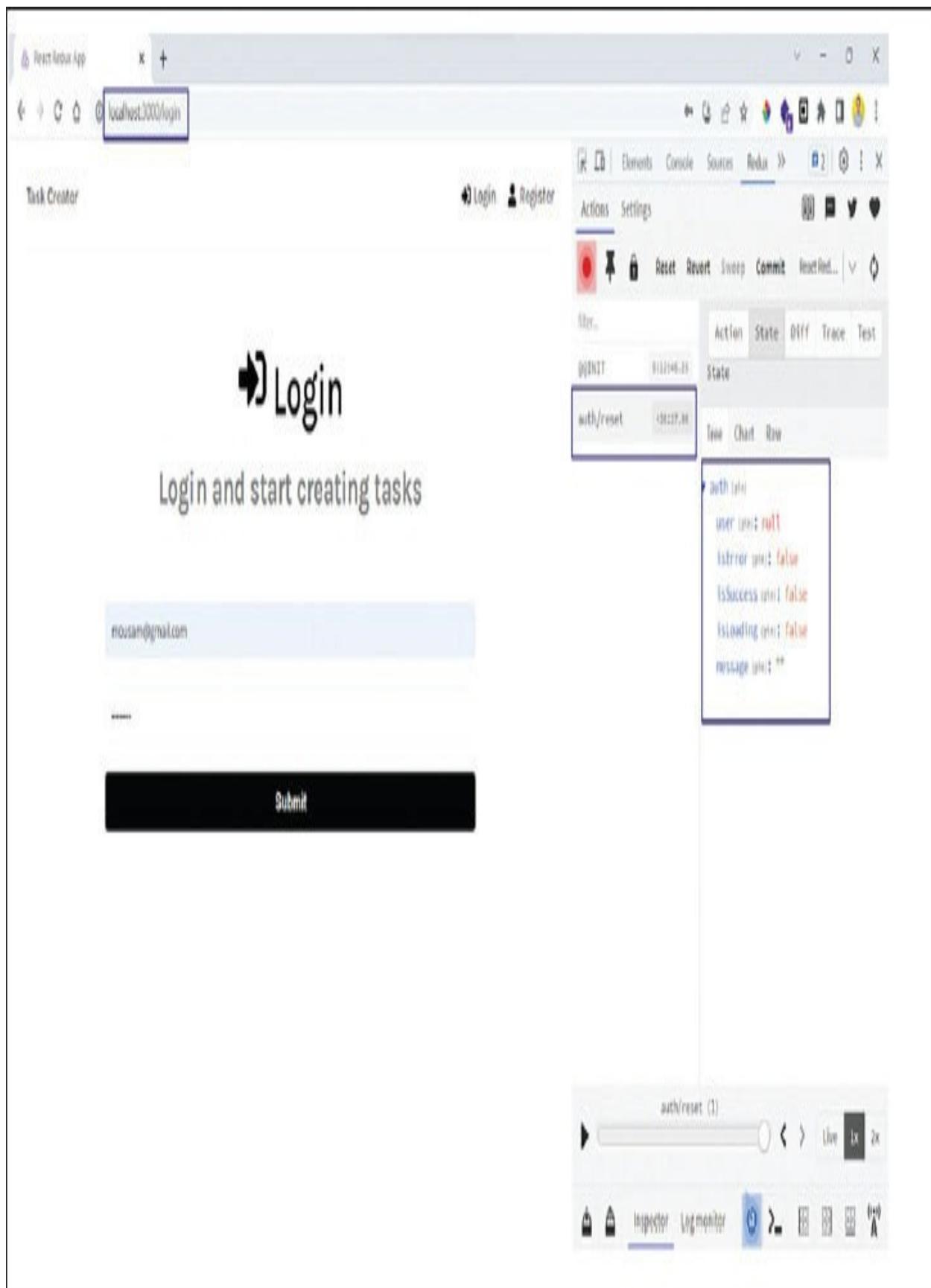


Figure 8.18: Initial states in the login

We have intentionally given the password as wrong. Here, a toast message of Invalid data is shown, which automatically disappears after 5 seconds.

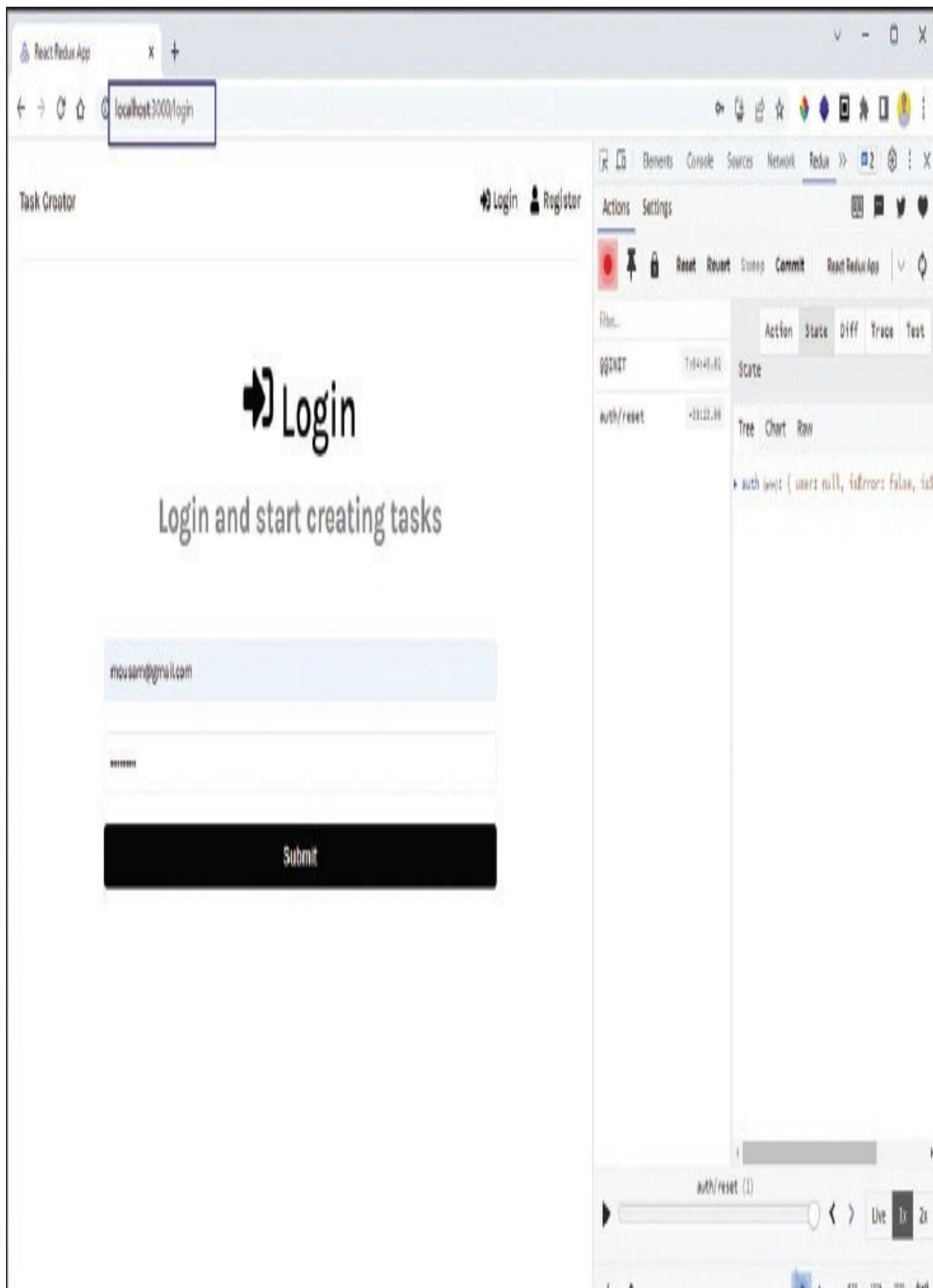


Figure 8.19: Toast message in the login

We have, this time, given the correct password and hit the Submit button. Here, we have been taken to the home page of the dashboard. But also notice that different actions have been done in the Redux devtools.

Here, first, the pending action ran from the extraReducers in the authSlice.js file. Here, the isLoading was set to true, and the rest states were kept as it was earlier.

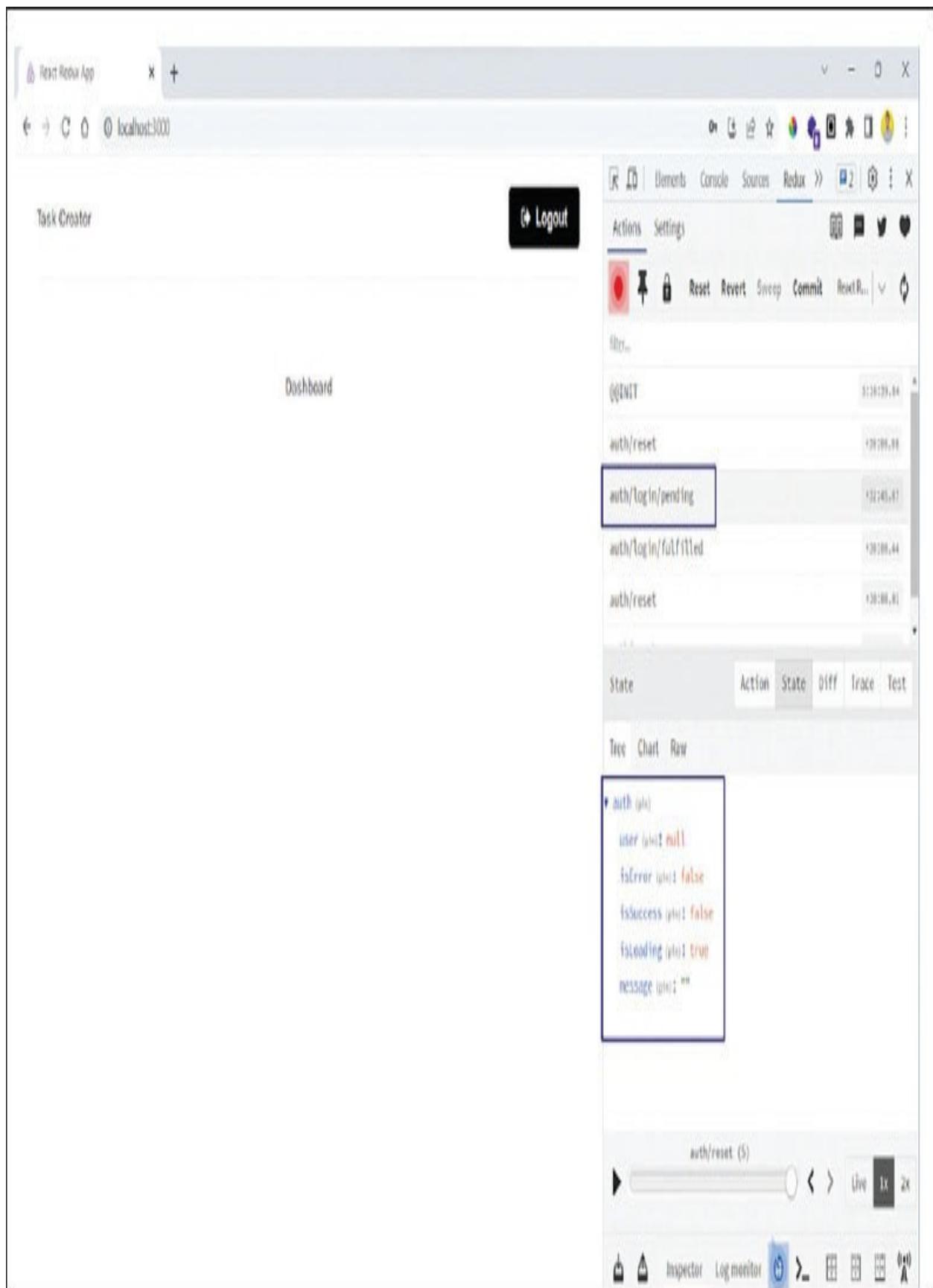


Figure 8.20: Pending state in the login

Since the API call to the backend was successful, the fulfilled action had run. This has made the isLoading false and isSuccess as true. And the user state was set to the payload received from the backend. This payload consists of _id, name, email and token.

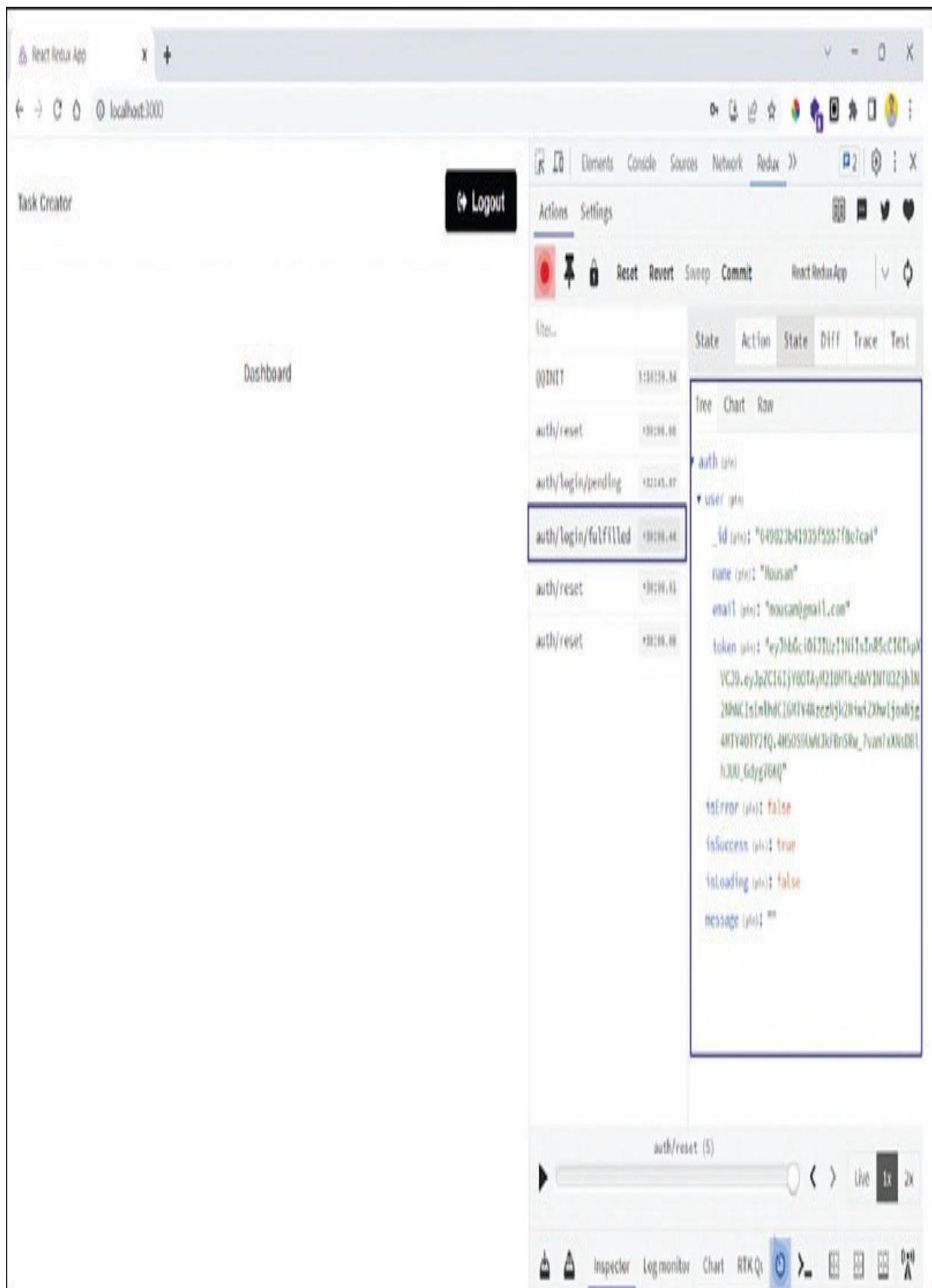


Figure 8.21: Fulfilled state in the login

The reset was run again, and it set everything back to an earlier state, but the user was kept intact.

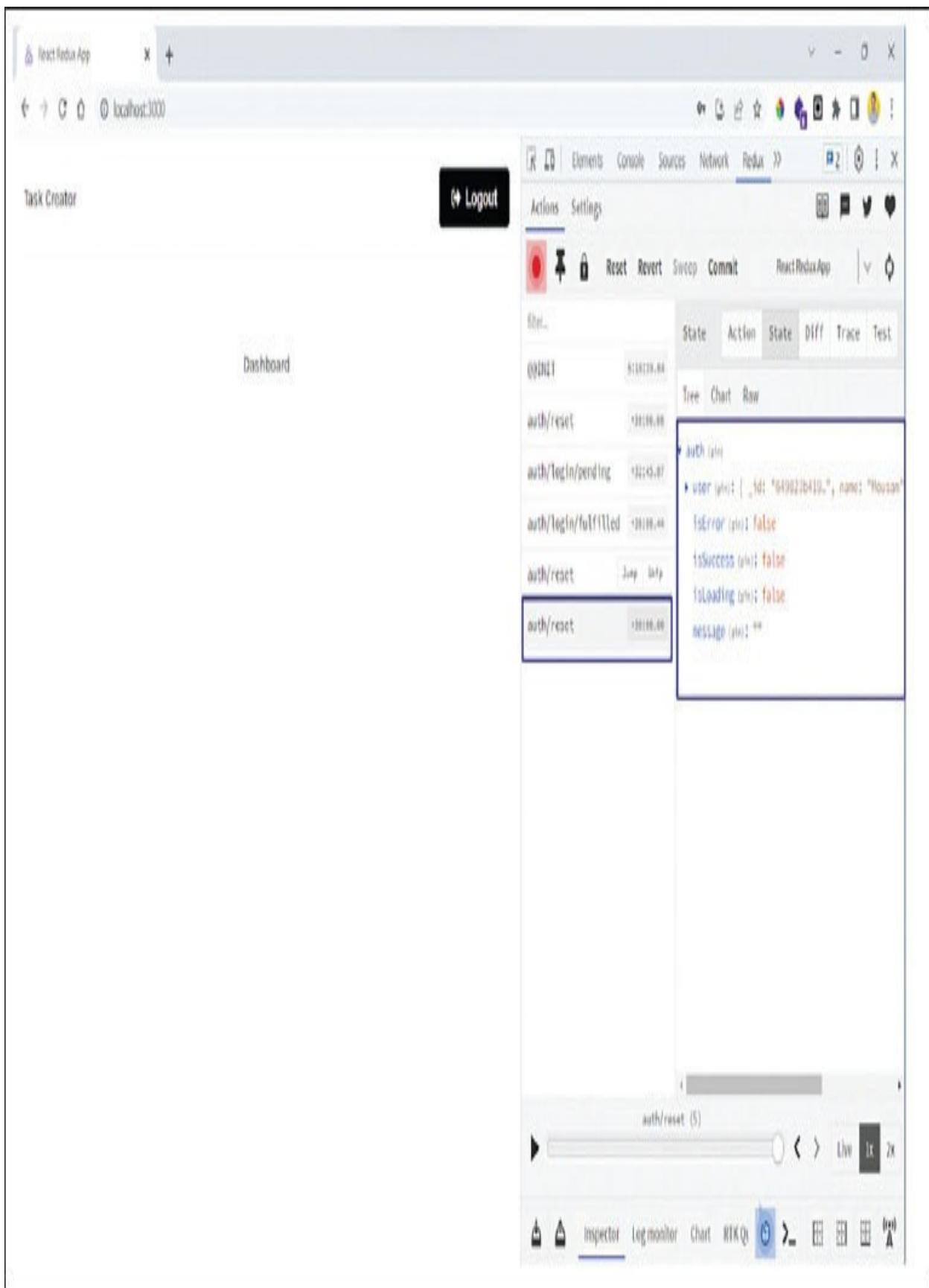


Figure 8.22: Reset state in the login

The user data was also saved in the Local Storage of the browser, containing all the fields of the user object.

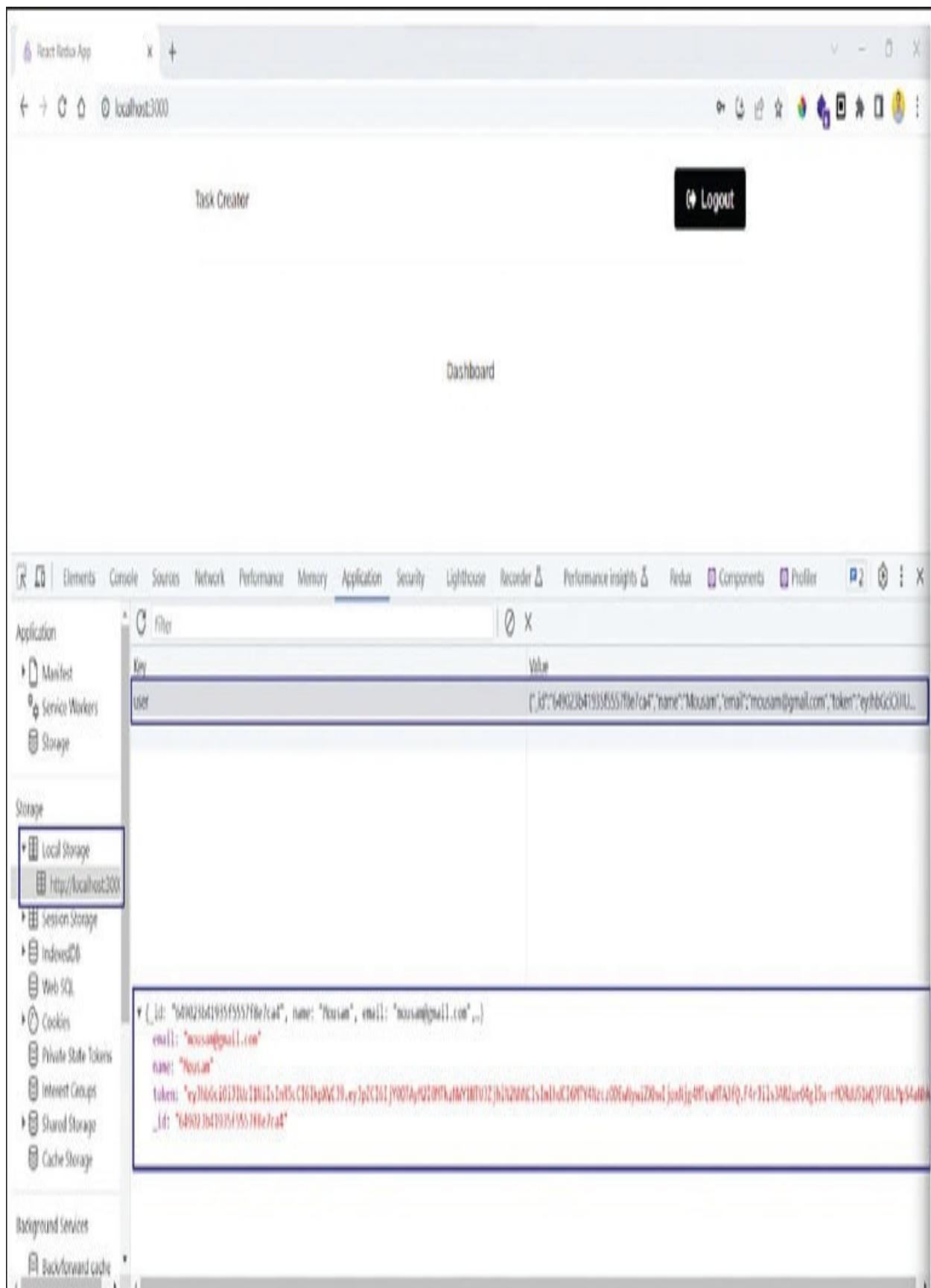


Figure 8.23: Local Storage in Browser

Conclusion

In this chapter, we have learned about implementing Logout in our project through the toolkit. The logout was implemented by simply removing the user from local storage and making its state null. We also created the logic logout button to be shown if the user is logged in.

After that, we created the Login functionalities, which were quite similar to Register in the previous chapter. Finally, we were able to log in as a user by submitting a form from the front end.

In the next chapter, we will implement the restriction of the dashboard and the creation of a Task form.

Points to remember

How to show a logout button if the user is logged in.

Using a Slice in the latest Redux toolkit, we combine the functionalities of reducer and action creator.

How to get global state from Redux in a component and also dispatch action creators from a component.

CHAPTER 9

Dashboard Creation and Task Form

Introduction

This chapter covers the implementation of the Dashboard in the frontend. We will also restrict the dashboard to authorized users. In this chapter, we are also going to create the Task slice in Redux. Finally, we are also going to create the Task Form.

Structure

In this chapter, we are going to discuss the following topics:

Changing Dashboard logic

Creating Task Slice

Creating Task Form

Changing Dashboard Logic

Our Dashboard page was accessible to everyone in the earlier chapter. We are going to change the logic so that only logged-in users can see the dashboard page.

In the Dashboard.js file, we are first importing useEffect from react, because we will do an async task through it. Next, we are importing useNavigate hook from react-router-dom, which will be used to navigate to the login page.

Also, the useSelector is imported from react-redux. It will be used to access the global state of the user. Inside the Dashboard component, we are first creating the variables of navigate and user through the useNavigate and useSelector hooks, respectively.

After that, from within an useEffect, which will run when the user navigates change, we are checking if a user is not available. If that is the case, we are navigating the user to the /login route.

```
import { useEffect } from 'react'  
import { useNavigate } from 'react-router-dom'  
import { useSelector } from 'react-redux'  
  
const Dashboard = () => {
```

```
const navigate = useNavigate()

const { user } = useSelector(state => state.auth)

useEffect(() => {

  if (!user) navigate(</login>)

}, [user, navigate])

return (

<div>Dashboard</div>

)

}

export default Dashboard
```

File Edit Selection View Go Run Terminal Help Dashboard.js - Task MERN - Visual Studio Code

EXPLORER ... JS Dashboard.js U X

frontend > src > components > JS Dashboard.js > (0) Dashboard > (0) useEffect() callback

```
1 import { useEffect } from 'react'
2 import { useNavigate } from 'react-router-dom'
3 import { useSelector } from 'react-redux'
4
5 const Dashboard = () => {
6   const navigate = useNavigate()
7   const { user } = useSelector(state => state.auth)
8
9   useEffect(() => {
10     if (!user) navigate('/login')
11   }, [user, navigate])
12
13   return (
14     <div>Dashboard</div>
15   )
16 }
17
18 export default Dashboard
```

backend frontend node_modules public src app components features App.js App.test.js index.css index.js setupTests.js .ignore package-lock.json package.json README.md

> OUTLINE
> TIMELINE

master* 0 △ 0 Ln 10, Col 16 Spaces: 2 UTF-8 CRLF () JavaScript ⚡ Go Live ⚡

Figure 9.1: Updating Dashboard.js file

Now, in the browser type `http://localhost:3000/` and hit enter.

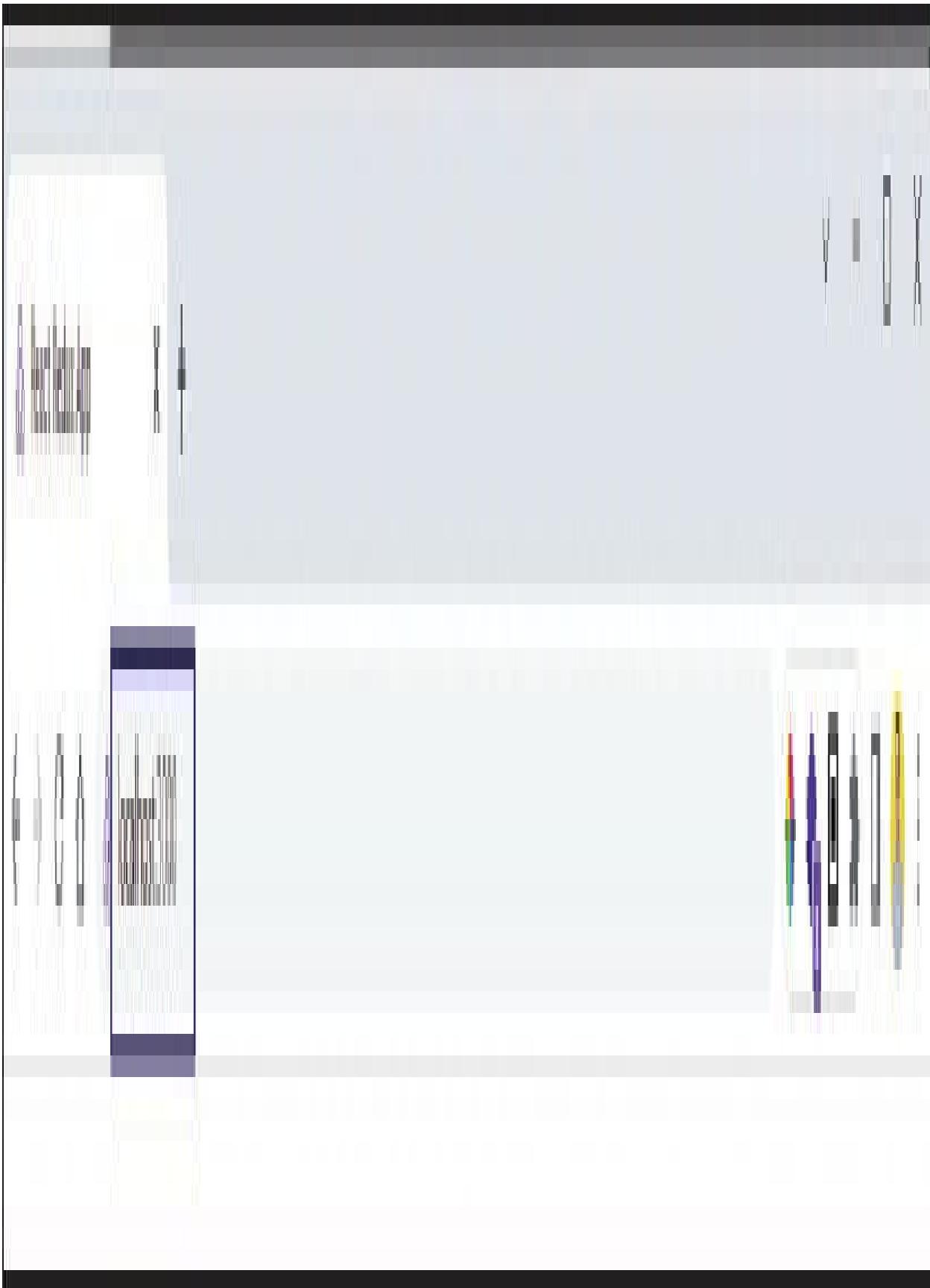


Figure 9.2: Visiting localhost in browser

We will be redirected to `http://localhost:3000/login` because of the new code in `Dashboard.js` file. We don't have a user state yet.

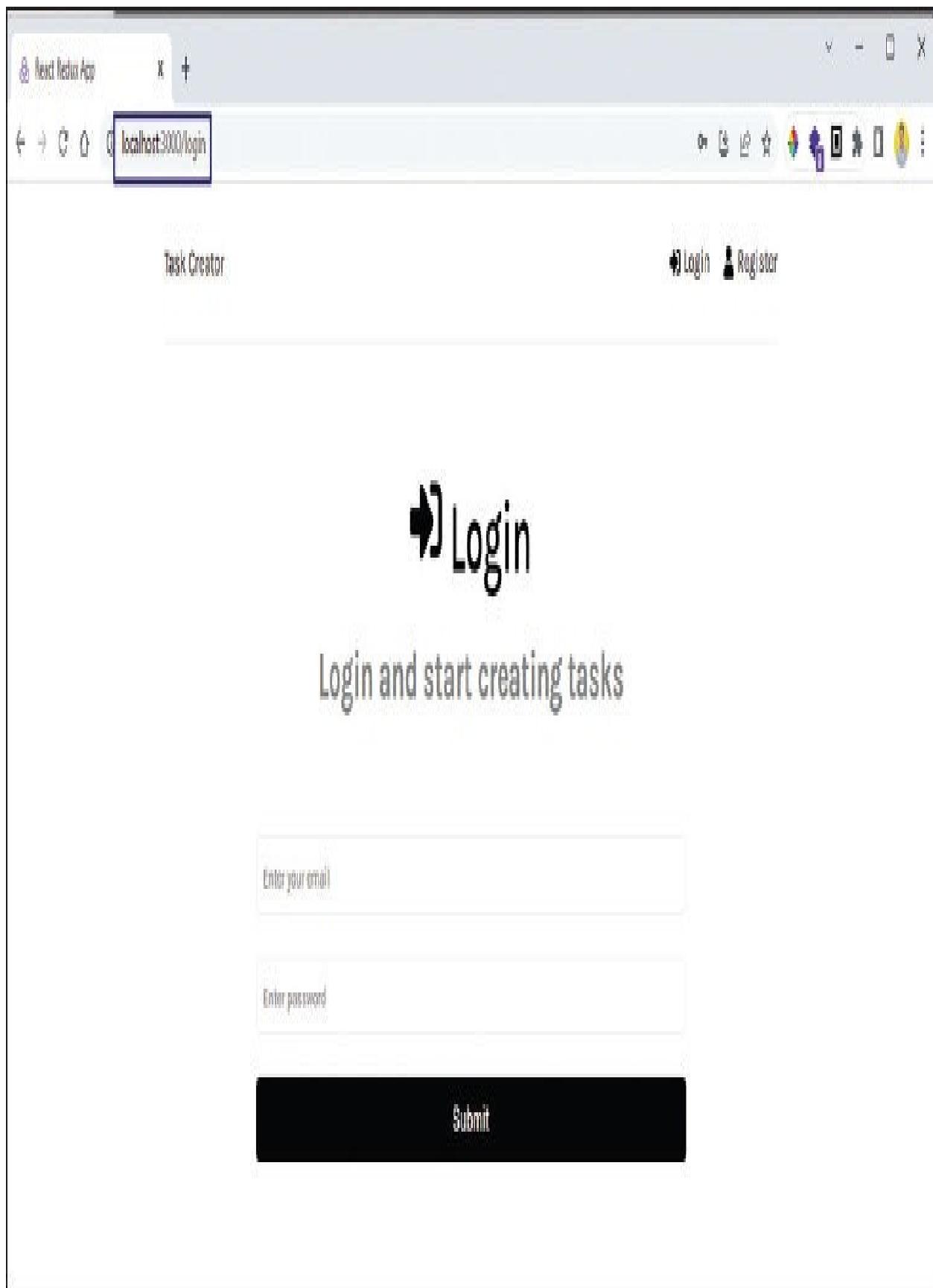


Figure 9.3: Dashboard route changing to login route

Now, in the preceding screen login with a valid email and password. And we will be redirected to the Dashboard page.

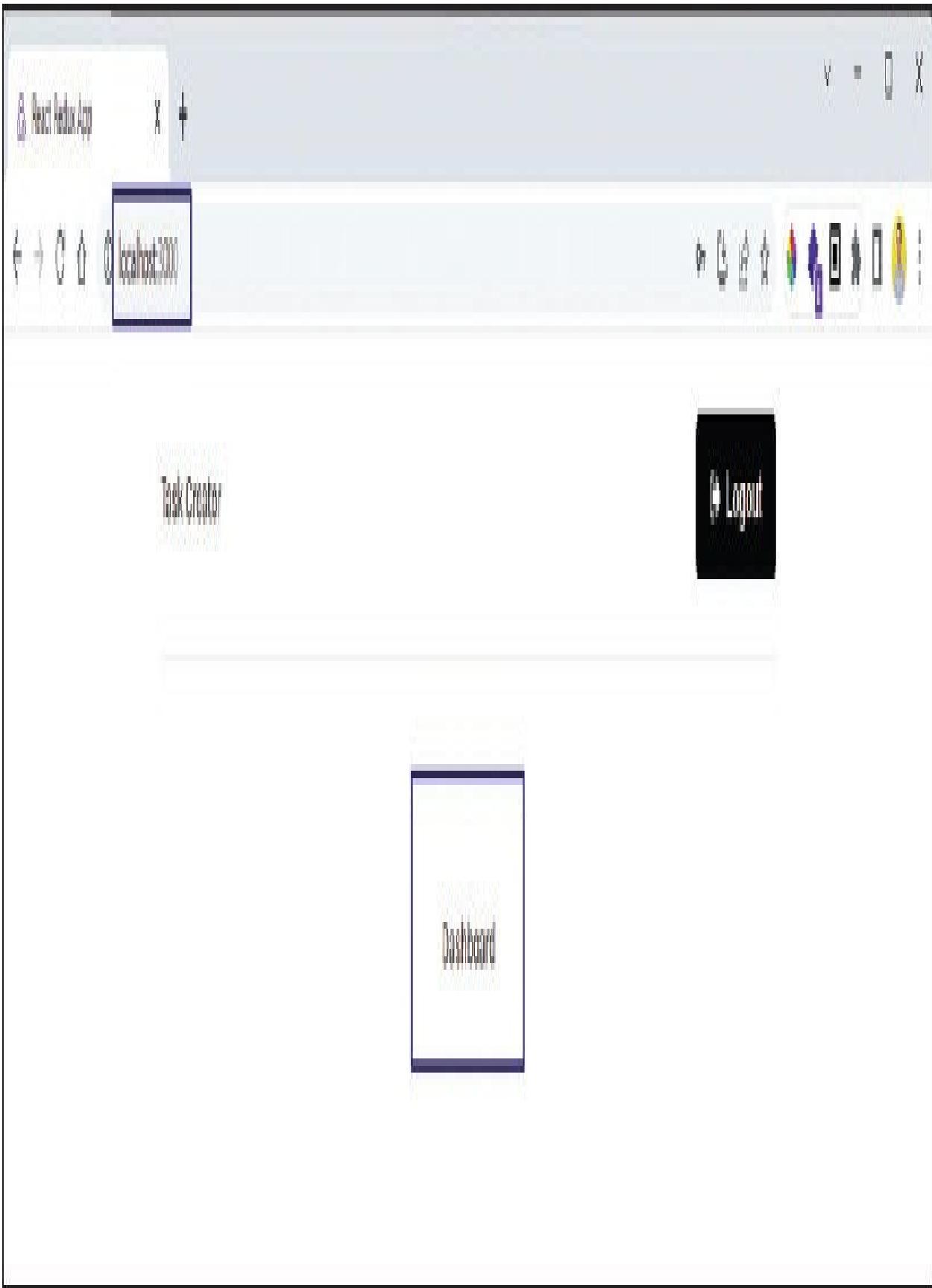


Figure 9.4: Dashboard shown only after logging

Creating Task Slice

Now, we will create the task slice and service which will be used in the dashboard to create tasks. Similar to the auth logic, we will create a folder called tasks inside the features folder.

Now, create a file taskSlice.js and taskService.js inside the tasks folder. Inside the taskSlice.js file, put the following code:

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
```

```
const initialState = {
```

```
  tasks: [],
```

```
  isError: false,
```

```
  isSuccess: false,
```

```
  isLoading: false,
```

```
  message: <>
```

```
}
```

```
export const taskSlice = createSlice({
```

```
  name: <task>,
```

```
initialState,  
reducers: {  
  reset: state => initialState  
}  
})  
  
export const { reset } = taskSlice.actions  
export default taskSlice.reducer
```

Here, we are first importing `createSlice` and `createAsyncThunk` from redux toolkit. The `createSlice` will be used to create the slice used for reducer and action. And the `createAsyncThunk` will be used for asynchronous actions, which is an API call in our case later on.

Next, we have the `initialState` of our slice which is an object. Here, we have `tasks` key first which is equal to an empty array. We also have values for `isError`, `isSuccess`, and `isLoading` which are set as false. These will be used when we do the API call. Lastly, we have a message as an empty string.

Next, we are exporting the `taskSlice`, which is created using the `createSlice` method. Here, we are first giving the name of the slice as a `task`. Then we are taking the `initialState`. After that, we have a `reducers` value, which has a `reset`. This `reset` can be used to restore the state to its initial state.

At the end, we are exporting the `reset` action and the reducer.

File Edit Selection View Go Run Terminal Help taskSlice.js - task-MERN - Visual Studio Code

EXPLORER ... Dashboard.js X taskSlice.js X taskService.js U

TASK-MERN frontend > src > features > tasks > taskSlice.js > default

frontend node_modules public src app components Dashboard.js Header.js Login.js Register.js Spinner.js

features auth tasks taskService.js taskSlice.js

App.js App.test.js # index.css index.js setupTests.js .gitignore package-lock.json package.json

OUTLINE TIMELINE

```
1 import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
2
3 const initialState = {
4   tasks: [],
5   isError: false,
6   isSuccess: false,
7   isLoading: false,
8   message: ''
9 }
10
11 export const taskSlice = createSlice({
12   name: 'task',
13   initialState,
14   reducers: {
15     reset: state => initialState
16   }
17 })
18
19 export const { reset } = taskSlice.actions
20 export default taskSlice.reducer
```

master* 0 0 0 In 20, Col 25 Spaces:4 UTF-8 CRLF () JavaScript Go Live

Figure 9.5: Basic taskSlice.js file

Now, we will import the task reducer in the store.js file. Because of the store in a Redux app, we are able to use global state in different components. So, now we will be able to access the tasks state in various components.

A screenshot of the Visual Studio Code interface showing a file named `store.js` in a project titled "Task-MERN". The code is configuring a Redux store with two reducers: `authReducer` and `taskReducer`.

```
store.js - Task-MERN - Visual Studio Code
File Edit Selection View Go Run Terminal Help
store.js - Task-MERN - Visual Studio Code
EXPLORER Dashboard.js taskSlice.js store.js taskService.js
TASK-MERN frontend > src > app > store.js ...
1 import { configureStore } from '@reduxjs/toolkit';
2 import authReducer from './features/auth/authSlice'
3 Import taskReducer from './features/tasks/taskSlice'
4
5 export const store = configureStore({
6   reducer: {
7     auth: authReducer,
8     tasks: taskReducer
9   }
10 });
11
```

The code editor shows the following structure:

- File menu: File, Edit, Selection, View, Go, Run, Terminal, Help.
- Tab bar: `Dashboard.js`, `taskSlice.js`, `store.js` (active), `taskService.js`.
- Explorer sidebar:
 - `frontend`
 - `> node modules`
 - `> public`
 - `> src
 - > app
 - store.js (highlighted)
 - > components
 - Dashboard.js
 - Header.js
 - Login.js
 - Register.js
 - Spinner.js`
- Code editor:

```
1 import { configureStore } from '@reduxjs/toolkit';
2 import authReducer from './features/auth/authSlice'
3 Import taskReducer from './features/tasks/taskSlice'
4
5 export const store = configureStore({
6   reducer: {
7     auth: authReducer,
8     tasks: taskReducer
9   }
10 });
11
```

Figure 9.6: Adding tasks state in store.js file

Now, back in <http://localhost:3000/> after logging in we will see the initial state of tasks in the Redux devtools tab of the developer console.

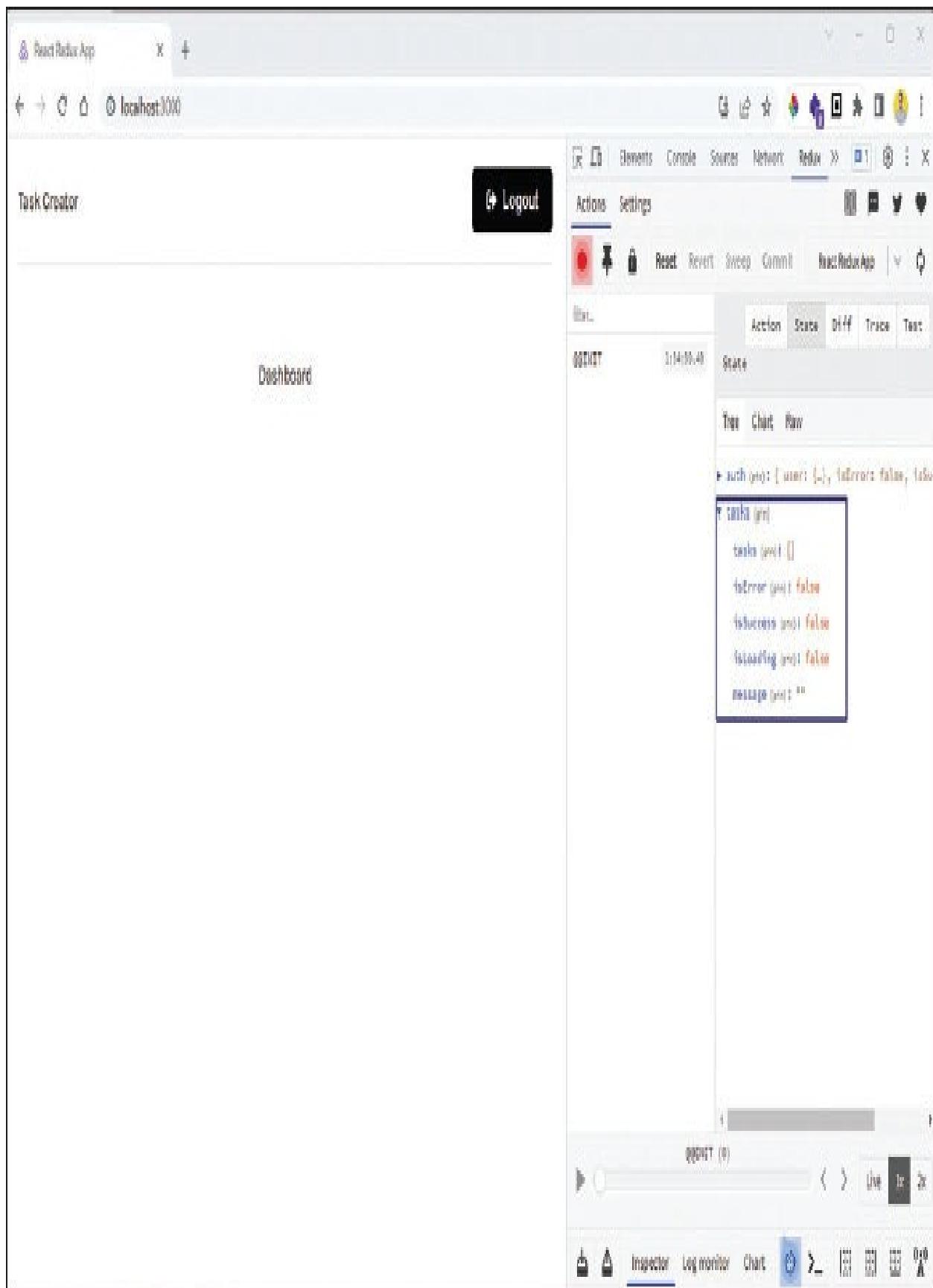


Figure 9.7: Initial tasks state in Redux devtools

Back in the Dashboard.js file, we will show a Welcome message now. Here, we are showing the user name also from the user global state. Since we should be successfully logged in when reaching this component, we will get this user name.

```
<>

<section className=>heading>

<h1>Welcome {user && user.name}</h1>

<p>Tasks Dashboard</p>

</section>

</>
```

File Edit Selection View Go Run Terminal Help Dashboard.js - task-MRN - Visual Studio Code

Dashboard.js X taskSlice.js X taskService.js X

frontend > src > components > Dashboard.js > Dashboard

```
1 import { useEffect } from 'react'
2 import { useNavigate } from 'react-router-dom'
3 import { useSelector } from 'react-redux'
4
5 const Dashboard = () => {
6   const navigate = useNavigate()
7   const { user } = useSelector(state => state.auth)
8
9   useEffect(() => {
10     if (!user) navigate('/login')
11   }, [user, navigate])
12
13   return (
14     <>
15       <section className='heading'>
16         <h1>Welcome {user && user.name}</h1>
17         <p>Tasks Dashboard</p>
18       </section>
19     </>
20   )
21 }
22
23 export default Dashboard
```

In 14, Col 7 Spaces: 2 UTF-8 CRLF JavaScript Go Live

Figure 9.8: Adding Username in Dashboard.js file

Back in `http://localhost:3000/` after logging, we will see the user name also, with a text of Tasks Dashboard.



Figure 9.9: Showing Username in localhost

Creating Task Form

Now, we will create a Task Form, which will be used to take the user input in the Dashboard for tasks. So, first create a file TaskForm.js inside the components folder.

Here, we are first importing the useState from react and the useDispatch hook from react-redux. Next, we have a basic TaskForm functional component, showing the text TaskForm inside a div.

```
import { useState } from 'react'  
import { useDispatch } from 'react-redux'  
  
const TaskForm = () => {  
  return (  
    <div>TaskForm</div>  
  )  
}  
  
export default TaskForm
```

The screenshot shows a dark-themed interface of Visual Studio Code. The top bar includes the 'File' menu, 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', and 'Help' options. The title bar indicates the project is 'TaskForm.js - Task-MERN - Visual Studio Code'. The left sidebar contains icons for 'EXPLORER', 'RECENT', 'REPO', 'REFRESH', and 'OPEN'. The main area shows a file tree under 'TASK-MERN' with subfolders like 'backend', 'frontend', 'node_modules', 'public', 'src', 'app', 'store.js', 'components', 'Dashboard.js', 'Headers.js', 'Login.js', 'Register.js', 'Spinner.js', and 'TaskForm.js'. The 'TaskForm.js' file is currently selected and highlighted with a green border. The code editor on the right displays the following content:

```
1 import { useState } from 'react'
2 import { useDispatch } from 'react-redux'
3
4 const TaskForm = () => {
5   return (
6     <div>TaskForm</div>
7   )
8 }
9
10 export default TaskForm
```

Figure 9.10: Creating TaskForm.js file

Now, we will import this TaskForm component in our Dashboard.js file. Here, we are first using an import statement for the TaskForm component. After that, we are adding it after the section.

```
import TaskForm from './TaskForm'

...
<></>

<section className=>heading>
  <h1>Welcome {user && user.name}</h1>
  <p>Tasks Dashboard</p>
</section>
<TaskForm />
</>
```

File Edit Selection View Go Run Terminal Help Dashboard · Task-MERN · Visual Studio Code

Dashboard.js X TaskForm.js X taskslice.js X taskService.js X

frontend > src > components > Dashboard > Dashboard

```
1 import { useEffect } from 'react'
2 import { useNavigate } from 'react-router-dom'
3 import { useSelector } from 'react-redux'
4 import TaskForm from './TaskForm'

5
6 const Dashboard = () => {
7   const navigate = useNavigate()
8   const { user } = useSelector(state => state.auth)
9
10  useEffect(() => {
11    if (!user) navigate('/login')
12  }, [user, navigate])
13
14  return (
15    <>
16      <section className='heading'>
17        <h1>Welcome {user && user.name}</h1>
18        <p>Tasks Dashboard</p>
19      </section>
20      <TaskForm />
21    </>
22  )
23}
24
```

In 20 Col 19 Spaces 2 UTF-8 CR LF () JavaScript ⚡ Go Live R Q

Figure 9.11: Adding TaskForm component in Dashboard.js file

Back in `http://localhost:3000/` we will see the TaskForm div being shown.



Figure 9.12: Showing TaskForm component in localhost

Next, we will complete the TaskForm component by adding the following code in TaskForm.js file:

```
const TaskForm = () => {  
  const [text, setText] = useState(<>)  
  
  const onSubmit = (e) => {  
    e.preventDefault()  
    setText(<>)  
  }  
  
  return (  
    <section className=>form>  
      <form onSubmit={onSubmit}>  
        <div className=>form-group>  
          <label htmlFor=>text>Task</label>  
          <input type=>text> id=>text> value={text} onChange={e =>  
            setText(e.target.value)}  
        />  
      </div>  
    </section>  
  )  
}
```

```
<div className=>form-group>

<button className=>btn btn-block type=>submit>Add Task</button>

</div>

</form>

</section>

)

}
```

Here, inside the TaskForm function, we are first having a state of text using the useState hook. Next, we have a submit function which is using e.preventDefault(), to prevent the page from refreshing when we hit the Add Task button. We are also setting the text state back to an empty screen here.

Inside the return, we have a section with a class of form. These styles are already there in the index.css file. Next, we have a form with an onSubmit handler which is calling the onSubmit function created earlier.

Next, we have a div with a class of form-group. Inside it, we have a label and an input box. This input box is changing the state of the text when the user types in it.

In the next div, we have a button with the class name for styles. Since this button is inside a form, it is of type submit. When we click this Add Task button, it will trigger the onSubmit handler of the form.

File Edit Selection View Go Run Terminal Help TaskForm.js Task-MERN Visual Studio Code

Dashboard.js U TaskForm.js U taskSlice.js U taskService.js U

frontend > src > components > TaskForm.js

```
4 const TaskForm = () => {
5   const [text, setText] = useState('')
6   const onSubmit = (e) => {
7     e.preventDefault()
8     setText('')
9   }
10
11   return (
12     <section className='form'>
13       <form onSubmit={onSubmit}>
14         <div className='form-group'>
15           <label htmlFor='text'>Task</label>
16           <input type='text' id='text' value={text} onChange={e => setText(e.target.value)}>
17           />
18         </div>
19         <div className='form-group'>
20           <button className='btn btn-block' type='submit'>Add Task</button>
21         </div>
22       </form>
23     </section>
24   )
25 }
26
```

Ln 13, Col 39 Spaces: 4 CRLF {} JavaScript Go Live ⚡

Figure 9.13: Adding login TaskForm.js file

Going back to `http://localhost:3000/` will show us a nice form in the Dashboard, with input box and a button.

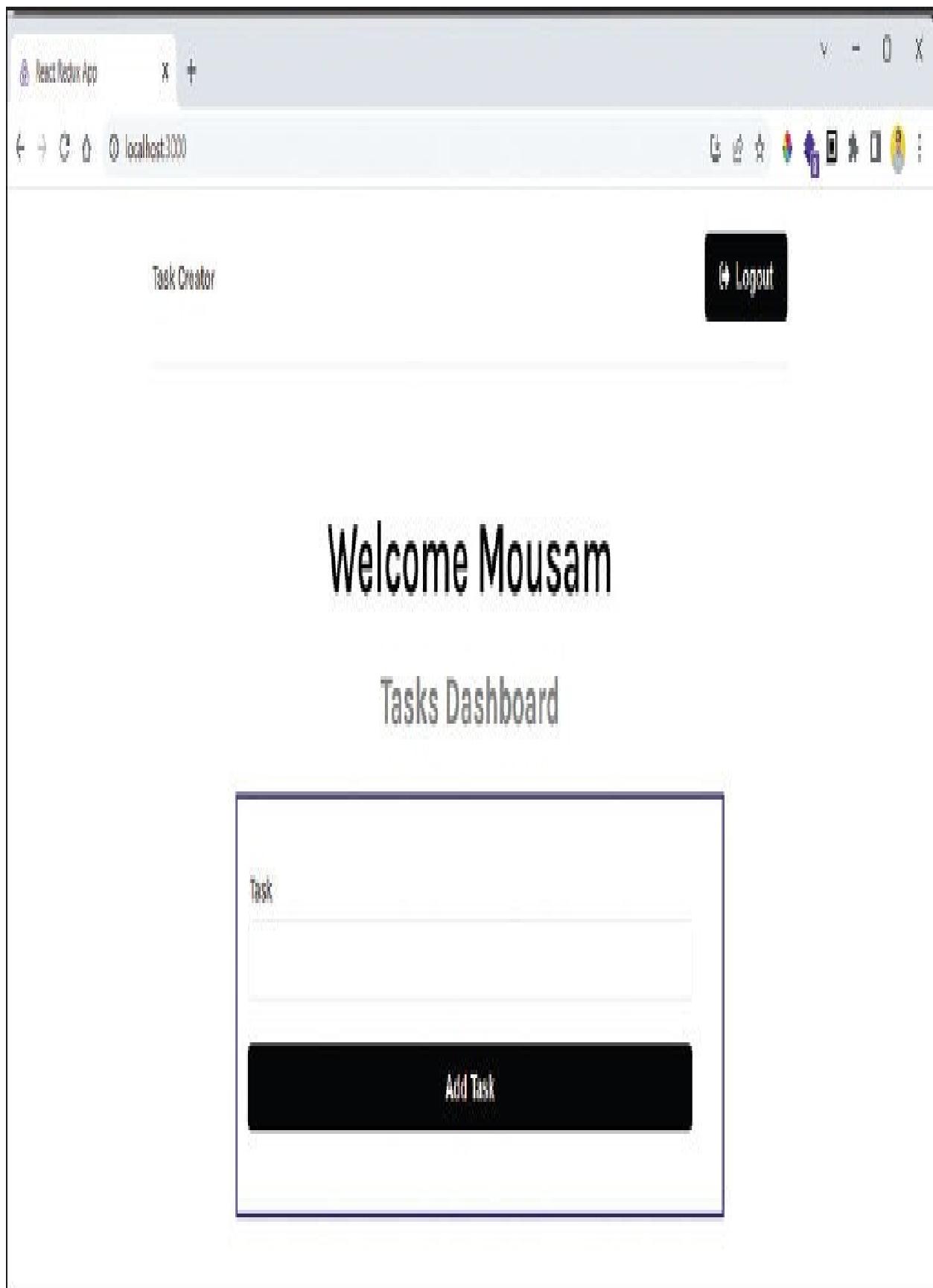


Figure 9.14: Updated TaskForm component in localhost

Lastly, in the TaskForm.js file, we will first import the `createTask` method from `taskSlice.js` file. We have not yet created this method, and will be created in the next chapter.

After that, inside the TaskForm function, we are first creating a dispatch variable by using the `useDispatch()` hook. Next, we are using the dispatch variable inside the `onSubmit` function. Here, we are using the dispatch variable to call the `createTask` method with the `text` state.

```
import { useState } from 'react'
import { useDispatch } from 'react-redux'
import { createTask } from '../features/tasks/taskSlice'

const TaskForm = () => {
  const [text, setText] = useState('')
  const dispatch = useDispatch()

  const onSubmit = (e) => {
    e.preventDefault()
    dispatch(createTask({ text }))
  }

  return (
    <form>
      <input type="text" value={text} onChange={setText} />
      <button type="submit" onClick={onSubmit}>Add Task</button>
    </form>
  )
}

export default TaskForm
```

```
setText(<>)
```

```
}
```

```
return (
```

```
...
```

```
)
```

```
}
```

File Edit Selection View Go Run Terminal Help taskForm.js - Task-MERN - Visual Studio Code

Dashboard.js U TaskForm.js U X taskSlice.js U taskService.js U

frontend > src > components > TaskForm.js > [0] TaskForm

```
1 import { useState } from 'react'
2 import { useDispatch } from 'react-redux'
3 import { createTask } from '../features/tasks/taskSlice'
4
5 const TaskForm = () => {
6   const [text, setText] = useState('')
7   const dispatch = useDispatch()
8
9   const onSubmit = (e) => {
10     e.preventDefault()
11     dispatch(createTask({ text }))
12     setText('')
13   }
14
15   return (
16     <section className='form'>
17       <form onSubmit={onSubmit}>
18         <div className='form-group'>
19           <label htmlFor='text'>Task</label>
```

Figure 9.15: Adding dispatch login in TaskForm.js file

Conclusion

In this chapter, we learned about changing the Dashboard logic in our project. Here, we have restricted the dashboard to the logged in user. After that, we created the task slice, which will be used to create tasks. We also created a Task Form. In this form, we can take the user input to create a task.

In the next chapter, we will complete the task functionalities. And also fetch the tasks from the server and display them in Dashboard. We will also add the delete tasks logic.

Points to Remember

Implementing logic for only logged-in users to see the dashboard.

Using a slice to create the global task state.

Creating a form in React for the input of tasks.

Multiple Choice Questions

What does the useNavigate hook do?

Navigate to home route

Navigate to any provided route

Navigate to login route

There is nothing like useNavigate hook

What does the useSelector hook do?

Access the local state

Access the component state

There is nothing like useSelector hook

Access the global state

In the redux toolkit the slice is a combination of

store and reducer

store and action

action and reducer

Reducer only

How to set a local state in a Functional React component?

Using useEffect hook

Using useState hook

Using this.state

State is not in functional components

What does the useDispatch hook do?

Used to dispatch action to Slice

Used to dispatch state to Slice

Used to get global state from Slice

There is nothing like useDispatch hook

Answers

b

d

c

b

a

CHAPTER 10

Using Thunk and Completing App

Introduction

This chapter covers the implementation of Thunk in our Task workflow. We will also fetch the tasks from the server. This chapter also completes our app, in which we will add logic to display the tasks and also delete them.

Structure

In this chapter, we are going to cover the following topics:

Creating task with Async Thunk

Fetching tasks from the server

Displaying tasks

Deleting tasks

Creating task with Async Thunk

Back in the taskSlice.js file, which we created in our previous chapter, we will first import the taskService. Next, we are exporting a variable createTask. Here, again we are using the createAsyncThunk function. We don't need to import Thunk because the redux toolkit comes in-built with Thunk.

In the createAsyncThunk function, the first parameter is the tasks/create, which means from the tasks folder, take the create function. Next, the parameter is an async arrow function which takes the taskData and thunkAPI as a parameter.

Inside the function, we have a try...catch block. From the try block, we are first getting the token using the getState() method. Next, we call the createTask function from the taskService.js file with the taskData and token and then return it.

In the catch part, we are simply catching different kinds of errors and returning them. Note that the catch block will only execute if there is an error in the try block.

```
import taskService from './taskService'

...

export const createTask = createAsyncThunk(
  'tasks/create',
  async (taskData, thunkAPI) => {
```

```
try {

  const token = thunkAPI.getState().auth.user.token

  return await taskService.createTask(taskData, token)

} catch (error) {

  const message =

    (error.response && error.response.data && error.response.data.message)

  || error.message

  || error.toString()

  return thunkAPI.rejectWithValue(message)

}

}

)
```

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
import taskService from './taskService'

const initialState = ...

export const createTask = createAsyncThunk(
  'tasks/create',
  async (taskData, thunkAPI) => {
    try {
      const token = thunkAPI.getState().auth.user.token
      return await taskService.createTask(taskData, token)
    } catch (error) {
      const message =
        (error.response && error.response.data && error.response.data.message)
        || error.message
        || error.toString()
      return thunkAPI.rejectWithValue(message)
    }
  }
)

export const taskSlice = createSlice({
```

Figure 10.1: Creating createTask in taskSlice.js file

Next, we will create the task service. This service will be used to save the task data in the database. But since only the logged-in user should be able to do this, we will use a token also. So, create a file taskService.js inside the tasks folder.

Here, we are first importing axios and also creating a variable called API_URL which is set to /api/tasks/. Next, we have created a createTask function which takes the taskData and the token. Note that this only is the function which we are calling from our taskSlice.js file.

Inside the function, we have a config object first, where we are creating a headers key. The value of it is an object, which has a key of Authorization and a value of “Bearer token”. Note that it is mandatory to add the keyword Bearer to the token while doing a POST call with the token.

Next, we are doing a POST call to the API_URL with the taskData and config. The result of the api call which we get in response.data and returning back the response from the function. Lastly, we are exporting the createTask function.

```
import axios from ‘axios’
```

```
const API_URL = ‘/api/tasks/’
```

```
const createTask = async (taskData, token) => {
```

```
const config = {  
  headers: {  
    Authorization: `Bearer ${token}`,  
  },  
}  
  
const response = await axios.post(API_URL, taskData, config)  
  
return response.data  
}  
  
const taskService = { createTask, getTasks, deleteTask }  
  
export default taskService
```

A screenshot of the Visual Studio Code interface showing a file named `taskService.js`. The code implements a service for creating tasks using the `axios` library.

```
1 import axios from 'axios'
2
3 const API_URL = '/api/tasks/'
4
5 const createTask = async (taskData, token) => {
6   const config = {
7     headers: {
8       Authorization: `Bearer ${token}`,
9     }
10   }
11   const response = await axios.post(API_URL, taskData, config)
12   return response.data
13 }
14
15 const taskService = { createTask }
16
17 export default taskService
18
```

Figure 10.2: Creating createTask in taskService.js file

Inside our taskSlice.js file in the taskSlice function, we will add extraReducers. We will put the reducer action, which changes the state in these extraReducers.

In our code, we have three cases for pending, fulfilled and rejected cases. Since we are doing an API call to through our createTask function, we have three states in it. When the API call starts, we have a pending state since the API calls generally take 1–2 seconds. In the pending state, isLoading will be made to true, and we will also show a spinner in this state in the front end.

The next state is the fulfilled case, which will happen if our API call is successful. Here, we are making the isLoading as false and isSuccess as true and storing the payload which we receive back in the tasks state.

If, for some reason, the API call is not successful, we have a rejected case. Here, we are making the isLoading as false, isError as true. We are also storing the error in the message state, which we will receive in the payload.

```
extraReducers: (builder) => {
  builder
    .addCase(createTask.pending, (state) => {
      state.isLoading = true
    })
    .addCase(createTask.fulfilled, (state, action) => {
      state.isSuccess = true
      state.payload = action.payload
    })
    .addCase(createTask.rejected, (state, action) => {
      state.isError = true
      state.message = action.payload
    })
}
```

```
.addCase(createTask.fulfilled, (state, action) => {
  state.isLoading = false
  state.isSuccess = true
  state.tasks.push(action.payload)
})

.addCase(createTask.rejected, (state, action) => {
  state.isLoading = false
  state.isError = true
  state.message = action.payload
})

}
```

File Edit Selection View Go Run Terminal Help taskSlice - task-MERN - Visual Studio Code

Dashboard.js X taskForm.js X taskSlice.js X taskService.js X

Frontend > src > features > tasks > taskSlice.js > taskSlice.js > extraReducers

```
28 export const taskSlice = createSlice({
29   name: 'task',
30   initialState,
31   reducers: {
32     reset: state => initialState
33   },
34   extraReducers: (builder) => {
35     builder
36       .addCase(createTask.pending, (state) => {
37         state.isLoading = true
38       })
39       .addCase(createTask.fulfilled, (state, action) => {
40         state.isLoading = false
41         state.isSuccess = true
42         state.tasks.push(action.payload)
43       })
44       .addCase(createTask.rejected, (state, action) => {
45         state.isLoading = false
46         state.isError = true
47         state.message = action.payload
48       })
49     }
50   })
```

Line 48 Col 4 Spaces 2 UFT-8 CRLF {} JavaScript ⚡ Go Live R Q

Figure 10.3: Creating extraReducers in taskSlice.js file

Now in Chrome/Firefox browser, go to <http://localhost:3000/> and log in if not done. Also, open the Redux devtools to see the redux states. We will see the initial login states of auth and tasks in the redux devtools.

Here, add anything in the Task input box and click the Add Task button.

← → ⌂ ⌂ ⌂ localhost:3000

Task Creator [Logout](#)

Actions Settings

React DevTools

Elements Console Sources Network Redux

Reset Report Sweep Commit ReactReduxApp

filter: Action State Diff Trace Test

INIT 3:16:10.11 State

Tree Chart Raw

auth (pro) { users: [], isError: false, isSuccess: true } tasks (val) { tasks: [], isError: false, isSuccess: true }

Task

Learn Flutter

Add Task

REDUX (0)

Live 1x 2x

Inspector Log monitor Chart

Figure 10.4: Adding text in Frontend

Here, first, the pending action ran from the extraReducers in the taskSlice.js file. Here, the isLoading was set to true, and the rest states were kept as it was earlier.

Next, the API call to the backend was successful, so the fulfilled action had run. This has made the isLoading as false, isSuccess as true and isError as false. And the tasks state was set to the payload received from the backend. This payload consists of text, user, _id, createdAt, and updatedAt.

localhost:3000

Task Creator [Logout](#)

Welcome Mousam

Tasks Dashboard

Task

[Add Task](#)

Actions Settings

File... Elements Console Sources Network Redux > [1](#) [2](#) X

Reset Revert Swap Commit React Reducer

File... Action State Diff Trace Test

initial

tasks/create/pending 00:00:00

tasks/create/fulfilled 00:00:00

auth (pending) { user: {}, isLoading: false, isSuccess: false, isError: true, message: "" }
tasks (1)
+ tasks (1)
+ 0 (1)
text (1): "Learn Flutter"
user (1): "64902364103574557fe7ca4"
id (1): "649177127982653c8888feb7"
createdAt (1): "2023-06-28T09:53:22.703Z"
updatedAt (1): "2023-06-28T09:53:22.703Z"
isSuccess (1): true
isLoading (1): false
message (1): ""

tasks/create/fulfilled (1)

Inspector Log monitor Chart

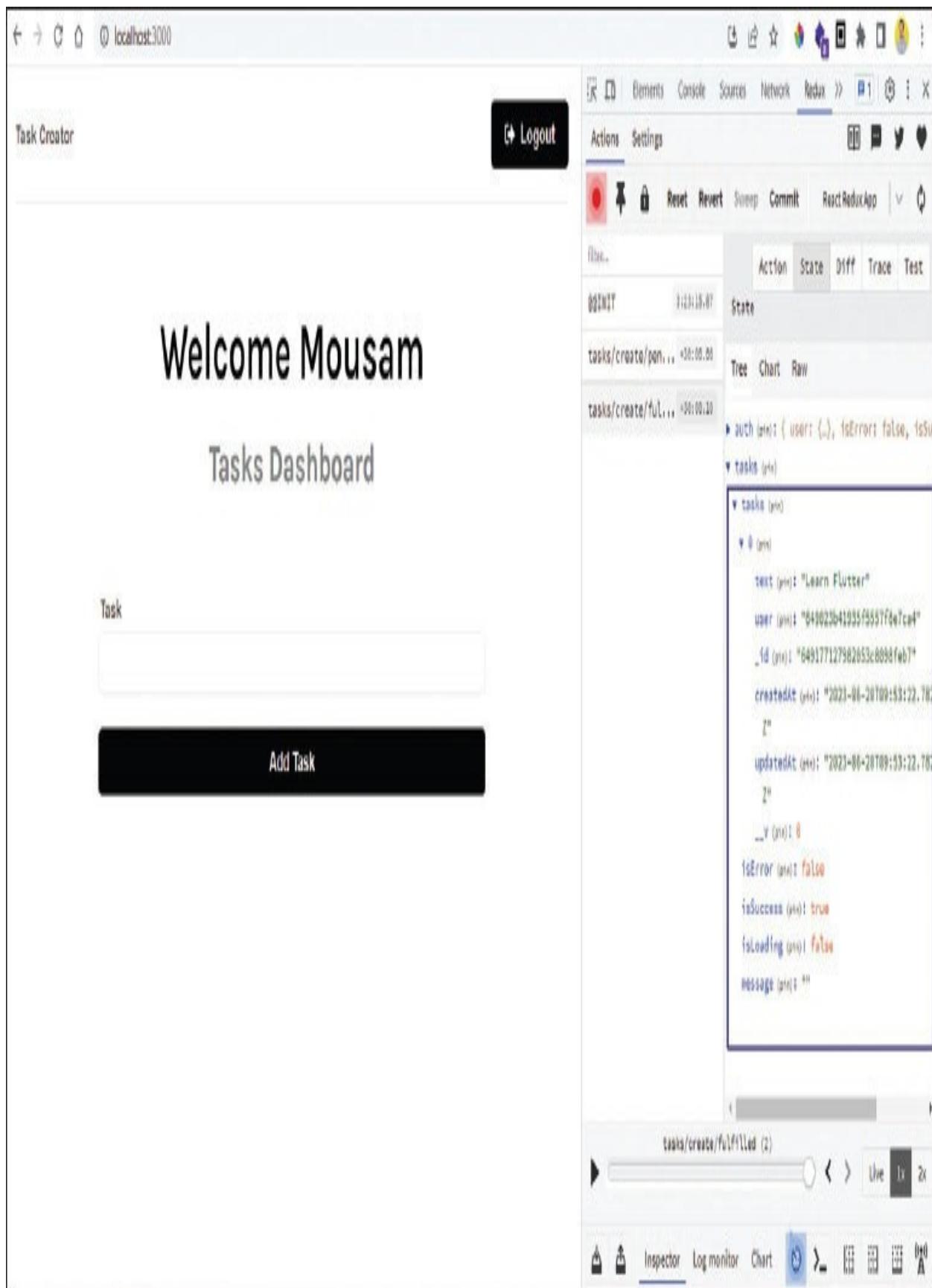


Figure 10.5: Task been added in Redux

Finally, we checked the value for tasks in the connected MongoDB database, and a new field was added to it.

← → ⌂ ⌂ cloud.mongodb.com/cluster/104c103a625214/metrics/replSet/407e4c5a12x107ee2a8/explore/backups/tasks/find

Atlas Home Help Network

Heroku App Databases App Services Cloud

Search Document 83,349 Logical Data 102,468 Total Documentation 1,169,933 Total 1,242,946

DEPLOYMENT tasksApp

Database tasks

Data Link tasks

Services

Triggers

Data API

Data Federation

Search

SECURITY

Setup

Credentials

Network Access

Advanced

Logs

System Status All Logs

©2023 MongoDB, Inc. Terms Privacy Attributing Contact Sales

RELOAD DOCUMENT

Reset Apply More Options

_id: ObjectId('401712798205e2001fe07')

text: 'Learn United States English'

user: ObjectId('401712798205e2001fe07')

createdAt: 2023-01-27T09:24:05.429Z

updatedAt: 2023-01-27T09:23:11.055Z

_v: 0

Figure 10.6: Task been added to the database

Fetching tasks from server

Now, we will fetch all tasks from the server. For this, in the taskSlice.js file, we will create a getTasks variable. Here, we are again using the createAsyncThunk function.

In the createAsyncThunk function, the first parameter is the tasks/getAll, which means from the tasks folder, take the getAll function. Next, the parameter is an async arrow function which takes _ and thunkAPI as a parameter. This is a common JavaScript way to say that no parameter is required.

Inside the function, we have a try...catch block. From the try block, we are first getting the token using the getState() method. Next, we call the getTasks function from the taskService.js file with the token and then return it.

In the catch part, we are simply catching different kinds of errors and returning them.

```
export const getTasks = createAsyncThunk('tasks/getAll',
  async (_, thunkAPI) => {
    try {
      const token = thunkAPI.getState().auth.user.token
      return await taskService.getTasks(token)
    } catch (error) {
      return thunkAPI.rejectWithValue(error.message)
    }
  }
)
```

```
    } catch (error) {  
  
        const message =  
            (error.response && error.response.data && error.response.data.message)  
                || error.message  
  
            || error.toString()  
  
        return thunkAPI.rejectWithValue(message)  
    }  
}  
)  
)
```

File Edit Selection View Go Run Terminal Help taskSlice - Task-MERN - Visual Studio Code

taskSlice.js

```
fronted > src > features > tasks > taskSlice.js > 10 getTasks > 11 createAsyncThunk(tasks/getAll)/callback > 12 message
```

```
22         || error.toString())
23     return thunkAPI.rejectWithValue(message)
24   }
25 }
26 )
27
28 export const getTasks = createAsyncThunk(
29   'tasks/getAll',
30   async (_, thunkAPI) => {
31     try {
32       const token = thunkAPI.getState().auth.user.token
33       return await taskService.getTasks(token)
34     } catch (error) {
35       const message =
36         (error.response && error.response.data && error.response.data.message) ||
37         error.message
38         || error.toString()
39       return thunkAPI.rejectWithValue(message)
40     }
41   }
42 )
43
44 export const taskSlice = createSlice({
45   name: 'task',
```

Figure 10.7: getTasks function in taskSlice.js file

Back in the taskService.js file, we have created a getTasks function which takes the token as a parameter. This is the function which we are calling from our taskSlice.js file.

Inside the function, we have a config object first, similar to the one created in the createTask function earlier in the chapter.

Next, we are doing a GET call to the API_URL with the config. The result of the api call which we get in response.data and returning back the response from the function. Lastly, we are exporting the getTasks function.

```
const getTasks = async token => {
  const config = {
    headers: {
      Authorization: `Bearer ${token}`,
    }
  }
  const response = await axios.get(API_URL, config)
  return response.data
}
```

```
const taskService = { createTask, getTasks }
```

A screenshot of the Visual Studio Code interface. The title bar shows "taskService.js (task-MERN - Visual Studio Code)". The code editor displays the following JavaScript code:

```
File Edit Selection View Go Run Terminal Help taskService.js (task-MERN - Visual Studio Code)

Dashboard.js ✘ TaskForm.js ✘ tasks.js ✘ taskService.js ✘

frontend > src > features > tasks > taskService.js > taskService > getTasks

11 const response = await axios.post(API_URL, taskData, config)
12 return response.data
13 }
14
15 const getTasks = async token => {
16   const config = {
17     headers: {
18       Authorization: `Bearer ${token}`,
19     }
20   }
21   const response = await axios.get(API_URL, config)
22   return response.data
23 }
24
25 const taskService = { createTask, getTasks }
26
27 export default taskService
28
```

The code is organized into two main functions: `createTask` and `getTasks`. The `getTasks` function takes a `token` as an argument and returns a promise that resolves to the data from a GET request to `API_URL`. The `taskService` object exports both `createTask` and `getTasks`.

Figure 10.8: getTasks function in taskService.js file

Inside our taskSlice.js file in the taskSlice function, we will add the new cases in the extraReducers.

In our code, we have three cases for pending, fulfilled and rejected cases. Since we are doing an API GET call to through our getTasks function, we have three states in it. When the API call starts, we have a pending state. In the pending state, isLoading will be made to true.

The next state is the fulfilled case, which will happen if our API call is successful. Here, we are making the isLoading as false and isSuccess as true and storing the payload which we receive back in the tasks state.

If, for some reason, the API call is not successful, we have a rejected case. Here, we are making the isLoading as false, isError as true. We are also storing the error in the message state, which we will receive in the payload.

```
.addCase(getTasks.pending, (state) => {  
    state.isLoading = true  
})  
.addCase(getTasks.fulfilled, (state, action) => {  
    state.isLoading = false  
    state.isSuccess = true
```

```
state.tasks = action.payload  
})  
.addCase(getTasks.rejected, (state, action) => {  
state.isLoading = false  
state.isError = true  
state.message = action.payload  
})
```

File Edit Selection View Go Run Terminal Help taskSlice - Task-MERN - Visual Studio Code

Dashboard.js TaskForm.js taskSlice.js taskService.js

frontEnd/src/features/tasks/taskSlice.js

```
60     .addCase(createTask.rejected, (state, action) => {
61         state.isLoading = false
62         state.isError = true
63         state.message = action.payload
64     })
65     .addCase(getTasks.pending, (state) => {
66         state.isLoading = true
67     })
68     .addCase(getTasks.fulfilled, (state, action) => {
69         state.isLoading = false
70         state.isSuccess = true
71         state.tasks = action.payload
72     })
73     .addCase(getTasks.rejected, (state, action) => {
74         state.isLoading = false
75         state.isError = true
76         state.message = action.payload
77     })
78 }
79 })
80
81 export const { reset } = taskSlice.actions
82 export default taskSlice.reducer
```

Figure 10.9: extraReducer cases in taskSlice.js file

Now, in the Dashboard.js file, we will show a button instead of the earlier text. In the next part, we will add logic to show all tasks from a new component when we click this button.

Fil Edit Selection View Go Run Terminal Help Dashboards Chapter 10 Visual Studio Code

App M Dashboard.js U TaskList.js U TaskForm.js U Login.js U TaskService.js U

frontend > src > components > Dashboard.js > Dashboard

```
1 import { useEffect } from 'react'
2 import { useNavigate } from 'react-router-dom'
3 import { useSelector } from 'react-redux'
4 import TaskForm from './TaskForm'
5
6 const Dashboard = () => {
7   const navigate = useNavigate()
8   const [ user ] = useSelector(state => state.auth)
9
10  useEffect(() => {
11    if (!user) navigate('/login')
12  }, [user, navigate])
13
14  return (
15    <>
16      <section className='heading'>
17        <h1>Welcome {user && user.name}</h1>
18        <div style={{ display: 'flex', justifyContent: 'center' }}>
19          <button className='btn' onClick={() => navigate('/alltasks')}>Check Tasks</button>
20        </div>
21      </section>
22      <TaskForm />
23    </>
24  )
}
```

master ✘ ⑧ ⑨ ⑩ Ln 18 Col 67 Spaces 2 UTF-8 CR/LF (JavaScript Go Live ⌂ ⌂ ⌂

Figure 10.10: Adding button in Dashboard.js file

Back in the browser at <http://localhost:3000/>, we will see the button for all tasks.

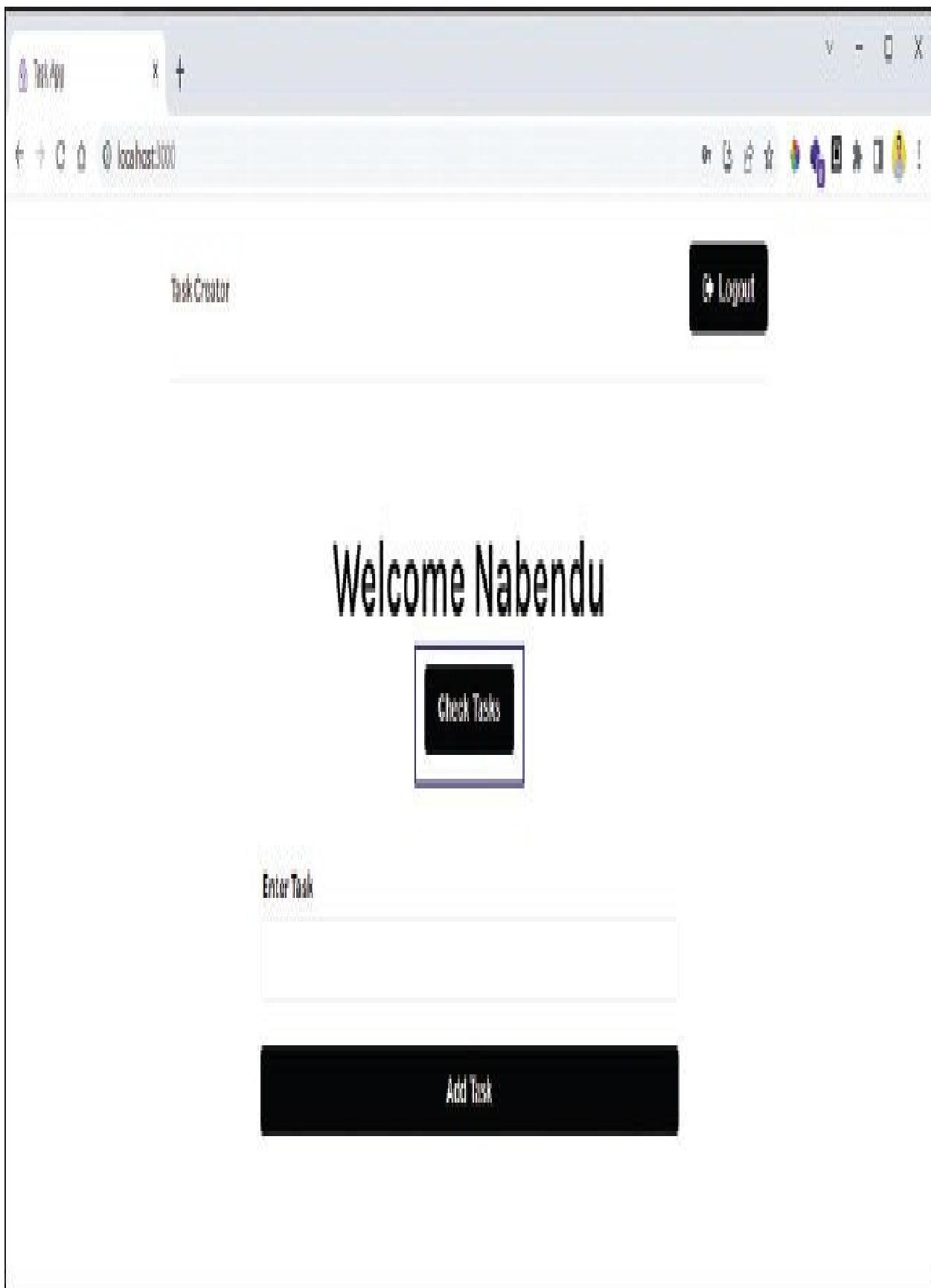


Figure 10.11: Button in localhost

Displaying the tasks

Now, we will display the tasks for which we have created the logic in the previous part. Here, in the App.js file, we will add a new route to show all tasks.

For this, we are adding a <Route> with the path as /alltasks and which will call the TaskList component. We are also importing this TaskList component, which we are going to create next.

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

...
...
<Route path='/alltasks' element={<TaskList />} />
```

File Edit Selection View Go Run Terminal Help App.js Chapter 10 - Visual Studio Code

JS App.js M X JS TaskList.js U JS Login.js U JS Dashboard.js U JS TaskItem.js U JS taskSlice.js U JS taskService.js U

fronted > src > JS App.js > App

```
1 import { BrowserRouter as Router, Routes, Route } from 'react-router-dom'
2 import Dashboard from './components/Dashboard';
3 import Login from './components/Login';
4 import Register from './components/Register';
5 import Header from './components/Header';
6 import { ToastContainer } from 'react-toastify'
7 import 'react-toastify/dist/ReactToastify.css'
8 import TaskList from './components/TaskList';

9

10 function App() {
11   return (
12     <>
13       <Router>
14         <div className='container'>
15           <Header />
16           <Routes>
17             <Route path='/' element={<Dashboard />} />
18             <Route path='/login' element={<Login />} />
19             <Route path='/register' element={<Register />} />
20             <Route path='/alltasks' element={<TaskList />} />
21           </Routes>
22         </div>
23       </Router>
24     <ToastContainer />
25   )
26 }

27 export default App;
```

In 20, Col 57 Spaces: 2 UTF-8 IF {} JavaScript Go Live ⌂ ⌂ ⌂ ⌂

Figure 10.12: Adding new route in App.js file

Next, create a new file TaskList.js in the components folder. Here, we are first importing the useEffect hook from react, which will be used to do the async task of calling different action creators through dispatch. Next, we import useSelector and useDispatch hooks from react-redux. As earlier, they will be used to access the global state and dispatch the action creator.

Next, we are importing the recently created getTasks function from the task slice and also importing the reset function. We are also importing a TaskItem component, which we have not created yet and the Spinner component from showing a Spinner. Lastly, we are importing the useNavigate hook, which will be used in our useEffect.

Now, inside the TaskList function, we are first creating a variable of dispatch and navigating using the useDispatch and useNavigate hooks, respectively. After that, using the useSelector hook, we get the global state of tasks. Also, notice that we are destructuring it to get the tasks, isLoading, isError, and message.

Next, inside the useEffect hook, we are first checking for errors and console logging it. If there are no errors, then we are calling the getTasks method through the dispatch function. We also have an unmounting state dispatching the reset method. The useEffect method also has a second parameter as an array of navigate, isError, message and dispatch. This means the useEffect will be fired whenever these will change. Lastly, the return statement is only showing a div, which we will update next.

```
import { useEffect } from 'react'
```

```
import { useSelector, useDispatch } from 'react-redux'
import { getTasks, reset } from '../features/tasks/taskSlice'
import TaskItem from './TaskItem'
import Spinner from './Spinner'
import { useNavigate } from 'react-router-dom'

const TaskList = () => {
  const dispatch = useDispatch()
  const navigate = useNavigate()
  const { tasks, isLoading, isError, message } = useSelector(state => state.tasks)

  useEffect(() => {
    if (isError) console.log(message)
    dispatch(getTasks())
  })
  return () => dispatch(reset())
}, [navigate, isError, message, dispatch])

return (
  <div>TaskList</div>
)
}
```

```
export default TaskList
```

File Edit Selection View Go Run Terminal Help taskList.js - Chapter 10 - Visual Studio Code

App.js Dashboard.js TaskList.js TaskItem.js taskService.js

frontend > src > components > TaskList.js > TaskList

```
1 import { useEffect } from 'react'
2 import { useSelector, useDispatch } from 'react-redux'
3 import { getTasks, reset } from '../features/tasks/taskSlice'
4 import TaskItem from './TaskItem'
5 import Spinner from './Spinner'
6 import { useNavigate } from 'react-router-dom'
7
8 const TaskList = () => {
9   const dispatch = useDispatch()
10  const navigate = useNavigate()
11  const { tasks, isLoading, isError, message } = useSelector(state => state.tasks)
12
13  useEffect(() => {
14    if (isError) console.log(message)
15    dispatch(getTasks())
16    return () => dispatch(reset())
17  }, [navigate, isError, message, dispatch])
18
19  return (
20    <div>TaskList</div>
21  )
22}
23
24 export default TaskList
```

In 20 Col 22 Spaces: 4 UTF-8 CR LF () JavaScript Go Live R D

Figure 10.13: The TaskList.js file

Next, we will update the return statement inside the TaskList.js file. Here, we are using the ternary operator and checking if the isLoading is true, then showing the Spinner component. Or else the other part is executed. Here, we are first checking if the tasks array length is greater than zero.

If it is the case, then we are looping through the tasks variable and sending each task to the TaskItem component.

```
isLoading ? <Spinner /> : (  
  <section className='content'>  
    {tasks.length > 0 && (  
      <div className='tasks'>  
        {tasks.map(task => <TaskItem key={task._id} task={task} />)}  
      </div>  
    )}  
  </section>  
)
```

A screenshot of the Visual Studio Code interface showing the `TaskList.js` file. The file content is as follows:

```
File Edit Selection View Go Run Terminal Help TaskList.js Chapter 10 Visual Studio Code
App.js M Dashboard.js U TaskList.js X TaskItem.js U taskService.js U
frontend/st/components/TaskList.js (32)
19 return (
20   isLoading ? <Spinner /> : (
21     <section className='content'>
22       {tasks.length > 0 && (
23         <div className='tasks'>
24           {tasks.map(task => <TaskItem key={task._id} task={task} />)}
25         </div>
26       )}
27     </section>
28   )
29 )
30 ]
31
32 export default TaskList
```

The code implements a `TaskList` component. It checks if `isLoading` is true and returns a `Spinner` component. Otherwise, it renders a `section` element with the class `'content'`. If there are tasks, it maps over them to render a `TaskItem` component for each task. Finally, it exports the `TaskList` component.

Figure 10.14: Adding return logic in TaskList.js file

Next, create the TaskItem.js file in the components folder. Here, we are first importing the useDispatch hook from react-redux, which we are going to use later while creating delete functionalities.

Inside the TaskItem function, we are first destructuring the props of the task. Next, we show the createdAt and text inside a div and h2, respectively. Also, notice that the createdAt is changed to the proper format using the new Date() and toLocaleString('en-US').

```
import { useDispatch } from 'react-redux'

const TaskItem = ({ task }) => {

  return (
    <div className='task'>
      <div>{new Date(task.createdAt).toLocaleString('en-US')}</div>
      <h2>{task.text}</h2>
    </div>
  )
}
```

```
export default TaskItem
```

A screenshot of the Visual Studio Code interface. The title bar shows "TaskItem.js · Chapter-10 · Visual Studio Code". The left sidebar has icons for file navigation, search, and other code-related functions. The main editor area displays the following code:

```
1 import { useDispatch } from 'react-redux'
2
3 const TaskItem = ({ task }) => {
4   return [
5     <div className='task'>
6       <div>new Date(task.createdAt).toLocaleString('en-US')</div>
7       <h2>{task.text}</h2>
8     </div>
9   ]
10 }
11
12 export default TaskItem
```

Figure 10.15: Creating TaskItem.js file

We will also update the TaskForm.js file to navigate to the/alltasks route and hence show the TaskList component. Here, we are first importing the useNavigate hook from react-router-dom.

After that, we use it to create a variable called navigate inside the TaskForm function. After that, in the onSubmit() method, which is called after the user clicks on the Add Tasks button, we add navigate() at the end to go to /alltasks.

Also, we are changing a bit of the text style of Enter Text using inline styling.

```
import { useNavigate } from 'react-router-dom'
```

```
const TaskForm = () => {
```

```
  const navigate = useNavigate()
```

```
  const onSubmit = (e) => {
```

```
    e.preventDefault()
```

```
    dispatch(createTask({ text }))
```

```
    setText('')
```

```
    navigate('/alltasks')
```

```
}

return (
  <section className='form'>
    <form onSubmit={onSubmit}>
      <div className='form-group'>
        <label htmlFor='text' style={{ fontWeight: 'bold' }}>Enter Task</label>
      </div>
    </form>
  </section>
)
}
```

File Edit Selection View Go Run Terminal Help TaskForm.js - Chapter 10 - Visual Studio Code

JS App.js M JS Dashboard.js U JS TaskForm.js U X JS TaskList.js U JS TaskItem.js U JS taskService.js U

frontend > src > components > JS TaskForm.js > TaskForm

```
1 import { useState } from 'react'
2 import { useDispatch } from 'react-redux'
3 import { createTask } from '../features/tasks/taskSlice'
4 import { useNavigate } from 'react-router-dom'

5
6 const TaskForm = () => {
7   const [text, setText] = useState('')
8   const dispatch = useDispatch()
9   const navigate = useNavigate()

10
11   const onSubmit = (e) => {
12     e.preventDefault()
13     dispatch(createTask({ text }))
14     setText('')
15     navigate('/alltasks')
16   }

17
18   return (
19     <section className='form'>
20       <form onSubmit={onSubmit}>
21         <div className='form-group'>
22           <label htmlFor='text' style={{ fontWeight: 'bold' }}>Enter Task</label>
23           <input type='text' id='text' value={text} onChange={e => setText(e.target.value)}>

```

master 0 0 ▲ 0 Lin 6, Col 7 Spaces: 4 UTF-8 CRLF {} JavaScript Go Live

Figure 10.16: Updating navigation in TaskForm.js file

Now, in the Chrome/Firefox browser, directly open <http://localhost:3000/alltasks> or click on the Check Tasks button to go there. Also, open the Redux devtools in the browser. Here, we will first see the pending case for getAll tasks.

In this state, since the API call has been going on, we will see the isLoading as true. The tasks will be an empty array, and the isError and isSuccess will be false. Also, the message will be an empty string.

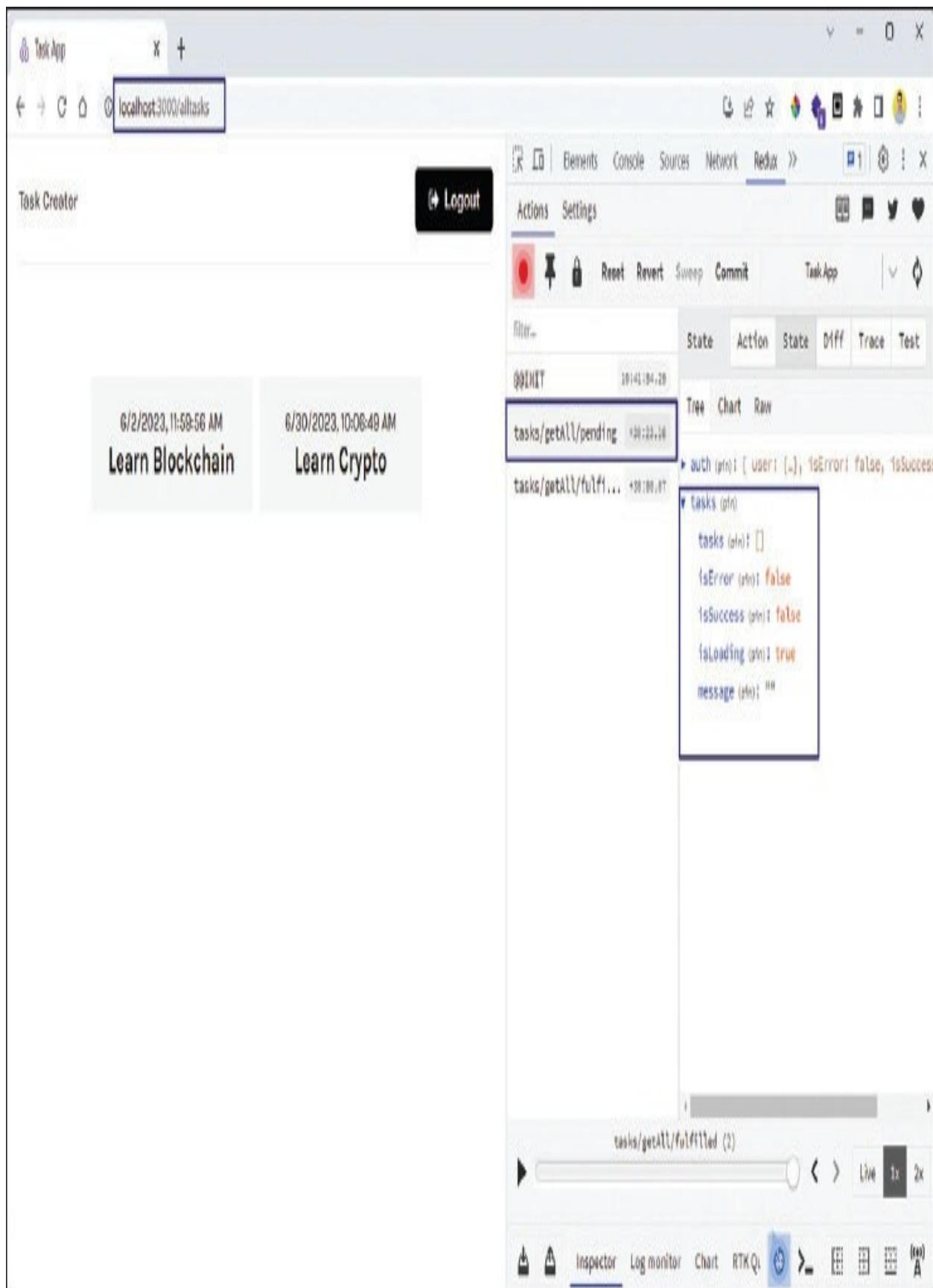


Figure 10.17: Pending state for all tasks

Once the API call is successful, we will see the tasks array is filled with data from the backend. It contains the text and the createdAt field, which we are showing in the browser.

Task App

localhost:3000/alrtasks

Logout

Task Creator

Learn Blockchain

6/2/2023, 11:59:56 AM

Learn Crypto

6/30/2023, 10:06:49 AM

Actions Settings

Reset Revert Swap Commit TaskApp

State Action State Diff Trace Test

@INIT 10141134.19

tasks/getAll/pending +10:13.18

tasks/getAll/fulfilled +10:10.07

Tree Chart Raw

tasks (2)

0 (2)

_id (str): "64798c4389011439d8b460a"
text (str): "Learn Blockchain"
user (str): "6478870aff59c7ed7cbf5671"
createdAt (str): "2023-06-02T06:29:56.978Z"
updatedAt (str): "2023-06-02T06:29:56.978Z"
__V (str): 0

1 (2)

_id (str): "640e5be1139d264df77fe01"
text (str): "Learn Crypto"
user (str): "6478870aff59c7ed7cbf5671"
createdAt (str): "2023-06-30T04:13:49.349Z"
updatedAt (str): "2023-06-30T04:13:49.349Z"
__V (str): 0

isError (bool): false
isSuccess (bool): true

tasks/getAll/fulfilled (2)

Inspector Log monitor Chart RTKQuery

Figure 10.18: Fulfilled state for all tasks

Now, back in `http://localhost:3000/`, enter a task and click on the Add Task button. Notice that when we came here, the reset case was running, and it made all states its initial value.

Task App X +

localhost:3000

Task Creator Logout Actions Settings

Welcome Nabendu

Check Tasks

Enter Task

Learn Rust

Add Task

Redux

TaskApp

filter... State Action State Diff Trace Test

INIT 18/41/2024, 20 tasks/getAll/pending 18/10/2024, 18 tasks/getAll/fulfilled 18/10/2024, 17 task/reset 18/10/2024, 17

Tree Chart Raw

auth (state): { user: (), isError: false, isSuccess: true } tasks (state): { items: [Object], loading: false } task (state): { message: "" }

task/reset (3)

Inspector Logmonitor Chart RTKQuery

```
task/reset (3)
  1. auth(state): { user: (), isError: false, isSuccess: true }
  2. tasks(state): { items: [Object], loading: false }
  3. task(state): { message: "" }
```

Figure 10.19: Reset before adding tasks

On clicking on the Add Task, we are taken to <http://localhost:3000/alltasks> and the new task is added. Also, notice the new task is added in the tasks state in Redux devtools.

Task App X +

localhost:3000/altasks

Task Creator Logout

Actions Settings

Reset Revert Swap Commit TaskApp

File	State	Action	State	Diff	Trace	Test
@INIT	2023-06-30T05:15:01.818Z					
tasks/getAll/pending	2023-06-30T05:15:10					
tasks/getAll/fulfilled	2023-06-30T05:15:17					
task/reset	2023-06-30T05:18					
tasks/create/pending	2023-06-30T05:18					
tasks/getAll/pending	2023-06-30T05:18					
tasks/create/fulfilled	2023-06-30T05:18					
tasks/getAll/fulfilled	2023-06-30T05:18					

Tree Chart Raw

auth (obj): { user: {}, isError: false, isSuccess: true }

* tasks (arr)

* task (arr)

* 0 (arr): { _id: "64798c6438e4", text: "Learn Blockchain", user: "6478870aff50c7ed7cbf567" }

* 1 (arr): { _id: "649e64d0139d204d77fc83", text: "Learn Crypto", user: "6478870aff50c7ed7cbf567" }

* 2 (arr): { _id: "649e64d0139d204d77fc83", text: "Learn Rust", user: "6478870aff50c7ed7cbf567" }

createdAt (obj): "2023-06-30T05:15:01.818Z"

updatedAt (obj): "2023-06-30T05:15:18.618Z"

_V (arr): 0

isError (bool): false

isSuccess (bool): true

isLoading (bool): false

message (str): ""

tasks/getAll/fulfilled (7)

Inspector Log monitor Chart RTKQuery

Figure 10.20: Added new task

Now, let's also see the tasks for another user. Here, we have logged in with another user's login and then clicked on the Check Tasks button.

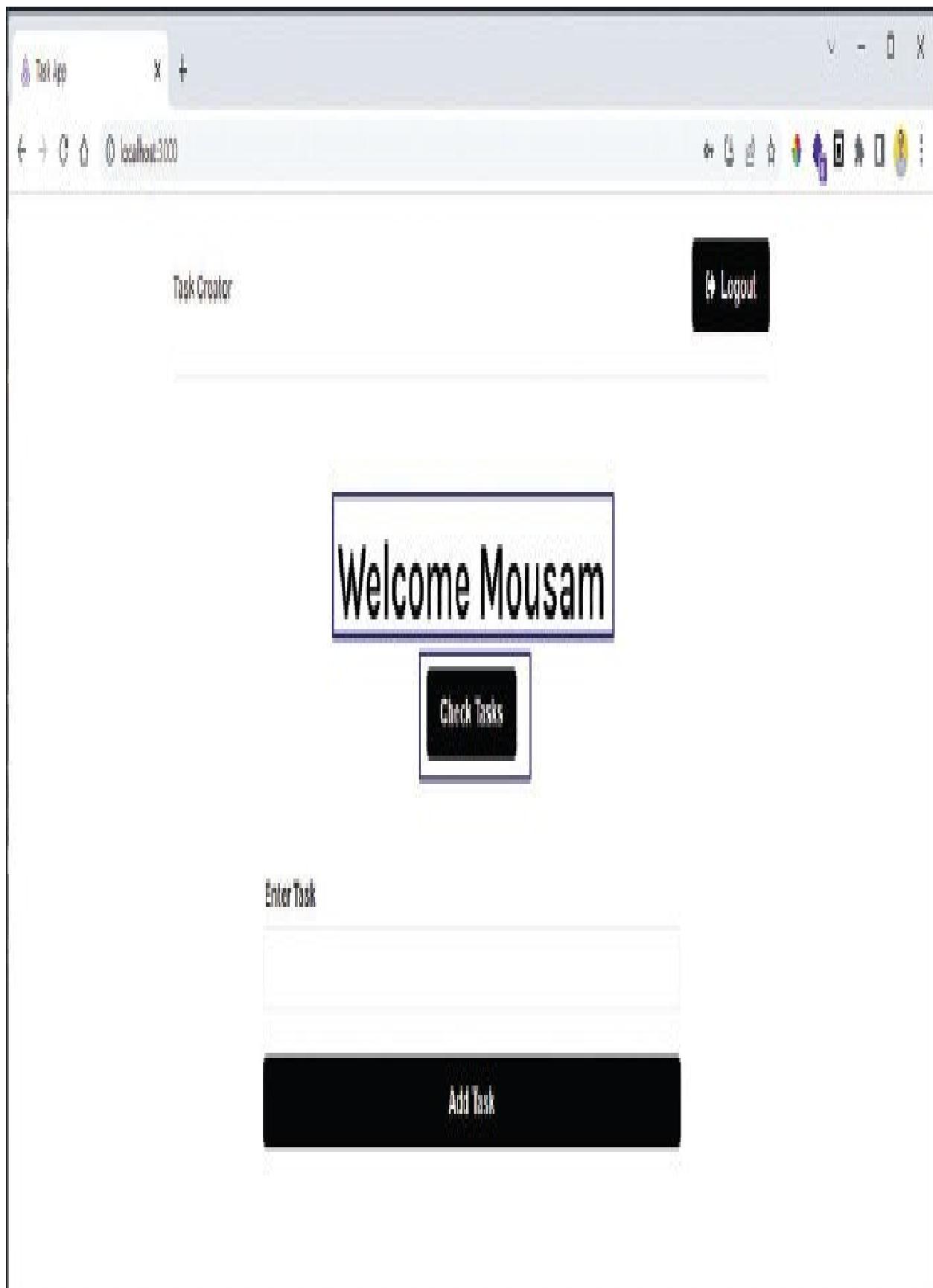


Figure 10.21: Checking tasks for different user

Here, notice that different tasks are shown for the user in the <http://localhost:3000/alltasks> link.

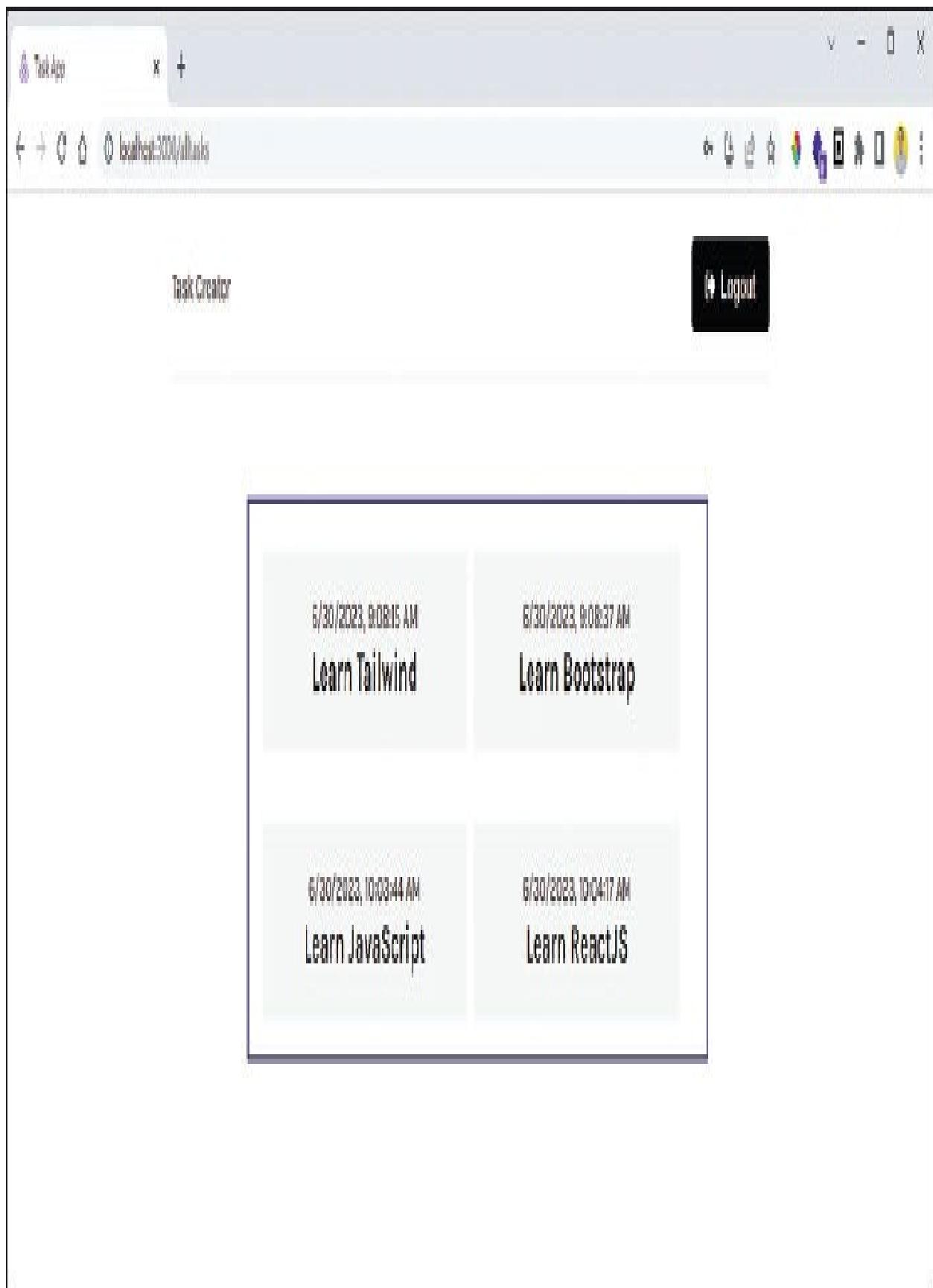


Figure 10.22: Tasks for different user

Deleting tasks

Now, we will add the last functionality in our project, which is to delete a task. So in the TaskItem.js file, we will first import the useDispatch hook from react-redux. Next, we are also importing the deleteTask from taskSlice, which we will create next.

Inside the TaskItem, we are creating a dispatch variable which will use the useDispatch. Next, we have added a button which will dispatch the deleteTask() with the task id.

```
import { useDispatch } from 'react-redux'

import { deleteTask } from '../features/tasks/taskSlice'

const TaskItem = ({ task }) => {

  const dispatch = useDispatch()

  return (
    <div className='task'>
      ...
      <button onClick={() => dispatch(deleteTask(task._id))}>
        X
      </button>
    </div>
  )
}
```

)

}

File Edit Selection View Go Run Terminal Help TaskItem.js - Task-MERN - Visual Studio Code

Dashboard.js X TaskItem.js X taskSlice.js X taskService.js X

frontend > src > components > TaskItem.js > TaskItem

```
1 import { useDispatch } from 'react-redux'
2 import { deleteTask } from '../features/tasks/taskSlice'
3
4 const TaskItem = ({ task }) => [
5   const dispatch = useDispatch()
6
7   return (
8     <div className='task'>
9       <div>{new Date(task.createdAt).toLocaleString('en-US')}</div>
10      <h2>{task.text}</h2>
11      <button onClick={() => dispatch(deleteTask(task._id))}>X</button>
12    </div>
13  )
14
15
16 export default TaskItem
```

Figure 10.23: Adding button to delete task

Now, in the taskSlice.js file, we will create a deleteTask variable. Here, we are again using the createAsyncThunk function.

In the createAsyncThunk function, the first parameter is the tasks/delete, which means from the tasks folder, take the delete function. Next, the parameter is an async arrow function which takes id and thunkAPI as a parameter.

Inside the function, we have a try...catch block. From the try block, we are first getting the token using the getState() method. Next, we call the deleteTask function from the taskService.js file with the id and the token and then return it.

In the catch part, we are simply catching different kinds of errors and returning them.

```
export const deleteTask = createAsyncThunk(
  'tasks/delete',
  async (id, thunkAPI) => {
    try {
      const token = thunkAPI.getState().auth.user.token
      return await taskService.deleteTask(id, token)
    } catch (error) {
```

```
const message =  
  (error.response && error.response.data && error.response.data.message)  
    || error.message  
    || error.toString()  
  
  return thunkAPI.rejectWithValue(message)  
}  
}  
)
```

File Edit Selection View Go Run Terminal Help taskSlice.js - task-MERN - Visual Studio Code

Dashboard.js U TaskItem.js U taskSlice.js X taskService.js U

```
frontend > src > features > tasks > taskSlice.js > deleteTask > createAsyncThunk(tasks/delete) callback
```

39 | return thunkAPI.rejectWithValue(message)
40 | }
41 | }
42 |
43 | export const deleteTask = createAsyncThunk(
44 | 'tasks/delete',
45 | async (id, thunkAPI) => {
46 | try {
47 | const token = thunkAPI.getState().auth.user.token
48 | return await taskService.deleteTask(id, token)
49 | } catch (error) {
50 | const message =
51 | (error.response && error.response.data && error.response.data.message)
52 | || error.message
53 | || error.toString()
54 | return thunkAPI.rejectWithValue(message)
55 | }
56 | }
57 | }
58 |
59 |
60 | export const taskSlice = createSlice({
61 | name: 'task',
62 | initialState,

master 0 0 ▲ 0 In 50, Col 22 Spaces: 2 UTF-8 CRLF {} JavaScript Go Live

Figure 10.24: deleteTask function in taskSlice.js file

Back in the taskService.js file, we have created a deleteTask function which takes the id and token as a parameter. This is the function which we are calling from our taskSlice.js file.

Inside the function, we have a config object first, similar to the one created in the getTasks function earlier in the chapter.

Next, we are doing a DELETE call to the API_URL plus id with the config. The result of the api call which we get in response.data and returning back the response from the function. Lastly, we are exporting the deleteTask function.

```
const deleteTask = async (id, token) => {  
  const config = {  
    headers: {  
      Authorization: `Bearer ${token}`,  
    },  
  };  
  
  const response = await axios.delete(API_URL + id, config)  
  
  return response.data  
}
```

```
const taskService = { createTask, getTasks, deleteTask }
```

A screenshot of Visual Studio Code showing the file `taskService.js`. The code defines a service for managing tasks using Axios. It includes functions for creating, getting, and deleting tasks.

```
File Edit Selection View Go Run Terminal Help taskService.js Task MERN - Visual Studio Code
Dashboard.js TaskItem.js taskSlice.js taskService.js X
frontend > src > features > tasks > taskService.js (8) deleteTask (8) config
67
25 const deleteTask = async (id, token) => {
26   const config = [
27     headers: {
28       Authorization: `Bearer ${token}`,
29     }
30   ]
31   const response = await axios.delete(API_URL + id, config)
32   return response.data
33 }
34
35 const taskService = [ createTask, getTasks, deleteTask ]
36
37 export default taskService
38
```

Figure 10.25: deleteTask function in taskService.js file

Now, in our taskSlice.js file in the taskSlice function, we will add the new cases in the extraReducers.

In our code, we have three cases for pending, fulfilled and rejected cases. Since we are doing an API DELETE call through our deleteTask function, we have three states in it. When the API call starts, we have a pending state. In the pending state, isLoading will be made to true.

The next state is the fulfilled case, which will happen if our API call is successful. Here, we are making the isLoading as false and isSuccess as true and then filtering the tasks state to remove the passed task and update the tasks state.

If, for some reason, the API call is not successful, we have a rejected case. Here, we are making the isLoading as false, isError as true. We are also storing the error in the message state, which we will receive in the payload.

```
.addCase(deleteTask.pending, (state) => {  
    state.isLoading = true  
})  
.addCase(deleteTask.fulfilled, (state, action) => {  
    state.isLoading = false  
    state.isSuccess = true
```

```
state.tasks = state.tasks.filter(task => task._id !== action.payload.id)

})

.addCase(deleteTask.rejected, (state, action) => {

state.isLoading = false

state.isError = true

state.message = action.payload

})
```

File Edit Selection View Go Run Terminal Help taskSlice.js - Task-MDU - Visual Studio Code

Dashboard.js U TaskItem.js U taskSlice.js X taskService.js U

```
frontend > src > features > tasks > taskSlice.js > taskSlice.js > extraReducers > addCase() callback
  97     .addCase(deleteTask.rejected, (state, action) => {
  98       state.isLoading = false
  99       state.isError = true
 100       state.message = action.payload
 101   })
 102   .addCase(deleteTask.pending, (state) => {
 103     state.isLoading = true
 104   })
 105   .addCase(deleteTask.fulfilled, (state, action) => {
 106     state.isLoading = false
 107     state.isSuccess = true
 108     state.tasks = state.tasks.filter(task => task._id !== action.payload.id)
 109   })
 110 export const { reset } = taskSlice.actions
 111 export default taskSlice.reducer
```

Figure 10.26: extraReducers for delete in taskSlice.js file

Now, go back to <http://localhost:3000/alltasks>, and we will see the X button. We can see that there are four tasks here and also notice the task in the Redux devtools.

Task App

localhost:3000/alltasks

Logout

Task Creator

Learn Tailwind (6/30/2023, 9:08:15 AM)

Learn Bootstrap (6/30/2023, 9:08:37 AM)

Learn JavaScript (6/30/2023, 10:03:44 AM)

Learn ReactJS (6/30/2023, 10:04:17 AM)

Actions Settings

Reset Revert Sweep Commit TaskApp

INIT 10:54:27.19

tasks/getAll/pending +10:00.06

tasks/getAll/fulfilled +10:00.11

Tree Chart Raw

auth (err): { user: {}, isError: false, isSuccess: false }

tasks (arr)

tasks (arr)

0 (obj): { _id: "649e4e4e2710", text: "Learn Tailwind", user: "649e4e4e3d19", createdAt: "2023-06-30T04:17:17Z", updatedAt: "2023-06-30T04:17:17Z" }

1 (obj): { _id: "649e4e4e5b2813", text: "Learn Bootstrap", user: "649e4e4e3d19", createdAt: "2023-06-30T04:17:17Z", updatedAt: "2023-06-30T04:17:17Z" }

2 (obj): { _id: "649e4e4e5b2813", text: "Learn Java", user: "649e4e4e3d19", createdAt: "2023-06-30T04:17:17Z", updatedAt: "2023-06-30T04:17:17Z" }

3 (obj)

id (str): "649e4e4e5b49130d264df77fc82a"

text (str): "Learn ReactJS"

user (str): "649e4e4e3d19"

createdAt (str): "2023-06-30T04:17:17Z"

updatedAt (str): "2023-06-30T04:17:17Z"

isError (bool): false

isSuccess (bool): true

isLoading (bool): false

message (str): ""

tasks/getAll/fulfilled (2)

Inspector Log monitor Chart RTKQuery

Figure 10.27: All tasks for a user in localhost

Once we click on the X button in <http://localhost:3000/alltasks>, we will first see the pending case for delete tasks.

In this state, since the API call has been going on, we will see the isLoading as true. The isError will be false, and the isSuccess will be true. Also, the message will be an empty string. The tasks array will still contain four elements.

Task App X +

localhost:3000/altasks

Task Creator Logout

6/30/2023, 9:08:15 AM X Learn Tailwind

6/30/2023, 9:08:37 AM X Learn Bootstrap

6/30/2023, 10:03:44 AM X Learn JavaScript

Actions Settings

Reset Revert Sweep Commit TaskApp

filter... State Action State Diff Trace Test

INIT 16/04/27 19

tasks/getAll/pending -> 00:00:04

tasks/getAll/fulfilled -> 00:00:21

tasks/delete/pending -> 00:00:12

tasks/delete/fulfilled -> 00:00:04

tasks (9)

 tasks (9)

 0 (pending) { _id: "649e4e2719", text: "Learn Tailwind", u...
 1 (pending) { _id: "649e4e3d10", text: "Learn Bootstrap", u...
 2 (pending) { _id: "649e5b2813", text: "Learn Java", u...
 3 (pending) { _id: "649e5b4013", text: "Learn React", u...
 isError (pending) false
 isSuccess (pending) true
 isLoading (pending) true
 message (pending): ""

tasks/delete/fulfilled (4)

Live 1x 2x

Inspector Log monitor Chart RTKQuery

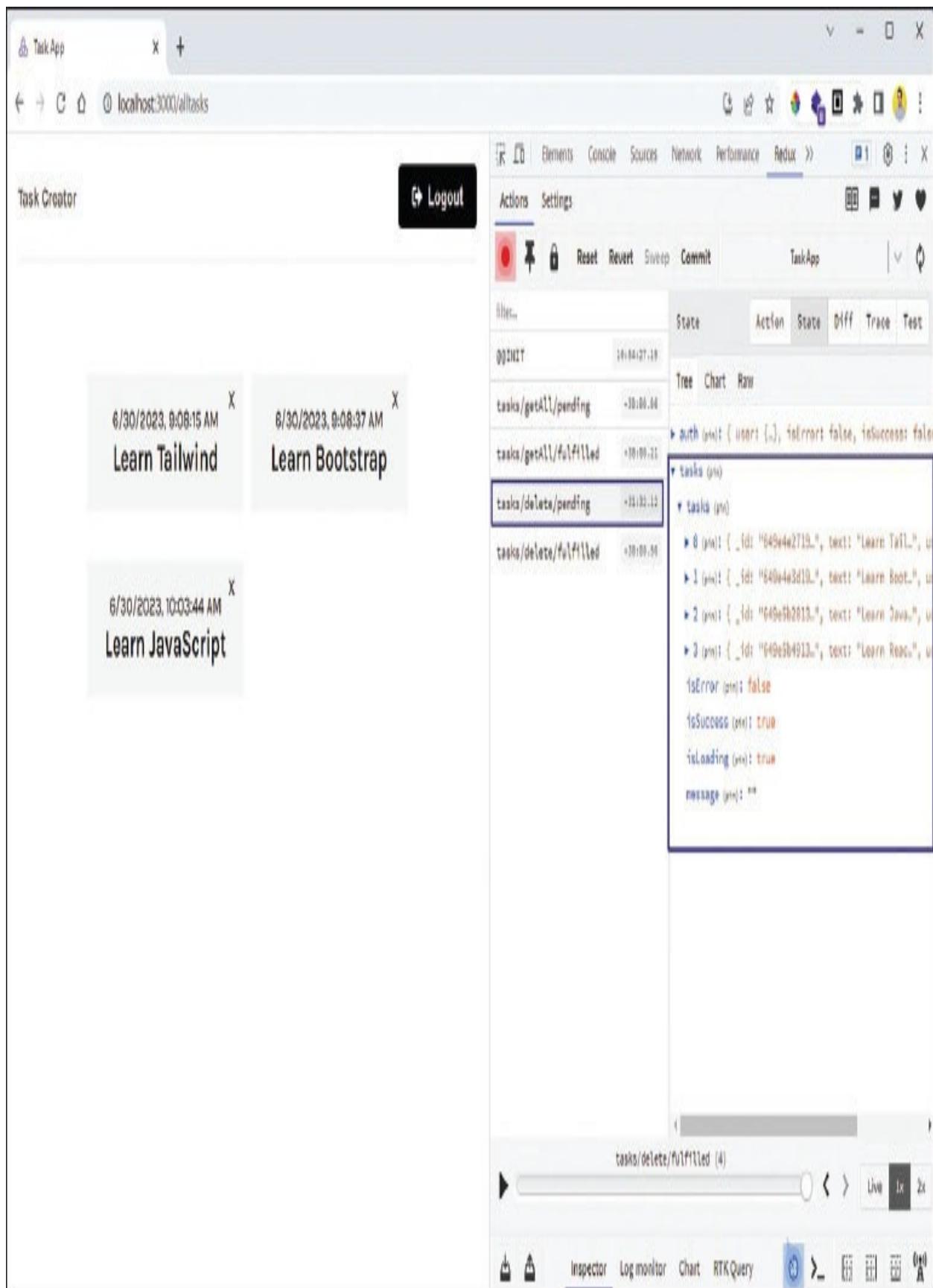


Figure 10.28: Pending state for delete in localhost

Once the API call is successful, we will see the fulfilled case for delete tasks. Also, the tasks state will now contain three elements.

In the browser also, we will see the Learn ReactJS element being deleted.

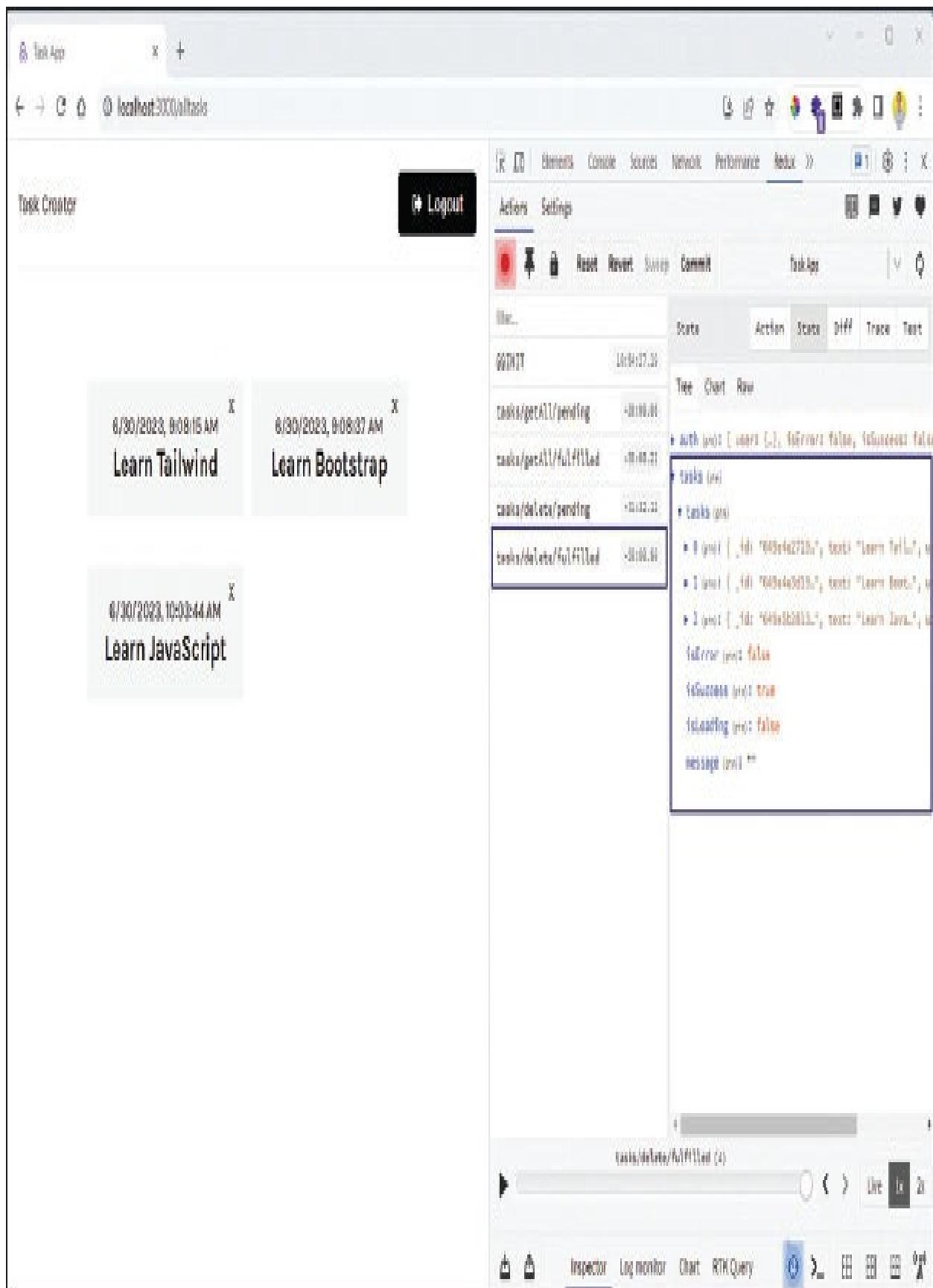


Figure 10.29: Fulfilled state for delete in localhost

Conclusion

In this chapter, we have learned about implementing Thunk in our Task workflow. Here, we have also fetched all tasks from the database. After that, we displayed all tasks in the front end.

Next, we have also learned to delete the task from the front end and database.

In the next chapter, we will test the front end made in React using Jest and React Testing Library.

Points to remember

Implementing tasks with async Thunk

Displaying all tasks on the front end

Deleting a task from the front end and database

CHAPTER 11

Frontend Testing

Introduction

This chapter covers the testing of our frontend using Jest and React Testing Library. These two libraries are in-built in React and help us do unit testing in our frontend code.

It's generally the developer's responsibility to do unit testing of code. Unit testing helps catch errors early, and in a lot of companies, the unit testing code coverage should be 80%. Unit testing involves some type of coding, which is again a developer domain. The testers generally do end-to-end testing using manual or automated testing, which is different than the unit testing done by developers.

Structure

In this chapter, we are going to discuss the following topics:

Setup testing in Frontend

Task Slice and Service test with Jest

Testing with React Testing Library

Configuring and Checking Coverage

Setup testing in Frontend

We don't need to do much setup for testing, as the react app created using create-react-app comes in-built with the testing library of Jest and also React Testing Library(RTL). We can check the same in package.json on our frontend.

“@testing-library/jest-dom”: “^5.16.5”,

“@testing-library/react”: “^13.4.0”,

“@testing-library/user-event”: “^14.4.3”

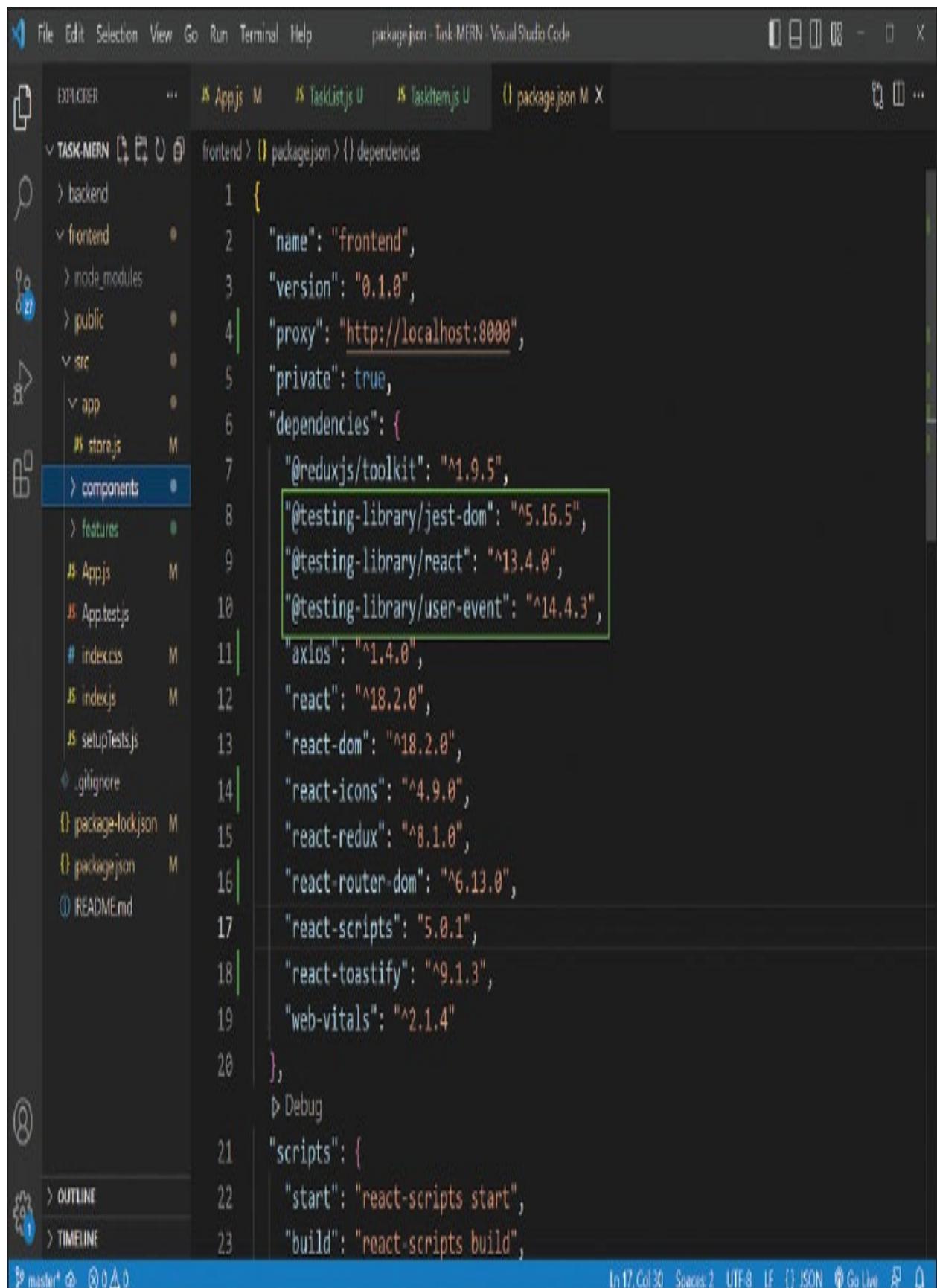


Figure 11.1: Checking React testing libraries

Now, from the terminal go to the frontend folder and run the npm test. It will fail since we have made a lot of changes in the App.js file.

```
C:\Windows\system32\cmd.exe v

naben@Nabendu MINGW64 ~ /Desktop/Task-MERN
$ cd frontend/
$ npm test

> frontend@0.1.0 test
> react-scripts test
FAIL  src/App.test.js
● Test suite failed to run

Jest encountered an unexpected token
```

Figure 11.2: Running tests

Now, we will remove the App.test.js file as it will cause errors in our tests.

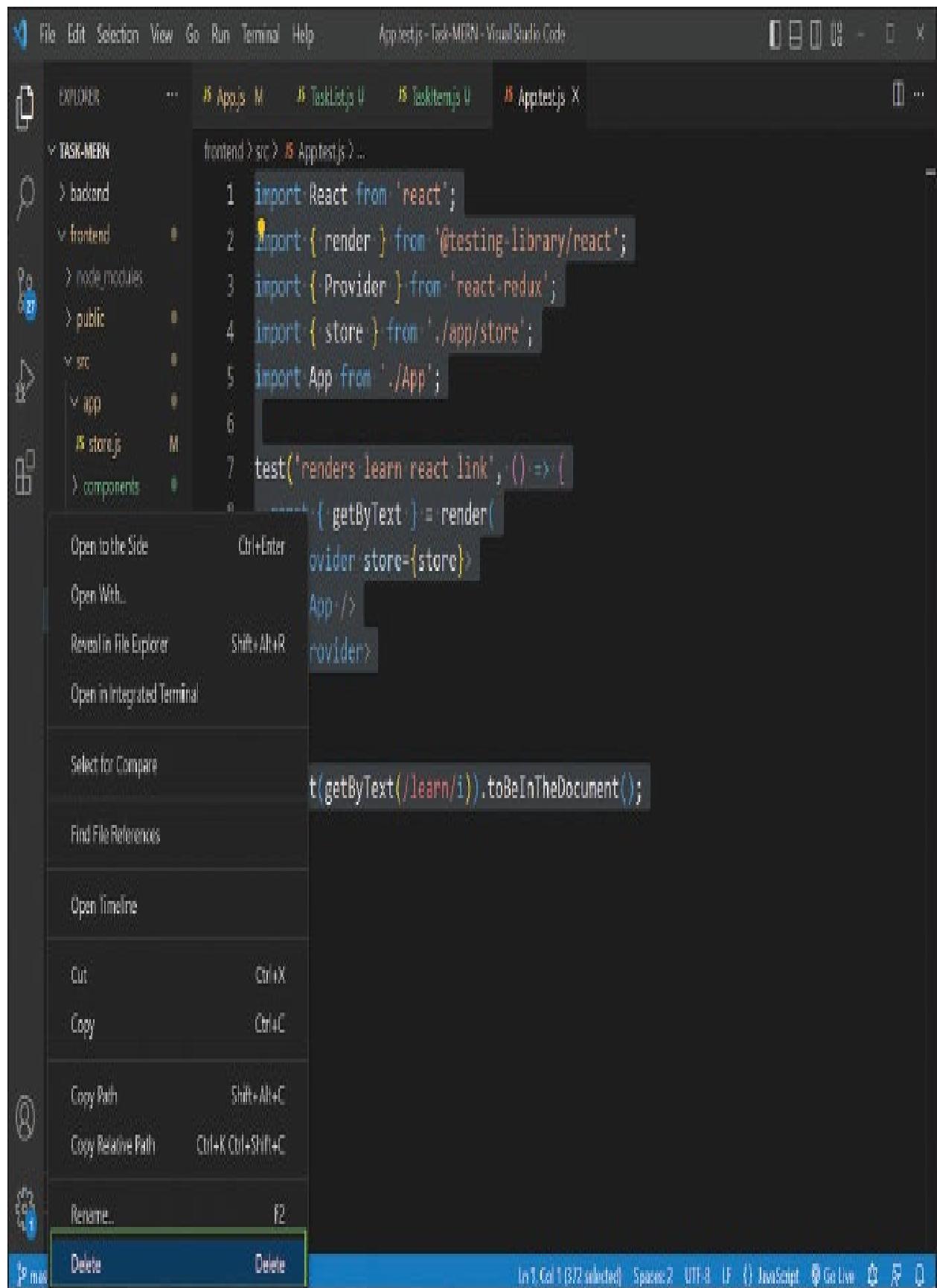


Figure 11.3: Deleting App.test.js file

We will write our first Jest test for the Spinner.js file. For this, we have also created a Spinner.test.js file. In Jest testing, we need to identify any element with something unique, but we don't have any in Spinner.js file. So, we have added the attribute of data-testid in both divs.

```
<div className='loadingSpinnerContainer' data-testid="spin-container">  
<div className='loadingSpinner' data-testid="inner-container" />  
</div>
```

File Edit Selection View Go Run Terminal Help Spiner.js - Task-MERN - Visual Studio Code

EXPLORER ... App.js M TaskList.js U TaskItem.js U Spinner.js U X SpinnerTest.js U

TASK-MERN frontend > src > components > Spinner.js > Spinner

> backend
> frontend
 > node_modules
 > public
 > src
 > app
 B story.js M
 > components
 B Dashboard.js U
 B Header.js U
 B Login.js U
 B Register.js U
 B Spinner.js U
 B SpinnerTest.js U
 B TaskForm.js U
 B TaskItem.js U
 B TaskList.js U
 > features
 B App.js M
 # index.css M
 # index.js M
 B layouts.js
 B package-lock.json M
 B ignore
(1) package.json M

> OUTLINE
> TIMELINE

```
1 const Spinner = () => {
  2   return (
    3     <div className='loadingSpinnerContainer' data-testid='spin-container'>
      4       <div className='loadingSpinner' data-testid='inner-container' />
    5     </div>
  6   )
  7 }
  8
 9 export default Spinner
```

In 4, Col 57 Source: 2 UTF-8: CRLF {} JavaScript Go Live ⚡ ⚡ ⚡

Figure 11.4: Adding data-testid in Spinner.js file

Finally, we will write our first test in the Spinner.test.js file. Here, we will first import the render and screen from the testing library. We will also import the Spinner component.

In jest testing, we have a ‘describe’ block wrapping all of our tests. Here, the first parameter is a string and the second is an arrow function. Next, we have a single test. Here, also the first parameter is a string which is “render correctly” in our case. The second parameter is an arrow function in which we will write the test statements.

Here, we are first rendering the Spinner component with the render method. Next, we created a variable called containerDiv, and with screen.getByTestId() got the first div with the name spin-container. Here, we are testing it with the expect() function and toBeInTheDocument().

We are repeating the same for the inner-container.

```
import { render, screen } from "@testing-library/react";
import Spinner from "./Spinner";

describe("Spinner", () => {
  test("render correctly", () => {
```

```
render(<Spinner />);

const containerDiv = screen.getByTestId("spin-container");
expect(containerDiv).toBeInTheDocument();

const innerDiv = screen.getByTestId("inner-container");
expect(innerDiv).toBeInTheDocument();

})
```

A screenshot of the Visual Studio Code (VS Code) interface. The title bar shows "SpinnerTest.js · Task MERN · Visual Studio Code". The left sidebar has icons for file, edit, selection, view, go, run, terminal, and help. The tabs at the top show "App.js M", "TaskList.js U", "TaskItem.js U", "Spinner.js U", and "SpinnerTest.js U X". The main editor area displays a Jest test file for a "Spinner" component:

```
frontend > src > components > Spinner.test.js > describe("Spinner") > test("render correctly") > callback
```

```
1 import { render, screen } from "@testing-library/react";
2 import Spinner from "./Spinner";
3
4 describe("Spinner", () => {
5   test("render correctly", () => [
6     render(<Spinner />);
7
8     const containerDiv = screen.getByTestId("spin-container");
9     expect(containerDiv).toBeInTheDocument();
10
11    const innerDiv = screen.getByTestId("inner-container");
12    expect(innerDiv).toBeInTheDocument();
13  ]
14})
```

Figure 11.5: Writing test in Spinner.test.js file

Now, on running the tests with npm test, the test in Spinner.test.js will pass.

```
PASS  src/components/Spinner.test.js

  Spinner
    ✓ render correctly (31 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:  0 total
Time:        1.975 s
Ran all test suites related to changed files.

Watch Usage
> Press a to run all tests.
> Press f to run only failed tests.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.
```

Figure 11.6: Running successful test from Spinner.test.js

Task Slice and Service Test with Jest

We will now test the taskSlice.js and taskService.js files. The tests for these components are a bit complicated because we need to test them using mock data.

For this, we will install the library of axios-mock-adapter with the following command. This library helps us to mock axios calls. Note that we are installing the same dev dependency as we don't need it in production.

```
npm i -D axios-mock-adapter
```

```
raber@laptop: ~ /Desktop/Fast-MERN/frontend (master)
```

```
$ npm i -D axios-mock-adapter
```

```
npm WARN deprecated request@2.88.2: Package no longer supported. Contact support@nodejs.org for more info.
```

```
added 4 packages, and audited 153 packages in 6s
```

```
236 packages are looking for funding
```

```
run 'npm fund' for details
```

```
4 vulnerabilities (3 moderate, 1 high)
```

```
To address issues that do not require attention, run:
```

```
npm audit fix
```

Figure 11.7: Installing axios-mock-adapter package

Next, we will install redux-mock-store in our frontend app. This library will mock the redux store in our app. We will install it with the following command:

```
npm i redux-mock-store
```

```
fabien@Fabienou-MacBook-Pro:~/Desktop/Tek4-VERN/-frontend (master)
```

```
$ npm i redux-mock-store
```

added 2 packages, and audited 1525 packages in 3s

236 packages are looking for funding
run 'npm fund' for details

⚠ 4 vulnerabilities (33 moderate, 6 high)

To address issues that do not require attention, run:

```
npm audit fix
```

To address all issues (including breaking changes), run:

```
npm audit fix --force
```

Figure 11.8: Installing redux-mock-store package

We also need to add a new .babelrc in the root directory of the frontend, and add the content for presets in it. Besides this, we need to install babel-jest and @babel/preset-env as dev dependency.

```
npm install --save-dev babel-jest @babel/preset-env
```

```
{  
  "presets": [  
    "@babel/preset-env"  
  ]  
}
```

The screenshot shows a dark-themed code editor interface with a sidebar on the left displaying project structure and a main editor area on the right.

File Explorer:

- Frontend
- App.js
- taskService.js
- taskStorage.js
- taskRouter.js
- babelrc
- package.json

Editor Area:

```
1 {
  2   "presets": [
  3     "babel/preset-env"
  4   ]
  5 }
```

Terminal:

```
root@localhost:~/Desktop/Task-ES6/frontend [master]
$ npm install --save-dev babel-jest babel/preset-env
added 25 packages, removed 4 packages, changed 14 packages, and audited 2546 packages in 9s

336 packages are looking for funding
  run 'npm fund' for details

  40 vulnerabilities (34 moderate, 6 high)

To address issues that do not require attention, run
  npm audit fix

To address all issues (including breaking changes), run
  npm audit fix --force
```

Bottom Bar:

ln4 Col8 Space4 Alt8 Ctrl8 ⌘ Esc with Comments ⌘ Color ⌘ ⌘

Figure 11.9: Changing babel configurations

We also need to do a setup in the package.json file. Here, we are using babel-jest to transform the js or jsx files. And we are also ignoring the node_modules and axios.

```
«jest»: {  
  “transform”: {  
    “^.+\\.jsx?$”: “babel-jest”  
  },  
  “transformIgnorePatterns”: [  
    “node_modules/(?!axios)/*”  
  ]  
}
```

A screenshot of the Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a tab for package.json. The title bar shows package.json - Live Preview - Visual Studio Code. The left sidebar lists files: App.js (M), tsService.js (U), tsMock.js (U), tsJestSetup.js (U), and package.json (X). The main editor area displays the package.json file with line numbers 39 to 62. A green rectangular selection highlights the Jest configuration starting from line 53. The bottom status bar shows icons for Undo, Redo, Save, Find, Replace, Insert, Cut, Copy, Paste, and Delete.

```
39  "jest": {}  
40  "transform": [  
41    "^.+\\.jsx?$": "babel-jest"  
42  ],  
43  "development": [  
44    "last 1 chrome version",  
45    "last 1 firefox version",  
46    "last 1 safari version"  
47  ],  
48 },  
49 "devDependencies": {  
50   "@babel/preset-env": "^7.12.7",  
51   "babel-jest": "^29.6.1"  
52 },  
53 "jest": {},  
54 "transform": [  
55   "^.+\\.jsx?$": "babel-jest"  
56 ],  
57 "transformIgnorePatterns": [  
58   "node_modules/(?!axios)/*"  
59 ],  
60 },  
61 },  
62 }
```

Figure 11.10: Adding babel config in package.json file

Now, we will create a taskSlice.test.js file inside the tasks folder. Here, we will first import configureMockStore from redux-mock-store, which will help us to mock redux. Next, we will import thunk middleware from redux-thunk.

We will import the MockAdapter from axios-mock-adapter which will help us mock axios. Next, we will also import axios which is used to do API calls in React. We will also import getTasks and taskService.

Lastly, we will create a variable of mockStore and use the configureMockStore and pass thunk in it. And we will also create a variable of mock and use the MockAdapter and pass axios in it.

```
import configureMockStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import MockAdapter from 'axios-mock-adapter';
import axios from 'axios';
import { getTasks } from './taskSlice';
import taskService from './taskService';

const mockStore = configureMockStore([thunk]);
const mock = new MockAdapter(axios);
```


A screenshot of the Visual Studio Code interface. The top bar shows the menu (File, Edit, Selection, View, Go, Run, Terminal, Help) and the title bar indicates the file is "taskService.js - Task MERN - Visual Studio Code". The main area displays the following code:

```
1 import configureMockStore from 'redux-mock-store';
2 import thunk from 'redux-thunk';
3 import MockAdapter from 'axios-mock-adapter';
4 import axios from 'axios';
5 import { getTasks } from './taskSlice';
6 import taskService from './taskService';
7
8 const mockStore = configureMockStore([thunk]);
9 const mock = new MockAdapter(axios);
```

Figure 11.11: Writing imports in taskSlice.test.js file

Next, in the taskSlice.test.js file, inside a describe block, we will first create a variable store. We will have beforeEach() and afterEach() functions. These functions run before and after every test.

In the beforeEach, we are using mockStore to mock our initial task state as per the taskSlice.js file. Also, we need to use a mock user token, as it is used in the taskSlice.js file.

In the afterEach, we are resetting the mock and the store.

```
describe('taskSlice', () => {  
  let store;  
  
  beforeEach(() => {  
    store = mockStore({  
      task: {  
        tasks: [],  
        isError: false,  
        isSuccess: false,  
        isLoading: false,  
        message: '',  
      },  
    });  
  });  
  
  it('should return the initial state', () => {  
    expect(store.getState()).toEqual({  
      task: {  
        tasks: [],  
        isError: false,  
        isSuccess: false,  
        isLoading: false,  
        message: '',  
      },  
    });  
  });  
});
```

```
},  
auth: {  
  user: { token: 'mock_token' },  
},  
});  
});  
afterEach(() => {  
  mock.reset();  
  store.clearActions();  
});  
});
```

A screenshot of Visual Studio Code showing an open file named `taskSlice.test.js`. The code is a Jest test for a `taskSlice` component. It uses a mock store to simulate state changes and an `auth` object with a user token. The test includes setup code, a test suite, and cleanup code.

```
File Edit Selection View Go Run Terminal Help taskSlice.test.js - TaskManager - Visual Studio Code
B App.js M B taskService.js U B taskSlice.js U B taskSlice.test.js X package.json M
frontend > src > features > tasks > taskSlice.test.js > describe('taskSlice') callback > beforeEach() callback > auth > user
18
11  describe('taskSlice', () => {
12    let store;
13    beforeEach(() => {
14      store = mockStore({
15        task: {
16          tasks: [],
17          isError: false,
18          isSuccess: false,
19          isLoading: false,
20          message: '',
21        },
22        auth: {
23          user: { token: 'mock_token' },
24        },
25      });
26    });
27
28    afterEach(() => {
29      mock.reset();
30      store.clearActions();
31    });
32  });
33
```

Figure 11.12: Writing describe in taskSlice.test.js file

Now, we will write our test in the taskSlice.test.js file. Here, we have a test function, where we first use the mock_token. Next, we have taken a dummy task which is an array of objects. Here, we have given some dummy fields.

Next, we created a constant variable called getTaskSpy. Here, using the jest.spyOn() which takes the first parameter as taskService and the second as getTasks. This should be resolved to the tasks.

Next, we are dispatching the getTasks() from taskSlice.js file. Finally, we are expecting the getTasksSpy to be called using the token.

```
test(<calls the taskService to fetch tasks>, async () => {  
  const token = 'mock_token';  
  const tasks = [  
    {  
      _id: '649e4e271947362dc297436a',  
      text: 'Learn Tailwind',  
      user: '649023b41935f5557f8e7ca4',  
      createdAt: '2023-06-30T03:38:15.287Z',  
      updatedAt: '2023-06-30T03:38:15.287Z',  
    },  
  ];  
  const taskService = {  
    getTasks: jest.fn().mockResolvedValue(tasks),  
  };  
  const getTaskSpy = jest.spyOn(taskService, 'getTasks');  
  const result = await getTasks(token);  
  expect(result).toEqual(tasks);  
  expect(getTaskSpy).toHaveBeenCalledWith(token);  
});
```

```
    __v: 0  
}  
];  
  
const getTasksSpy = jest.spyOn(taskService,  
‘getTasks’).mockResolvedValue(tasks);  
  
await store.dispatch(getTasks());  
  
expect(getTasksSpy).toHaveBeenCalledWith(token);  
});
```

File Edit Selection View Go Run Terminal Help taskSlice.test.js - Task-MERN - Visual Studio Code

taskSlice.js M taskSlice.co.js U taskSlice.test.js X package.json M

frontend > src > features > tasks > taskSlice.test.js > describe(taskSlice) callback > test(calls the taskService to fetch tasks) callback > tasks

```
30     store.clearActions();
31 );
32
33 test('calls the taskService to fetch tasks', async () => {
34     const token = 'mock_token';
35     const tasks = [
36         [
37             _id: '649e4e27194736dc297436a',
38             text: 'Learn Tailwind',
39             user: '649023b41935f5557f8e7ca4',
40             createdAt: '2023-06-30T03:38:15.287Z',
41             updatedAt: '2023-06-30T03:38:15.287Z',
42             __v: 0
43         ],
44     ];
45
46     const getTasksSpy = jest.spyOn(taskService, 'getTasks').mockResolvedValue(tasks);
47     await store.dispatch(getTasks());
48     expect(getTasksSpy).toHaveBeenCalledWith(token);
49 });
50 });
51
```

Figure 11.13: Writing tests in taskSlice.test.js file

Now, on running the test with npm test our taskSlice.test.js file will pass.

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/frontend/src/features/tasks (master)
```

```
$ npm test
```

```
> frontend@0.1.0 test
```

```
> react-scripts test
```

```
PASS  src/features/tasks/taskSlice.test.js
```

```
PASS  src/components/Spinner.test.js
```

```
Test Suites: 2 passed, 2 total
```

```
Tests:    2 passed, 2 total
```

```
Snapshots: 0 total
```

```
Time:    2.569 s
```

```
Ran all test suites related to changed files.
```

Watch Usage

- › Press a to run all tests.
- › Press f to run only failed tests.
- › Press q to quit watch mode.
- › Press p to filter by a filename regex pattern.
- › Press t to filter by a test name regex pattern.
- › Press Enter to trigger a test run.

Figure 11.14: Running tests for taskSlice.test.js file

Now, we will write the test for the task service component. So, create a taskService.test.js file inside the tasks folder. Here, we are first importing axios for API calls. Then we are importing MockAdapter for mocking axios. Lastly, we are importing the taskService component. We also created a mock variable using MockAdapter and passed the axios.

```
import axios from 'axios';
import MockAdapter from 'axios-mock-adapter';
import taskService from './taskService';

const mock = new MockAdapter(axios);
```

A screenshot of a dark-themed code editor, likely Visual Studio Code, showing a file named 'taskService.js'. The file contains the following code:

```
1 import axios from 'axios';
2 import MockAdapter from 'axios-mock-adapter';
3 import taskService from './taskService';
4
5 const mock = new MockAdapter(axios);
```

Figure 11.15: Writing imports for taskService.test.js file

Now, we will create a describe block in the taskService.test.js file. Here, we have first the afterEach() which will reset the mock after every test. Next, inside the test, we first have a token constant of token. Then we have a tasks array with an object containing a task.

Now, we are using the mock.onGet() to hit the API endpoint of /api/tasks using the token. Here, we expect the reply to have 200 and the tasks array. Next, we are calling the taskService file getTasks function using the token. The response is expected to be the tasks array.

```
describe('taskService', () => {
  afterEach(() => {
    mock.reset();
  });

  test(<fetches tasks successfully>, async () => {
    const token = 'mock_token';
    const tasks = [
      {
        _id: '649e4e271947362dc297436a',
        text: 'Learn Tailwind',
      }
    ];
    const response = await axios.get('/api/tasks', { headers: { Authorization: `Bearer ${token}` } });
    expect(response.status).toBe(200);
    expect(response.data).toEqual(tasks);
    const result = await taskService.getTasks(token);
    expect(result).toEqual(tasks);
  });
});
```

```
    user: '649023b41935f5557f8e7ca4',  
    createdAt: '2023-06-30T03:38:15.287Z',  
    updatedAt: '2023-06-30T03:38:15.287Z',  
    __v: 0  
};  
];
```

```
mock.onGet('/api/tasks', { headers: { Authorization: `Bearer ${token}` } }).reply(200, tasks);  
  
const response = await taskService.getTasks(token);  
  
expect(response).toEqual(tasks);  
});  
});
```

A screenshot of Visual Studio Code showing a test file for a task service. The file is named `taskService.test.js` and is located in the `src/features/tasks` directory. The code uses Mocha and Chai for testing, and it includes a mock implementation of the `taskService` using the `sinon` library.

```
File Edit Selection View Go Run Terminal Help taskService.test.js Task MERN Visual Studio Code

App.js M taskService.js U taskService.test.js X taskSlice.js U taskSlice.test.js U

frontend > src > features > tasks > taskService.test.js > describe('taskService') callback

7 describe('taskService', () => {
8   afterEach(() => {
9     mock.reset();
10  });
11
12 test('fetches tasks successfully', async () => {
13   const token = 'mock_token';
14   const tasks = [
15     {
16       _id: '649e4e271947362dc297436a',
17       text: 'learn Tailwind',
18       user: '649e23b41935f5557f8e7ca4',
19       createdAt: '2023-06-30T03:38:15.187Z',
20       updatedAt: '2023-06-30T03:38:15.187Z',
21       __v: 0
22     }
23   ];
24
25   mock.onGet('/api/tasks/', { headers: { Authorization: `Bearer ${token}` } }).reply(200, tasks);
26   const response = await taskService.getTasks(token);
27   expect(response).toEqual(tasks);
28 });
29 });

master ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬ ⑭ ⑮ ⑯ ⑰ ⑱ ⑲ ⑳
```

Figure 11.16: Writing test for taskService.test.js file

On running the npm test again, the test in taskService.test.js file passes.

```
Windows PowerShell      X  C:\WINDOWS\system32\cmd.exe X  C:\WINDOWS\system32\cmd.exe X + ✎
PASS  src/features/tasks/taskService.test.js
PASS  src/features/tasks/taskSlice.test.js
PASS  src/components/Spinner.test.js

Test Suites: 3 passed, 3 total
Tests:    3 passed, 3 total
Snapshots: 0 total
Time:     2.753 s
Ran all test suites related to changed files.

Watch Usage
> Press a to run all tests.
> Press f to run only failed tests.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.
```

Figure 11.17: Running test for taskService.test.js file

Testing with React Testing Library

In this section, we will test the Task Item and Store components. So, first, create a file TaskItem.test.js in the components folder and add the following code in it. Here, we will first import the render and Provider. Next, we are importing the configureMockStore from redux-mock-store, which will help us to mock redux.

We are also importing the component of TaskItem for testing. Now, inside a describe block, we have first created a task, with dummy data.

Lastly, we will create a variable of mockStore and use the configureMockStore and pass an empty array in it. And we will also create a variable of the store and call the mockStore.

```
import { render } from '@testing-library/react';

import { Provider } from 'react-redux';

import configureStore from 'redux-mock-store';

import TaskItem from './TaskItem';

describe('TaskItem', () => {

  const task = {

    _id: 'task-123',

    createdAt: '2023-07-09T10:00:00',
```

```
text: 'Learn Blockchain',  
};  
  
const mockStore = configureStore([]);  
const store = mockStore({});  
});
```

A screenshot of the Visual Studio Code interface showing a test file named `TaskItem.test.js`. The file is part of a project titled "Task-MERN". The code implements unit tests for the `TaskItem` component using the `@testing-library/react` library.

```
1 import { render } from '@testing-library/react';
2 import { Provider } from 'react-redux';
3 import configureStore from 'redux-mock-store';
4 import TaskItem from './TaskItem';
5
6 describe('TaskItem', () => [
7   const task = {
8     _id: 'task-123',
9     createdAt: '2023-07-09T10:00:00',
10    text: 'Learn Blockchain',
11  };
12
13  const mockStore = configureStore([]);
14  const store = mockStore({});
15
16 ]);
17
```

Figure 11.18: Writing imports & describe for TaskItem.test.js file

Next, we will add tests in the TaskItem.test.js file, inside the describe block. Here, we are rendering the TaskItem component, in which we are passing our dummy task. We are also wrapping it with the redux Provider. The redux provider is taking our mock store.

We are extracting the getByText from the render function, through which we are doing our tests. Now, we have two expected tests in which we are checking if the text and createdAt is in the document.

```
test(<renders task details correctly>, () => {  
  const { getByText } = render(  
    <Provider store={store}>  
      <TaskItem task={task} />  
    </Provider>  
  );  
  
  expect(getByText(task.text)).toBeInTheDocument();  
  expect(getByText(new Date(task.createdAt).toLocaleString('en-US'))).toBeInTheDocument();  
});
```

File Edit Selection View Go Run Terminal Help TaskItem.test.js • Task-MERN • Visual Studio Code

JS App.js M JS TaskList.js U JS TaskForm.js U JS TaskItem.js U X

frontend > src > components > TaskItem.test.js > ...

```
12
13 const mockStore = configureStore([]);
14 const store = mockStore({});
15
16 test('renders task details correctly', () => {
17   const { getByText } = render(
18     <Provider store={store}>
19       <TaskItem task={task} />
20     </Provider>
21   );
22
23   expect(getByText(task.text)).toBeInTheDocument();
24   expect(getByText(new Date(task.createdAt).toLocaleString('en-US'))).toBeInTheDocument();
25 });
26
27 });
28
```

Figure 11.19: Writing test for TaskItem.test.js file

On running the npm test again, the test in TaskItem.test.js file passes.

```
PASS  src/features/tasks/taskService.test.js
PASS  src/features/tasks/taskSlice.test.js
PASS  src/components/Spinner.test.js
PASS  src/components/TaskItem.test.js
```

Test Suites: 4 passed, 4 total

Tests: 4 passed, 4 total

Snapshots: 0 total

Time: 3.673 s

Ran all test suites related to changed files.

Watch Usage: Press w to show more.

Figure 11.20: Running test for TaskItem.test.js file

Now, we will write the tests for our store.js file. So, create a store.test.js file and add the following content in it. Here, we are first importing the configureStore from redux toolkit. Next, we are also importing the authReducer and taskReducer from the respective slice files.

Now, inside a describe block we are writing out a test to create the store with correct reducers. Here, again we have a store variable created with configureStore. Inside it we are passing the reducer with auth and tasks. Notice that it is similar to what we are doing in the store.js file.

Next, we get the state of the reducers with the getState() and storing it in a storeReducers variable. Finally, we are checking whether this storeReducers have the properties of auth and tasks.

```
import { configureStore } from '@reduxjs/toolkit';

import authReducer from '../features/auth/authSlice';

import taskReducer from '../features/tasks/taskSlice';

describe('store', () => {

  test(<creates the store with the correct reducers>, () => {

    const store = configureStore({

      reducer: {
```

```
auth: authReducer,  
tasks: taskReducer  
}  
});  
  
const storeReducers = store.getState();  
expect(storeReducers).toHaveProperty('auth');  
expect(storeReducers).toHaveProperty('tasks');  
});  
});
```

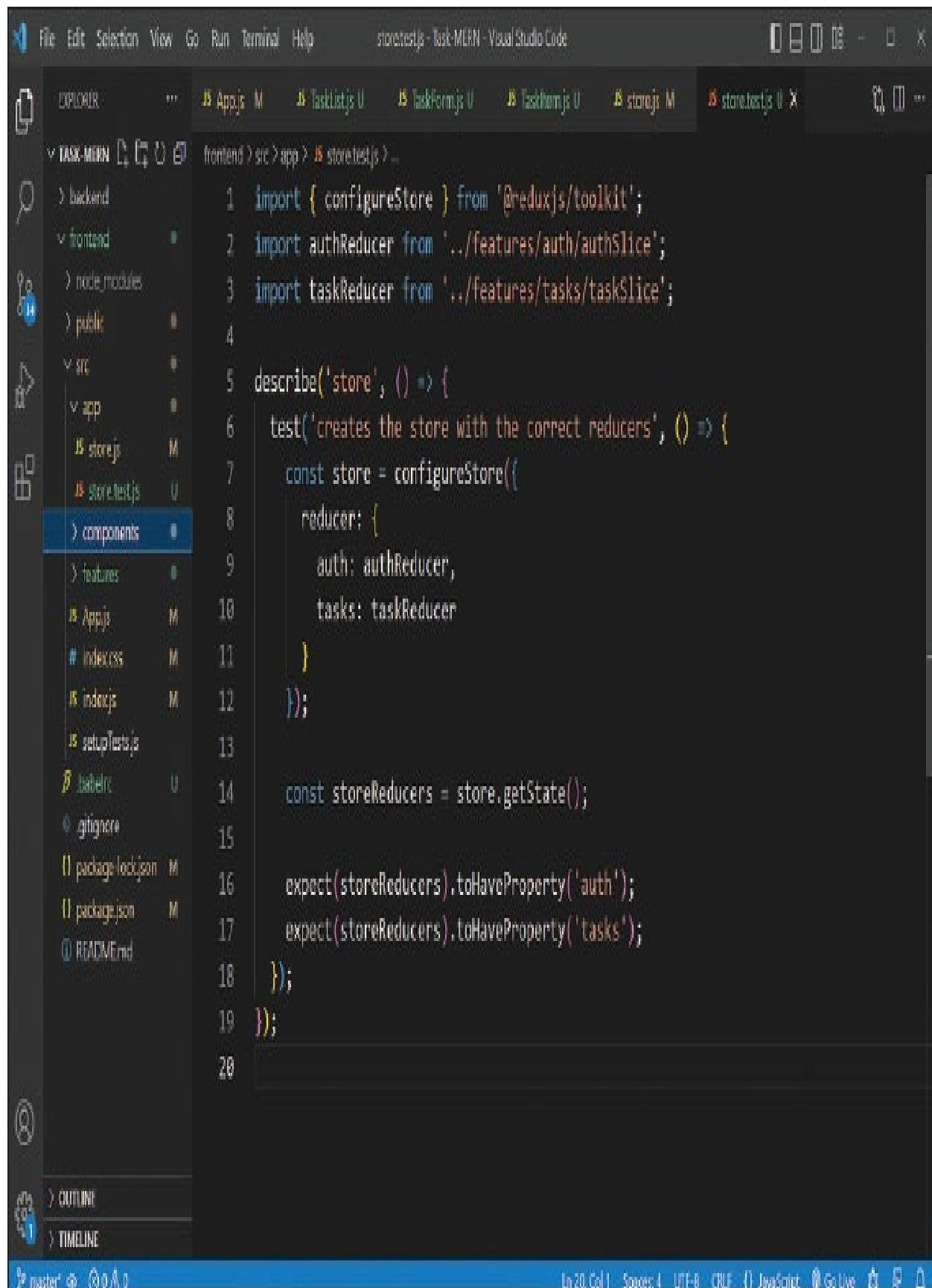


Figure 11.21: Writing test for store.test.js file

On running the npm test again, the test in store.test.js file passes.

```
Windows PowerShell      X  C:\WINDOWS\system32\cmd.exe X  C:\WINDOWS\system32\cmd.exe X + X
PASS  src/features/tasks/taskService.test.js
PASS  src/features/tasks/taskSlice.test.js
PASS  src/app/store.test.js
PASS  src/components/Spinner.test.js
PASS  src/components/TaskItem.test.js

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:  0 total
Time:        3.591 s
Ran all test suites related to changed files.

Watch Usage
> Press a to run all tests.
> Press f to run only failed tests.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.
```

Figure 11.22: Running test for store.test.js file

Now, we will write the tests for our index.js file. So, create an index.test.js file and add the following content in it. Here, we are first importing the required things, which include render, Provider, store, and the App component.

Next, we have written a simple test in which we are extracting a container variable from the render function. Inside the render, we are passing the App component, wrapped with the Provider component, where we are passing the store. This is similar to what we are doing in the index.js file.

Now, we have a variable called rootComponent in which we are storing the firstChild. This is actually the App component. Next, we are testing and expecting this rootComponent to be in the document.

Also, when we run this test with the npm run test, it will pass.

```
import React from 'react';
import { render } from '@testing-library/react';
import { Provider } from 'react-redux';
import { store } from './app/store';
import App from './App';
```

```
test('renders the root component correctly', () => {
  const { container } = render(
    <Provider store={store}>
      <App />
    </Provider>
  );
  const rootComponent = container.firstChild;
  expect(rootComponent).toBeInTheDocument();
});
```

File Edit Selection View Go Run Terminal Help Indextest.js · Task-MERN · Visual Studio Code

EXPLORER ⋮ index.js U TaskForm.js U TaskItem.js U store.js M store.test.js U index.js M indextest.js U X ⋮

✓ TASK-MERN frontend > src > indextest.js > test('renders the root component correctly') callback

> backend
< frontend
> node_modules
> public
< src
> app
> components
> features
JS App.js M
index.css M
JS index.js M
JS index.test.js U
JS setupTests.js
.babelrc U
↳ .gitignore
{} package-lock.json M
{} package.json M
① README.md

```
1 import React from 'react';
2 import { render } from '@testing-library/react';
3 import { Provider } from 'react-redux';
4 import { store } from './app/store';
5 import App from './App';
6
7 test('renders the root component correctly', () => {
8   const { container } = render(
9     <Provider store={store}>
10       <App />
11     </Provider>
12   );
13
14   const rootComponent = container.firstChild;
15
16   expect(rootComponent).toBeInTheDocument();
17 });
18
```

Figure 11.23: Writing test for index.test.js file

Configuring and Checking Coverage

In this section, we will learn to configure our frontend app to include coverage. After that, we will check the coverage for our app. In a lot of production scenarios, it is expected to provide 80% coverage.

First, in the package.json file, we will add the following line in scripts. Here, the test will be known as coverage and it will run the npm test with options of coverage and watchAll.

```
“coverage”: “npm test -- --coverage --watchAll”
```

File Edit Selection View Go Run Terminal Help package.json - Task MERN Visual Studio Code

App.js M package.json N X

frontend > package.json > scripts

```
17 "react-dom": "^18.2.0",
18 "react-icons": "^4.9.0",
19 "react-redux": "^8.1.0",
20 "react-router-dom": "^6.14.1",
21 "react-scripts": "5.0.1",
22 "react-toastify": "^9.1.3",
23 "redux-mock-store": "^1.5.4",
24 "web-vitals": "^2.1.4"
25 },
26 "scripts": [
27   "start": "react-scripts start",
28   "build": "react-scripts build",
29   "test": "react-scripts test",
30   "eject": "react-scripts eject",
31   "coverage": "npm test -- --coverage --watchAll"
32 ],
33 "eslintConfig": {
34   "extends": [
35     "react-app",
36     "react-app/jest"
37   ]
38 },
39 "browserslist": [
```

Figure 11.24: Adding coverage command in package.json file

After that, open a terminal and run the command `npm run coverage`. It will first test all the test files. We will notice that all of them passed.

naben@naben: MINGW64 ~/Desktop/Task-NERN/frontend (master)

```
$ npm run coverage
```

> frontend@0.1.0 coverage

> npm test -- --coverage --watchAll

> frontend@0.1.0 test

> react-scripts test --coverage --watchAll

PASS src/components/Spinner.test.js

PASS src/features/tasks/taskSlice.test.js

PASS src/components/TaskItem.test.js

PASS src/features/tasks/taskService.test.js

PASS src/app/store.test.js

PASS src/index.test.js

Figure 11.25: Running coverage command

Once the tests are done, we will get coverage for all files. The coverage checks how many lines of code are covered in all files. We see a lot of red because in this app, we didn't do extensive testing and covered all the files.

File	% Stats	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	37.79	14.75	26.56	39.06	
src	25	100	100	25	
App.js	100	100	100	100	
index.js	0	100	100	0	8-11
src/app	100	100	100	100	
store.js	100	100	100	100	
src/components	45.16	26.66	36.66	48.1	
Dashboard.js	88.88	50	75	85.71	19
Header.js	70	50	66.66	66.66	12-14
Login.js	66.66	62.5	50	76.47	24,31-33
Register.js	4.34	0	0	5.26	10-41
Spinner.js	100	100	100	100	
TaskForm.js	54.54	100	33.33	54.54	12-15,23
TaskItem.js	75	100	50	75	11
TaskList.js	7.69	0	0	10	9-24
src/features/auth	32.69	6.25	13.33	34	
authService.js	35.71	0	0	38.46	5-9,13-17
authSlice.js	31.57	8.33	16.66	32.43	17-23,29-36,56-84
src/features/tasks	30.5	0	16.66	31.03	
taskService.js	57.14	100	33.33	57.14	6-12,26-32
taskSlice.js	22.22	0	13.33	22.72	15-23,36-39,47-55,64,69-105


```

Test Suites: 6 passed, 6 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        7.169 s
Ran all test suites.

```

Figure 11.26: Checking coverage of files

We can also see the coverage in a nice html file. A coverage folder will be created in our frontend app. So, open it and then the lcov-report folder inside it. Here, we will find an index.html file, which contains all of our reports of coverage.

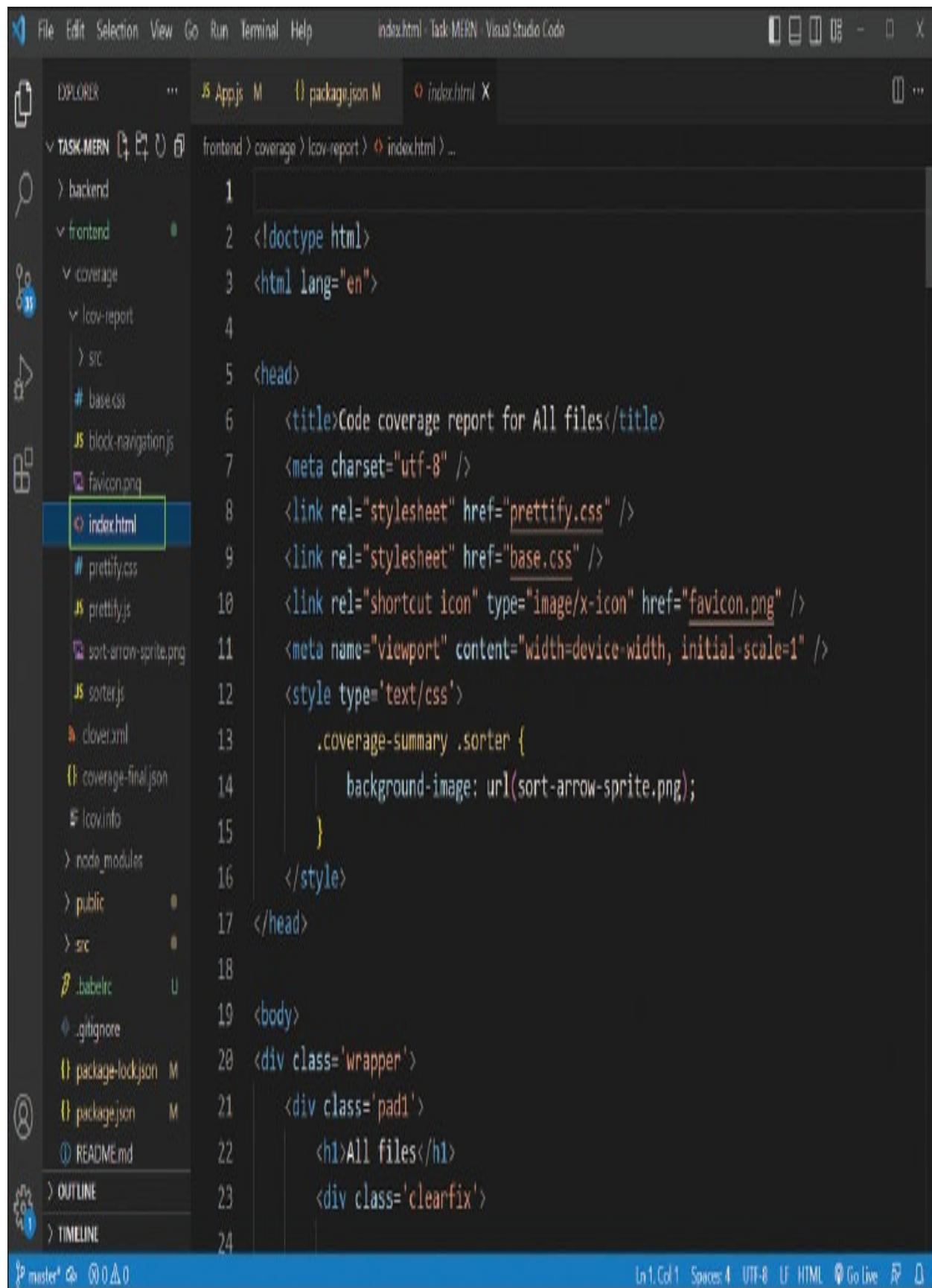


Figure 11.27: HTML file for coverage

Now, on opening the index.html file in the browser we will see the coverage of the major files in a nice graphical format.



Figure 11.28: Coverage in a browser

Conclusion

In this chapter, we have learned to test our frontend code written in React. We have used the in-built libraries of Jest and React testing library to unit test our various components.

In the next chapter, we will test the backend built in NodeJS using Jest and Supertest.

Points to remember

To write tests we need to target an element. Sometimes we don't have a unique field, and we can use data-testid for the same.

We don't want to call the real API endpoint in testing. So, we add mocking capabilities in testing.

We can also mock data, services, and functions, and through it, write complex tests to cover all scenarios.

Once we are done with testing, we need to check the coverage for all test cases.

CHAPTER 12

Backend Testing

Introduction

This chapter covers the testing of our backend using Jest. This library is not in built into NodeJS, but we have to install it manually, as with most things related to NodeJS.

Structure

In this chapter, we will cover the following topics:

Setting up testing in the backend

Testing Register User controller

Testing getTasks and setTask controllers

Testing Update Task controller

Setting up testing in the backend

We need to do all the setup for testing, as the NodeJS app doesn't come with any testing installed. So, first change to the backend directory and then open it with VS Code with the code. command.

The screenshot shows a terminal window with the following session:

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN
$ cd backend/
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
$ code .
```

The terminal window has a dark background and light-colored text. The prompt indicates the user is in a directory named 'backend'. The command 'code .' is highlighted with a red rectangle, indicating it is the current active command.

Figure 12.1: Switching to the Backend

Head over to <https://jestjs.io/docs/getting-started> site and copy the command to install Jest in your code.

The screenshot shows a web browser window with the URL jestjs.io/docs/getting-started. The page is titled "Getting Started" and is part of the Jest documentation. The left sidebar lists various topics: Introduction, Getting Started (which is selected), Using Matchers, Testing Asynchronous Code, Setup and Teardown, Mock Functions, Jest Platform, Jest Community, More Resources, Guides, Framework Guides, and Upgrade Guides. The main content area starts with a heading "Getting Started" and instructions to install Jest using npm, Yarn, or pnpm. It includes a code block for a sum function and a note about creating a sum.test.js file. At the bottom, there's a code block for a test file.

JEST 29.5

Introduction > Getting Started

Version 29.5

Getting Started

Install Jest using your favorite package manager:

npm Yarn pnpm

```
npm install --save-dev jest
```

Let's get started by writing a test for a hypothetical function that adds two numbers. First, create a `sum.js` file:

```
function sum(a, b) {
  return a + b;
}

module.exports = sum;
```

Then, create a file named `sum.test.js`. This will contain our actual test:

```
const sum = require('./sum');
```

Figure 12.2: Jest Docs

In the backend folder from the terminal, we will give the following command. Notice that we are installing jest as dev dependent since we don't require it in production.

```
npm install --save-dev jest
```

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
```

```
$ npm install --save-dev jest
```

```
added 395 packages, and audited 396 packages in 33s
```

```
42 packages are looking for funding  
run 'npm fund' for details
```

```
4 vulnerabilities (3 moderate, 1 critical)
```

```
To address issues that do not require attention, run:
```

```
npm audit fix
```

```
To address all issues (including breaking changes), run:
```

```
npm audit fix --force
```

```
Run 'npm audit' for details.
```

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
```

```
$ |
```

Figure 12.3: Installing Jest

Copy the jest.config.js, from the code at the end of this chapter. Here, make sure that the testMatch file is not commented on.

File Edit Selection View Go Run Terminal Help jest.config.js - Visual Studio Code

EXPLORER ... **jest.config.js** X

BACKEND
connect
controllers
middleware
models
node_modules
routes
env
gitignore
jest.config.js
package-lock.json
package.json
server.js

```
// testEnvironmentOptions: {},  
// Adds a location field to test results  
// testLocationInResults: false,  
// The glob patterns Jest uses to detect test files  
testMatch: ['**/_tests_/**/*.[jt]s?(x)', '**/?(*.)+(spec|test).[tj]s?(x)'],  
// An array of regexp pattern strings that are matched against all test paths,  
// matched tests are skipped  
testPathIgnorePatterns: [  
    // "\\\node_modules\\\""  
],  
// The regexp pattern or array of patterns that Jest uses to detect test files  
testRegex: [],  
// This option allows the use of a custom results processor  
testResultsProcessor: undefined,  
// This option allows use of a custom test runner  
testRunner: "jest-circus/runner",  
// This option sets the URL for the isdom environment. It is reflected in
```

Line 1, Col 1 Spaces: 4 UTF-8 LF JavaScript Go Live ⚡ ⚡ ⚡

Figure 12.4: Configuring Jest

Finally, we will update the package.json file to have a script called test, which will call jest.

File Edit Selection View Go Run Terminal Help package.json - Visual Studio Code

server.js package.json

package.json > ..

```
1  {
2    "name": "backend",
3    "version": "1.0.0",
4    "description": "",
5    "main": "server.js",
6    "scripts": {
7      "start": "node server.js",
8      "dev": "nodemon server.js",
9      "test": "jest"
10    },
11    "keywords": [],
12    "author": "",
13    "license": "ISC",
14    "dependencies": {
15      "bcryptjs": "^2.4.3",
16      "dotenv": "^16.0.3",
17      "express": "^4.18.2",
18      "express-async-handler": "^1.2.0",
19      "jsonwebtoken": "^9.0.0",
20      "mongoose": "^7.2.0"
21    },
22    "devDependencies": {
23      "jest": "^29.6.2",
24    }
25  }
```

In 5 Col 23 Spaces 2 LF-B LF JSON Go live

Figure 12.5: Adding Jest

Testing registerUser controller

We will test the registerUser controller now. It is called when we hit the / route with a post call. In the userRoutes.js file we call the registerUser function through a router.post().

The screenshot shows a dark-themed instance of Visual Studio Code. The title bar reads "userRoutes.js - backend - Visual Studio Code". The left sidebar displays a file tree for a "BACKEND" project, listing files like server.js, userRoutes.js, userController.js, taskController.js, taskRoutes.js, env, .gitignore, jest.config.js, package-lock.json, package.json, and server.js. The "userRoutes.js" file is currently selected and open in the main editor area. The code in the editor is:

```
1 const express = require('express')
2 const router = express.Router()
3 const { registerUser, loginUser, getCurrentUser } = require('../controllers/userController')
4 const { protect } = require('../middleware/authMiddleware')
5
6 router.post('/', registerUser)
7 router.post('/login', loginUser)
8 router.get('/current', protect, getCurrentUser)
9
10 module.exports = router
11
```

The bottom status bar shows "Ln 1, Col 1 Spaces:4 UTF-8 CR LF JavaScript Go Live".

Figure 12.6: Calling Controller

The registerUser function in the userController.js file is as follows, which we created in earlier chapters. Here, we get the name, email, and password from the body. After validating all fields, we also check if the email exists with findOne(). If it exists, we are throwing an error.

We also encrypt the password with bcrypt before saving it in the database with bcrypt. Finally, we send a status of 201 and _id, name, and email back.

```
const registerUser = asyncHandler(async (req, res) => {  
  const { name, email, password } = req.body  
  if (!name || !email || !password) {  
    res.status(400)  
    throw new Error('All fields are mandatory')  
  }  
  const userExists = await User.findOne({ email })  
  if (userExists) {  
    res.status(400)  
    throw new Error('User Exists')  
  }  
}
```

```
const salt = await bcrypt.genSalt(10)

const hashedPassword = await bcrypt.hash(password, salt)

const user = await User.create({ name, email, password: hashedPassword })

if (user) {

  res.status(201).json({ _id: user.id, name: user.name, email: user.email, token: generateJWTtoken(user._id) })

} else {

  res.status(400)

  throw new Error('Invalid user data')

}

})
```

File Edit Selection View Go Run Terminal Help userController.js - backend - Visual Studio Code

```
const registerUser = asyncHandler(async (req, res) => {
    const { name, email, password } = req.body
    if (!name || !email || !password) {
        res.status(400)
        throw new Error('All fields are mandatory')
    }

    const userExists = await User.findOne({ email })
    if (userExists) {
        res.status(400)
        throw new Error('User Exists')
    }

    const salt = await bcrypt.genSalt(10)
    const hashedPassword = await bcrypt.hash(password, salt)
    const user = await User.create({ name, email, password: hashedPassword })
    if (user) {
        res.status(201).json({ _id: user.id, name: user.name, email: user.email, token: generateJWTtoken(user._id) })
    } else {
        res.status(400)
        throw new Error('Invalid user data')
    }
})
```

In 22 Col 1 Spaces: 4 UTF-8 CRLF {} JavaScript ⚡ Go Live ⌂ ⌂ ⌂

Figure 12.7: Register User Controller

We will write the test for this file. So, create a `__tests__` folder in the root directory and a `userController.test.js` file inside it. Here, we first import the `registerUser` from the `userController.js` file.

In `jest.mock()`, we pass the path to `userModel` as the first parameter and the arrow function as the second parameter. Inside the arrow function, we have created a variable of `mockUser` which is an object containing `_id`, `name`, and `email`.

Then, we return the mocked function of `findOne` and `create` from it. Next, we again use the `jest.mock()` and mock the `sign` function from the `userController.js` file.

Finally, we use `bcrypt` to mock the `genSalt` and `hash` functions used in the `userController.js` file.

```
const { registerUser } = require('../controllers/userController');
```

```
// Mock User model functions
```

```
jest.mock('../models/userModel', () => {
```

```
// Mock User model
```

```
const mockUser = {
```

```
_id: 'user-id',  
name: 'John Doe',  
email: 'johndoe@example.com',  
};  
  
return {  
findOne: jest.fn().mockResolvedValue(null),  
create: jest.fn().mockResolvedValue(mockUser),  
};  
});  
  
// Mock JWT and bcrypt  
jest.mock('jsonwebtoken', () => ({  
sign: jest.fn().mockReturnValue('mock-token'),  
}));  
  
const bcrypt = require('bcryptjs');  
  
bcrypt.genSalt = jest.fn().mockResolvedValue('mock-salt');  
  
bcrypt.hash = jest.fn().mockResolvedValue('mock-hashed-password');
```

A screenshot of the Visual Studio Code interface showing a Jest test file for a user controller. The file is named `userControllerTest.js` and is located in a directory structure under `src/test/_tests_`. The code uses Jest's `jest.mock` function to mock various modules like `userModel`, `jsonwebtoken`, and `bcryptjs` to test the `registerUser` function.

```
1 const { registerUser } = require('../controllers/userController');
2
3 // Mock User model functions
4 jest.mock('../models/userModel', () => {
5   // Mock User model
6   const mockUser = {
7     _id: 'user-id',
8     name: 'John Doe',
9     email: 'johndoe@example.com',
10    };
11
12  return {
13    findOne: jest.fn().mockResolvedValue(null),
14    create: jest.fn().mockResolvedValue(mockUser),
15  };
16 });
17 // Mock JWT and bcrypt
18 jest.mock('jsonwebtoken', () => ({
19   sign: jest.fn().mockReturnValue('mock-token'),
20 }));
21 const bcrypt = require('bcryptjs');
22 bcrypt.genSalt = jest.fn().mockResolvedValue('mock-salt');
23 bcrypt.hash = jest.fn().mockResolvedValue('mock-hashed-password');
```

Figure 12.8: Register User Test setup

After the initial configuration, we will write the first test. Here, inside the test function from jest, we have two parameters. The first parameter is a string, which will be displayed when we run the test.

The second parameter is an arrow function, which will contain our whole test logic. Here, we have created a variable of req first, which is an object. This object contains a key of the body, which contains a value as an object. This object contains the name, email and password data.

In the next res object, we contain two values of status and json which are mock of status and json function from register function.

We call the registerUser function with the req and res using await. Then we expect the status to be 201 in this success scenario.

```
// Test

test('should register a new user', async () => {
  const req = {
    body: {
      name: 'John Doe',
      email: 'johndoe@example.com',
    }
  }

  const res = {
    status: jest.fn(() => 201),
    json: jest.fn()
  }

  await registerUser(req, res)

  expect(res.status).toHaveBeenCalledWith(201)
  expect(res.json).toHaveBeenCalledWith({
    message: 'User registered successfully'
  })
})
```

```
password: 'password',  
},  
};  
  
const res = {  
status: jest.fn().mockReturnThis(),  
json: jest.fn(),  
};  
  
await registerUser(req, res);  
expect(res.status).toHaveBeenCalledWith(201);  
});
```

File Edit Selection View Go Run Terminal Help userControllerTest.js - backend - Visual Studio Code

server.js userRoutes.js UserController.js userControllerTest.js

```
test('should register a new user', async () => {
    const req = {
        body: {
            name: 'John Doe',
            email: 'johndoe@example.com',
            password: 'password',
        },
    };
    const res = {
        status: jest.fn().mockReturnThis(),
        json: jest.fn(),
    };

    await registerUser(req, res);
    expect(res.status).toHaveBeenCalledWith(201);
});
```

Figure 12.9: Register User First Test

Now, in the terminal from the backend folder, run the command `npm run test` and we will see the test has been passed. Also, notice that the name of the test which we have given “should register a new user” is being displayed.

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
```

```
$ npm run test
```

```
> backend@1.0.0 test
```

```
> jest
```

```
PASS ...tests.../userController.test.js
```

```
  ✓ should register a new user (4 ms)
```

```
Test Suites: 1 passed, 1 total
```

```
Tests:       1 passed, 1 total
```

```
Snapshots:  0 total
```

```
Time:       0.433 s, estimated 2 s
```

```
Ran all test suites.
```

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
```

```
$
```

Figure 12.10: Register User Test run

We write the test case for an error in a missing field. Again, we have used the test method from jest but given it a different name. The req and the res are almost similar to our earlier test, but in the req, we are missing the email field.

After that, we call the registerUser through an expect, but it will throw an error because of the missing field. So, we chain it with rejects and toThrow from jest where we pass the “All fields are mandatory” message. Note that this message is the same as the message in the register function inside the userController.js file.

Lastly, we also check whether the status of 400 will be received because of the failed scenario.

```
test('should return a 400 error if any field is missing', async () => {  
  const req = {  
    body: {  
      name: 'John Doe',  
      email: '', // Missing email field intentionally  
      password: 'password',  
    },  
  };  
  
  const res = {
```

```
status: jest.fn().mockReturnThis(),  
json: jest.fn(),  
};  
  
await expect(registerUser(req, res)).rejects.toThrow('All fields are mandatory');  
expect(res.status).toHaveBeenCalledWith(400);  
});
```

A screenshot of the Visual Studio Code interface showing a Jest test file named `userController.test.js`. The code is written in JavaScript and uses Jest's `expect` library to test a function named `registerUser`.

```
File Edit Selection View Go Run Terminal Help userController.test.js - backend - Visual Studio Code
server.js userRoutes.js userController.js userController.test.js X
... userController.test.js ...
44 test('should return a 400 error if any field is missing', async () => {
45   const req = {
46     body: {
47       name: 'John Doe',
48       email: '', // Missing email field intentionally
49       password: 'password',
50     },
51   };
52
53   const res = {
54     status: jest.fn().mockReturnThis(),
55     json: jest.fn(),
56   };
57
58   await expect(registerUser(req, res)).rejects.toThrow('All fields are mandatory');
59   expect(res.status).toHaveBeenCalledWith(400);
60 });
61
62
```

Figure 12.11: Register User Second Test

Again, from the terminal run the command npm run test and we will see the new test has been passed.

```
naben@Nabendu: MINGW64 ~/Desktop/Task-MERN/backend
```

```
$ npm run test
```

```
> backend@1.0.0 test
```

```
> jest
```

```
 PASS  __tests__/userController.test.js
```

```
  | should register a new user (3 ms)
```

```
  | should return a 400 error if any field is missing (10 ms)
```

```
Test Suites: 1 passed, 1 total
```

```
Tests:       2 passed, 2 total
```

```
Snapshots:  0 total
```

```
Time:        0.675 s, estimated 1 s
```

```
Ran all test suites.
```

```
naben@Nabendu: MINGW64 ~/Desktop/Task-MERN/backend
```

```
$
```

Figure 12.12: Register User Test run

Testing getTasks and setTask controller

We will now test the getTasks and setTask functions from the taskController.js file, which we created in earlier chapters.

These are simple functions which are used to get the task and add a new task in the database. In the getTasks we find the task with the find() method in which we pass the user id.

In the setTask, we first check if the text is present and throw an error if it is not present. After that, we use the create() method to send the text and user id to the database and save it. Finally, we send back a status of 200 along with the task to the browser.

```
const getTasks = asyncHandler(async (req, res) => {  
  const tasks = await Task.find({ user: req.user.id })  
  res.status(200).json(tasks);  
})
```

```
const setTask = asyncHandler(async (req, res) => {  
  if(!req.body.text){  
    res.status(400)  
    throw new Error('Please enter a task')  
  }
```

}

```
const task = await Task.create({ text: req.body.text, user: req.user.id })  
res.status(200).json(task);  
})
```

File Edit Selection View Go Run Terminal Help taskController.js - backend - Visual Studio Code

EXPLORER ... server.js JS taskRoutes.js JS taskController.js X JS taskController.test.js JS UserController.js JS UserController.test.js ...

BACKEND > _tests_ controllers > JS taskController.js > (unknow)

> connect
> controllers
JS taskController.js
> middleware
> models
JS taskModel.js
JS userModel.js
> node_modules
> routes
JS .env
JS ignore
JS jest.config.js
{} package-lock.json
{} package.json
JS server.js

```
1 const asyncHandler = require('express-async-handler')
2 const Task = require('../models/taskModel')
3 const User = require('../models/userModel')

4
5 const getTasks = asyncHandler(async (req, res) => {
6   const tasks = await Task.find({ user: req.user.id })
7   res.status(200).json(tasks);
8 }

9
10 const setTask = asyncHandler(async (req, res) => {
11   if(!req.body.text){
12     res.status(400)
13     throw new Error('Please enter a task')
14   }

15   const task = await Task.create({ text: req.body.text, user: req.user.id })
16   res.status(200).json(task);
17

18 })

19
20 const updateTask = asyncHandler(async (req, res) => {
21   const task = await Task.findById(req.params.id)
22
23   if (!task) {
24     res.status(400)

Ln 69, Col 1 (62 selected) Spaces: 4 UTF-8 CRLF () JavaScript Go Live ⌂ ⌂ ⌂
```

Figure 12.13: Get Task and Set Task Controllers

Now, we will create a file taskController.test.js in the `__tests__` folder. Here, we first import the `getTasks` and `setTask` from the `taskController.js` file. After that, we import `task` from the `models` and also mock the `task` model with `jest.mock()`.

We write our first test which is to get a task. Here, our `req` variable is an object with a `user` key which contains a value of the object. This object has an `id` as a key and a `user-id` as a value.

Then, we mock the `tasks`, which is an array of objects. Now, each object contains a `_id`, `text` and `user`. Next, we mock the `find()` method in the `getTasks()` method inside the `taskController.js` file.

Our `res` object contains the `status` and `json` as key and the respective `jest.fn()` for mocking them. Next, we call the `getTasks()` with `req` and `res`.

Finally, we write the `expect` statements where we check the `res.status` to be `200` and also the `res.json` to contain the `tasks`.

```
const { getTasks, setTask, updateTask } = require('../controllers/taskController');
const Task = require('../models/taskModel');
jest.mock('../models/taskModel');
```

```
test('should get tasks for a user', async () => {
  const req = { user: { id: <user-id> } };
  // Mocking tasks for the user
  const tasks = [
    { _id: 'task-id-1', text: 'Task 1', user: 'user-id' },
    { _id: 'task-id-2', text: 'Task 2', user: 'user-id' },
  ];
  // Mocking the find method to return tasks for the user
  Task.find.mockResolvedValue(tasks);

  const res = {
    status: jest.fn().mockReturnThis(),
    json: jest.fn(),
  };

  await getTasks(req, res);
  // Ensure that the response contains the expected tasks
  expect(res.status).toHaveBeenCalledWith(200);
  expect(res.json).toHaveBeenCalledWith(tasks);
});
```

The screenshot shows a Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Active File:** taskControllerTest.js - backend - Visual Studio Code
- Left Sidebar:** Explorer, Backstage, Tests, controllers, models, middleware, routes, env, gitignore, jestconfig, package-lock.json, package.json, server.js, Outline, Timeline.
- Tests View:** Shows a list of tests: taskController.test.js (selected), taskRoutes.js, taskController.js, taskControllerTest.js (X), userController.js, userControllerTest.js.
- Code Editor:** The code for taskControllerTest.js is displayed, containing a Jest test for the getTasks function of the taskController.

```
const { getTasks, setTask } = require('../controllers/taskController');
const Task = require('../models/taskModel');
jest.mock('../models/taskModel');

test('should get tasks for a user', async () => {
  const req = { user: { id: 'user-id' } };
  // Mocking tasks for the user
  const tasks = [
    { _id: 'task-id-1', text: 'Task 1', user: 'user-id' },
    { _id: 'task-id-2', text: 'Task 2', user: 'user-id' },
  ];
  // Mocking the find method to return tasks for the user
  Task.find.mockResolvedValue(tasks);

  const res = {
    status: jest.fn().mockReturnThis(),
    json: jest.fn(),
  };
  await getTasks(req, res);
  // Ensure that the response contains the expected tasks
  expect(res.status).toHaveBeenCalledWith(200);
  expect(res.json).toHaveBeenCalledWith(tasks);
});
```

Figure 12.14: Get Task Test

The next test is for setting a task and it is almost similar to the previous test for getting a task. Here again, first we have a req object.

Then, we mock the task object. Note that it is different from our earlier test in which we have a task array. Next, we mock the create() method in the setTask() method inside the taskController.js file.

Our res object contains the status and json as key and is similar to the earlier test. Next, we call the setTask with the req and res.

We have written the expect statements where we check the res.status to be 200. Also, the res.json to contain the task.

```
test('should set a new task for a user', async () => {  
  const req = { user: { id: <user-id> }, body: { text: <New Task> } };  
  
  // Mocking the created task  
  
  const task = { _id: <new-task-id>, text: <New Task>, user: <user-id> };  
  
  // Mocking the create method to return the new task
```

```
Task.create.mockResolvedValue(task);

const res = {
  status: jest.fn().mockReturnThis(),
  json: jest.fn(),
};

await setTask(req, res);

// Ensure that the response contains the new task
expect(res.status).toHaveBeenCalledWith(200);
expect(res.json).toHaveBeenCalledWith(task);
});
```

File Edit Selection View Go Run Terminal Help taskController.test.js - backend - Visual Studio Code

server.js taskRoutes.js taskController.js taskController.test.js X

tests > taskController.test.js > test('should get tasks for a user') callback

```
22 | expect(res.json).toHaveBeenCalledWith(tasks);
23 | });
24 |
25 test('should set a new task for a user', async () => {
26   const req = { user: { id: 'user-id' }, body: { text: 'New Task' } };
27
28   // Mocking the created task
29   const task = { _id: 'new-task-id', text: 'New Task', user: 'user-id' };
30
31   // Mocking the create method to return the new task
32   Task.create.mockResolvedValue(task);
33
34   const res = {
35     status: jest.fn().mockReturnThis(),
36     json: jest.fn(),
37   };
38
39   await setTask(req, res);
40
41   // Ensure that the response contains the new task
42   expect(res.status).toHaveBeenCalledWith(200);
43   expect(res.json).toHaveBeenCalledWith(task);
44 });
45
```

Ln 7 Col 1 Sources 4 UTF-8 CRLF ↻ JavaScript ⚡ Go Live ⌂ ⌂ ⌂

Figure 12.15: Set Task Test

We write the test for the error, when we have a missing task text. Here again, we have a req object but we have been given an empty body.

Next, the res object has status and json keys which have mock values with jest.fn(). After that, we call the setTask through an expect, but it will throw an error because of the missing field. So, we chain it with rejects and toThrow from jest where we pass the “Please enter a task” message.

Note that this message is the same as the message in the setTask function inside the taskController.js file. Lastly, we also check whether the status of 400 will be received because of the failed scenario.

```
test('should return a 400 error for missing task text', async () => {  
  const req = { user: { id: 'user-id' }, body: {} }; // Missing the "text" field  
  
  const res = {  
    status: jest.fn().mockReturnThis(),  
    json: jest.fn(),  
  };  
  
  await expect(setTask(req, res)).rejects.toThrow('Please enter a task');
```

```
expect(res.status).toHaveBeenCalledWith( $\epsilon \dots$ );  
});
```

A screenshot of Visual Studio Code showing a Jest test file named `taskController.test.js`. The code is written in JavaScript and uses Jest's `test` and `expect` functions to test a function called `setTask`.

```
File Edit Selection View Go Run terminal Help taskController.test.js - Visual Studio Code

server.js tests tsconfig.json taskController.js taskController.test.js ...

_line_ > taskController.test.js > 0 test('should get tasks for a user') callback
P 45
46 test('should return a 400 error for missing task text', async () => {
47   const req = { user: { id: 'user-id' }, body: {} }; // Missing the "text" field
48
49   const res = {
50     status: jest.fn().mockReturnThis(),
51     json: jest.fn(),
52   };
53
54   await expect(setTask(req, res)).rejects.toThrow('Please enter a task');
55   expect(res.status).toHaveBeenCalledWith(400);
56 });
57
```

Figure 12.16: Set Task Error Test

We will run the npm run test from the terminal and our new tests in the taskController.test.js file passes.

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
```

```
$ npm run test
```

```
> backend@1.0.0 test
```

```
> jest
```

```
PASS __tests__/_userController.test.js
```

```
PASS __tests__/_taskController.test.js
```

```
Test Suites: 2 passed, 2 total
```

```
Tests:       5 passed, 5 total
```

```
Snapshots:  0 total
```

```
Time:        1.685 s, estimated 5 s
```

```
Ran all test suites.
```

```
naben@Nabendu MINGW64 ~/Desktop/Task-MERN/backend
```

```
$
```

Figure 12.17: Running Tests

Testing updateTask controller

We will now test the update task controller from the taskController.js file. Here, we first find the task with the findById method by passing the id. Next, we check if the task is present. If it is not present, we throw an error of “Task not found”.

We also find the user with findById and pass the user id. Again, if the user is not found we throw the error “No such user found”.

Next, the test is for checking if the task’s user id and user’s id are the same. If they are not the same, we throw the error “User is not authorized to update”.

Finally, we update the task with the method of findByIdAndUpdate by passing the task id and thereq.body. We also send back a status of 200, along with the updated task to the sender.

```
const updateTask = asyncHandler(async (req, res) => {  
  const task = await Task.findById(req.params.id)  
  
  if (!task) {  
    res.status(400)  
    throw new Error('Task not found')  
  }  
  // ...  
  res.status(200).json(task)  
})
```

```
}
```

```
const user = await User.findById(req.user.id)
```

```
if(!user){
```

```
    res.status(401)
```

```
    throw new Error('No such user found')
```

```
}
```

```
if (task.user.toString() !== user.id) {
```

```
    res.status(401)
```

```
    throw new Error('User is not authorized to update')
```

```
}
```

```
const updatedTask = await Task.findByIdAndUpdate(req.params.id, req.body, {  
  new: true })
```

```
res.status(200).json(updatedTask)
```

```
)
```

A screenshot of a code editor window showing a file named `taskController.js`. The code is a Node.js module for updating tasks. It uses `asyncHandler` to handle asynchronous operations, `Task` and `User` models from databases, and `jsonwebtoken` for user authentication.

```
File Edit Selection View Go Run Terminal Help taskController.js - Task Status Cycle
server.js taskModel.js taskController.js taskController.test.js

controller.js taskController.js updateTask asyncHandler(req, res) {
  const task = await Task.findById(req.params.id)

  if (!task) {
    res.status(404)
    throw new Error('Task not found')
  }

  const user = await User.findById(req.user.id)

  if (!user) {
    res.status(401)
    throw new Error('No such user found')
  }

  if (task.user.toString() !== user.id) {
    res.status(401)
    throw new Error('User is not authorized to update')
  }

  const updatedTask = await Task.findByIdAndUpdate(req.params.id, req.body, { new: true })
  res.status(200).json(updatedTask)
}
```

Figure 12.18: Update Task Controller

Back in the taskController.test.js file, we have to update our imports first. We first import the updateTask from the controllers. Next, we also import the user model and also mock the user model with jest.mock().

```
const { getTasks, setTask, updateTask } = require('../controllers/taskController');

const Task = require('../models/taskModel');

const User = require('../models/userModel');

jest.mock('../models/taskModel');

jest.mock('../models/userModel');
```

```
File Edit Selection View Go Run Terminal Help taskController.js taskController.js X

taskController.js > taskControllerTest.js

1 const { getTasks, setTask, updateTask } = require('../controllers/taskController');
2 const Task = require('../models/taskModel');
3 const User = require('../models/userModel');
4 jest.mock('../models/taskModel');
5 jest.mock('../models/userModel');

6

7 test('should get tasks for a user', async () => {
8   const req = { user: { id: 'user-id' } };
9   // Mocking tasks for the user
10  const tasks = [
11    { _id: 'task-id-1', text: 'Task 1', user: 'user-id' },
12    { _id: 'task-id-2', text: 'Task 2', user: 'user-id' }
13  ];
14  // Mocking the find method to return tasks for the user
15  Task.find.mockResolvedValue(tasks);
16
17  const res = {
18    status: jest.fn().mockReturnThis(),
19    json: jest.fn(),
20  };
21  await getTasks(req, res);
22  // Ensure that the response contains the expected tasks
23  expect(res.status).toHaveBeenCalledWith(200);
24  expect(res.json).toHaveBeenCalledWith(tasks);
```

Figure 12.19: Importing user models

Now, we will write our first test for the update task in the taskController.test.js file. Here, we check if we are getting the 401 error if the user is not found. First, we have created a variable of taskId and userId.

We created a req object with params, user and body as keys. Each one of them contains an object as the value. The params object has a key-value pair of id and taskId. Then the user object has a key-value pair of id and userId and the body object has a key-value pair of text and ‘Updated Task’.

We create a variable of taskToUpdate which is an object. It contains the keys of _id, text and user with respective values.

We mock the task findById to return the mocked task of taskToUpdate. Similarly, we also mock the user findById and return a null which tells the user is not found.

Our res object contains status and json with respective mock functions. After that, we call the updateTask through an expect, but it will throw an error because of the missing user. So, we chain it with rejects and toThrow from jest where we are passing the “No such user found” message.

Note that this message is the same as the message in the updateTask function inside the taskController.js file. Lastly, we also check whether the status of 401 will be received because of the failed scenario.

```
test('should return a 401 error if user is not found', async () => {
  const taskId = 'task-id-1';
  const userId = 'non-existent-user-id';
  const req = { params: { id: taskId }, user: { id: userId }, body: { text: 'Updated Task' } };
  // Mocking the task to be updated
  const taskToUpdate = { _id: taskId, text: 'Original Task', user: 'user-id' };
  // Mocking the findById method to return the task to be updated
  Task.findById.mockResolvedValue(taskToUpdate);
  // Mocking the findById method to return null, indicating user not found
  User.findById.mockResolvedValue(null);
  const res = {
    status: jest.fn().mockReturnThis(),
    json: jest.fn(),
  };
  await expect(updateTask(req, res)).rejects.toThrow('No such user found');
  expect(res.status).toHaveBeenCalledWith(401);
});
```

File Edit Selection View Go Run Terminal Help taskController.test.js - Backend · Visual Studio Code

server.js testRoutes.js taskController.js taskController.test.js X

```
_tests_ > taskController.test.js ..
```

59
60 test('should return a 401 error if user is not found', async () => {
61 const taskId = 'task-id-1';
62 const userId = 'non-existent-user-id';
63 const req = { params: { id: taskId }, user: { id: userId }, body: { text: 'Updated Task' } };
64
65 // Mocking the task to be updated
66 const taskToUpdate = { _id: taskId, text: 'Original Task', user: 'user-id' };
67
68 // Mocking the findById method to return the task to be updated
69 Task.findById.mockResolvedValue(taskToUpdate);
70
71 // Mocking the findById method to return null, indicating user not found
72 User.findById.mockResolvedValue(null);
73
74 const res = [
75 status: jest.fn().mockReturnThis(),
76 json: jest.fn(),
77];
78
79 await expect(updateTask(req, res)).rejects.toThrow('No such user found');
80 expect(res.status).toHaveBeenCalledWith(401);
81 };

B2

Ln 58 Col 4 Spaces 4: CRLF: () JavaScript: Go Live: ⌂ ⌂ ⌂

Figure 12.20: User not found error test

The next test we have for the user is not authorized to update the task. Here again, we have a taskId and userId variable. After that, we have a req object exactly similar to our previous test.

Our taskToUpdate also has a similar structure as the previous test. But notice here that the user value is different from the owner's. After that, we are mocking the task findById which will return the task to be updated.

We mock the user findById, which will return our authorized user with the id of user-id-2. Again, our res object contains status and json with respective mock functions.

We call the updateTask through an expect, but it will throw an error because of the unauthorized user. So, we chain it with rejects and toThrow from jest where we pass the “User is not authorized to update” message.

Lastly, we also check whether the status of 401 will be received because of the failed scenario.

```
test(<should return a 401 error if user is not authorized to update the task>, async  
() => {
```

```
    const taskId = 'task-id-1';
```

```
    const userId = 'user-id-2'; // Different user ID from the task owner
```

```
const req = { params: { id: taskId }, user: { id: userId }, body: { text: 'Updated Task' } };

// Mocking the task to be updated, owned by a different user

const taskToUpdate = { _id: taskId, text: 'Original Task', user: 'user-id-1' };

// Mocking the findById method to return the task to be updated

Task.findById.mockResolvedValue(taskToUpdate);

// Mocking the findById method to return the user

User.findById.mockResolvedValue({ _id: userId });

const res = {

  status: jest.fn().mockReturnThis(),

  json: jest.fn(),

};

await expect(updateTask(req, res)).rejects.toThrow('User is not authorized to update');

expect(res.status).toHaveBeenCalledWith(401);

});
```

A screenshot of the Visual Studio Code interface showing a Jest test file named `taskController.test.js`. The code is written in JavaScript and uses Jest's `jest.fn()` to mock database methods. It tests whether an unauthorized user receives a 401 error when attempting to update a task.

```
File Edit Selection View Go Run Terminal Help taskController.test.js - Visual Studio Code

server.js taskRoutes.js taskController.js taskController.test.js

_tsts_> taskController.test.js

83 test('should return a 401 error if user is not authorized to update the task', async () => {
84   const taskId = 'task-id-1';
85   const userId = 'user-id-2'; // Different user ID from the task owner
86   const req = { params: { id: taskId }, user: { id: userId }, body: { text: 'Updated Task' } };
87
88   // Mocking the task to be updated, owned by a different user
89   const taskToUpdate = { _id: taskId, text: 'Original Task', user: 'user-id-1' };
90
91   // Mocking the findById method to return the task to be updated
92   Task.findById.mockResolvedValue(taskToUpdate);
93
94   // Mocking the findById method to return the user
95   User.findById.mockResolvedValue({ _id: userId });
96
97   const res = {
98     status: jest.fn().mockReturnThis(),
99     json: jest.fn(),
100   };
101
102   await expect(updateTask(req, res)).rejects.toThrow('User is not authorized to update');
103   expect(res.status).toHaveBeenCalledWith(401);
104 });
105
```

Figure 12.21: User not authorized error test

We will again run the npm run test from the terminal and all the new tests will also pass.

```
naben@Nabendu MING64 ~/Desktop/Task-MERN/backend
```

```
$ npm run test
```

```
> backend@1.0.0 test
```

```
> jest
```

```
PASS __tests__/userController.test.js
```

```
PASS __tests__/taskController.test.js
```

```
Test Suites: 2 passed, 2 total
```

```
Tests: 7 passed, 7 total
```

```
Snapshots: 0 total
```

```
Time: 1.785 s, estimated 2 s
```

```
Ran all test suites.
```

```
naben@Nabendu MING64 ~/Desktop/Task-MERN/backend
```

```
$
```

Figure 12.22: Running Tests

Conclusion

In this chapter, we have learned to test our backend code written in NodeJS. We have used the popular libraries of jest to unit test our various components.

In the final chapter, we will deploy both our frontend and backend code separately.

Points to remember

Configuring jest from the start for a NodeJS application

Adding mocking capabilities for testing

Writing complex tests using mock data and services

Writing tests for common backend scenarios of get task, set task and update task

CHAPTER 13

Deployment

Introduction

This chapter covers the deployment of both our frontend and backend code. We will use it with freely available services. This includes deploying the front end with Netlify and the back end with Back4app.

Structure

In this chapter, we will discuss the following topics:

Frontend code repo creation

Backend code repo creation

Backend deployment in Back4app

Backend integration with frontend

Frontend deployment in Netlify

Fixing CORS errors and final deployments

Frontend code repo creation

We will first create a repository for our frontend code in GitHub. For this, you need to have a GitHub account and also Git installed on your Windows machine. By default, Mac and Linux machines have Git installed.

First login to your GitHub account at <https://github.com/>.

The screenshot shows the GitHub desktop application interface. At the top, there's a header bar with the GitHub logo, a search bar containing 'github.com', and various icons for account management and repository operations. Below the header is a navigation bar with 'Dashboard' selected.

The main content area is divided into several sections:

- Top Repositories**: A list of repositories owned by the user, including 'nabendu82/Sneha-MERN', 'nabendu82/Ankit-MERN', 'nabendu82/MERN-Jun-23', 'nabendu82/MERN-Apr-23', 'nabendu82/BSH', 'nabendu82/hashedtech 2', and 'nabendu82/MERN-Physical'. There is also a 'New' button and a search bar for finding repositories.
- For you [beta]**: This section features a prominent 'Welcome to the new feed!' message with a note about updating cards and ranking. It includes a 'Send feedback' button and a 'Trending repositories' section with a 'Show more' link.
- Following**: A section showing the latest changes from followed users and repositories.
- Latest changes**: A list of recent events and updates, such as 'Log in with multiple GitHub accounts on GitHub Mobile' (10 hours ago), 'New datalow API for writing custom CodeQL queries' (2 days ago), 'GitHub Copilot August 10th Update' (4 days ago), and 'Secret scanning supports on-demand token validity checks' (4 days ago). Each item has a 'View changes' link.
- Explore repositories**: A section for discovering new repositories, featuring 'modelscope / facechain' (Python, 1.5k stars) and 'ykb / sketch-threejs' (JavaScript, 2.1k stars).
- Recommended for you**: A section showing recommended projects, including 'akella / webgl-mouseover-effects' (JavaScript, 360 stars) and 'sjoerdvanBommel / threeviewer' (JavaScript, 38 stars).

Figure 13.1: GitHub

Now, click on the Plus sign (+) in the top right corner. And then in the dropdown, click on New repository.

The screenshot shows the GitHub desktop application interface. At the top, there's a header bar with the GitHub logo, a search bar containing 'github.com', and various icons for account management and repository operations. Below the header is a navigation bar with 'Dashboard' selected. The main area displays a 'For you' feed with several cards:

- Welcome to the new feed!**: A card with a message about updating the feed and a 'Send feedback' button.
- Trending repositories**: A card showing the repository `modelscope/facechain` with a brief description: "facechain is a deep learning toolchain for generating your Digital Twin.", written in Python, and 15 stars.
- Recommended for you**: A card showing the repository `akella/webgl-mouseover-effects` with a brief description: "Demos for the tutorial on how to achieve an interactive mouseover/hover effect", written in JavaScript, and 300 stars.
- Explore repositories**: A card showing the repository `ykob/sketch-threejs` with a brief description: "Interactive sketches made with three.js.", written in JavaScript, and 2.1k stars.
- sjerdvBommel/threeweb**: A card showing the repository `sjerdvBommel/threeweb` with a brief description: "YouTube tutorial source code", and 18 stars.

On the right side of the dashboard, there's a vertical sidebar with a 'New repository' button highlighted with a green border. Other options in the sidebar include 'Import repository', 'New codespace', 'New gist', 'New organization', and 'New project'. At the bottom left, there's a link to 'https://github.com/sei/'.

Figure 13.2: New repository

In the next screen give the repo a name like task-mern-frontend. Also, give a nice description. Make sure the repo is public and then click on the Create repository button.

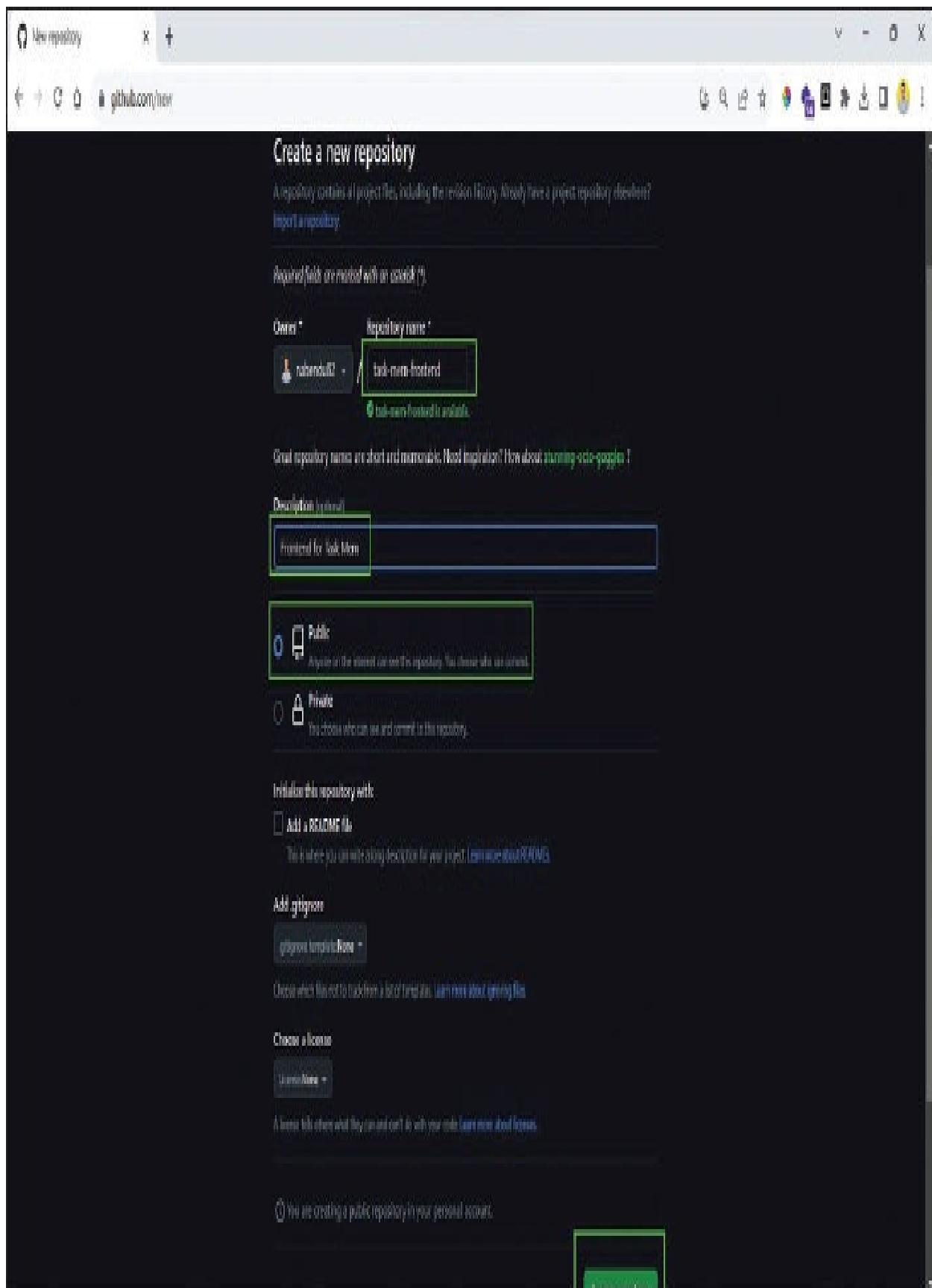


Figure 13.3: Creating a New repository

In the next screen, we will get all the commands to create a repo. Then, push the code to GitHub.

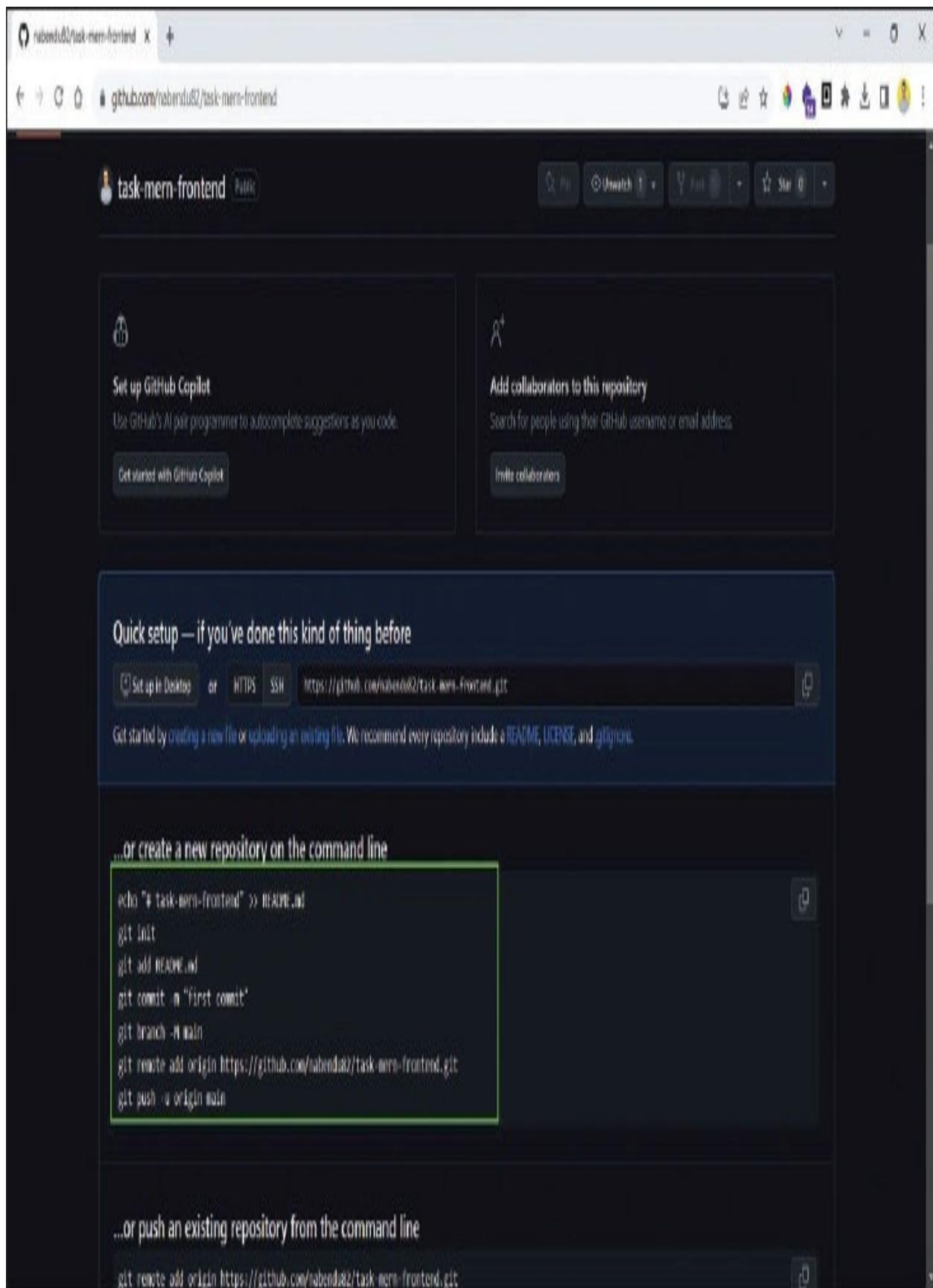


Figure 13.4: Commands for the New repository

Now, from the terminal (Windows, Mac or Linux), first go to the folder where our frontend code is located. After that, enter the following commands. Here, the first command is creating a new git repository.

The second command is adding all the files to the staging area. Note that both the commands can be found in the preceding screenshot, although a bit modified.

git init

git add.

```
MINGW64/c/Users/naben/OneDrive/Desktop/Task-MERN/frontend
haben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (master)
$ git init
Reinitialized existing Git repository in C:/Users/naben/OneDrive/Desktop/Task-MERN/frontend/.git/
haben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (master)
$ git add .
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in README.md.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in package-lock.json.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in package.json.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in public/index.html.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in public/manifest.json.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in public/robots.txt.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/App.js.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/app/store.js.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/index.css.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/index.js.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in src/setupTests.js.
The file will have its original line endings in your working directory

haben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (master)
$
```

Figure 13.5: git init and add

Next, we will run the command to commit the code in the staging area to the commit area. Notice, here we can give any message and we have given a Deployment commit.

```
git commit -m "Deployment commit"
```

MINGW64:/c/Users/haben/OneDrive/Desktop/Task-MERN/frontend

```
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (master)
$ git commit -m "Deployment commit"
[master bld4626] Deployment commit
 34 files changed, 1606 insertions(+), 548 deletions(-)
  create mode 100644 .babelrc
  rewrite public/index.html (64%)
  delete mode 100644 src/App.css
  rewrite src/App.js (95%)
  create mode 100644 src/app/store.test.js
  create mode 100644 src/components/Dashboard.js
  create mode 100644 src/components/Header.js
  create mode 100644 src/components/Login.js
  create mode 100644 src/components/Register.js
  create mode 100644 src/components/Spinner.js
  create mode 100644 src/components/Spinner.test.js
  create mode 100644 src/components/TaskForm.js
  create mode 100644 src/components/TaskItem.js
  create mode 100644 src/components/TaskItem.test.js
  create mode 100644 src/components/TaskList.js
  create mode 100644 src/features/auth/authService.js
  create mode 100644 src/features/auth/authSlice.js
  delete mode 100644 src/features/counter/Counter.js
  delete mode 100644 src/features/counter/Counter.module.css
  delete mode 100644 src/features/counter/counterAPI.js
  delete mode 100644 src/features/counter/counterSlice.js
  delete mode 100644 src/features/counter/counterSlice.spec.js
  create mode 100644 src/features/tasks/taskService.js
  create mode 100644 src/features/tasks/taskService.test.js
  create mode 100644 src/features/tasks/taskSlice.js
  create mode 100644 src/features/tasks/taskSlice.test.js
  rewrite src/index.css (93%)
  rename src/{App.test.js => index.test.js} (58%)
  delete mode 100644 src/logo.svg
  delete mode 100644 src/reportWebVitals.js
```

```
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (master)
```

\$

Figure 13.6: git commit

Next, we will first give the command to create a branch called main. The next command will add the unique remote path given to us by GitHub. The last push command will push our code from the local system to GitHub.

Note that all of these commands are exactly the same as what we have got from our GitHub command list earlier.

```
git branch -m main
```

```
git remote add origin https://github.com/nabendu82/task-mern-frontend.git
```

```
git push -u origin main
```

```
MINGW64/c/Users/naben/OneDrive/Desktop/Task-MERN/frontend
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (master)
$ git branch -M main

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)
$ git remote add origin https://github.com/nabendu82/task-mern-frontend.git

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)
$ git push -u origin main
Enumerating objects: 66, done.
Counting objects: 100% (66/66), done.
Delta compression using up to 12 threads
Compressing objects: 100% (64/64), done.
Writing objects: 100% (66/66), 193.31 KiB | 6.44 MiB/s, done.
Total 66 (delta 8), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (8/8), done.
To https://github.com/nabendu82/task-mern-frontend.git
 * [new branch]    main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)
$ |
```

Figure 13.7: git push

Back in GitHub, we will find all of our code being uploaded in our repo.

task-mern-frontend X +

github.com/nabendu82/task-mern-frontend

nabendu82 / task-mern-frontend

Type ⌘ to search

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

task-mern-frontend Public

main · 1 branch · 0 tags

Go to file Add file · Code About

Deployment commit	Author	Time Ago
Deployment commit	nabendu82	2 minutes ago
Deployment commit	public	2 minutes ago
Deployment commit	src	2 minutes ago
Deployment commit	babelrc	2 minutes ago
Initialize project using Create React App	.gitignore	2 months ago
Initialize project using Create React App	README.md	2 months ago
Deployment commit	package-lock.json	2 minutes ago
Deployment commit	package.json	2 minutes ago

Frontend for Task Mern

Readme · Activity · 0 stars · 1 watching · 0 forks

No releases published · Create a new release

README.md

Getting Started with Create React App and Redux

This project was bootstrapped with [Create React App](#), using the [Redux](#) and [Redux Toolkit](#) template.

No packages published · Publish your first package

Languages

The screenshot shows a GitHub repository page for 'task-mern-frontend'. The repository is public and has one branch ('main') and no tags. The last commit was a deployment commit by 'nabendu82' 2 minutes ago. The repository's purpose is described as 'Frontend for Task Mern'. The README file contains the text 'Getting Started with Create React App and Redux' and notes that it was bootstrapped with Create React App using the Redux and Redux Toolkit template. The repository has 0 stars, 1 watching, and 0 forks. There are no releases published.

Figure 13.8: Frontend repo

Backend code repo creation

We will now create a repo for our backend code and transfer all code from the local computer to this remote GitHub repository. The steps will be exactly the same as the Frontend part.

Now, to create a new repo, we will again click on the plus(+) sign and then the New repository.

task-mern-frontend

github.com/nabendu82/task-mern-frontend

nabendu82 / task-mern-frontend

Type ⌘ to search

+ New repository Import repository New codespace New gist New organization

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

task-mern-frontend Public

main · 1 branch · 0 tags

Go to file Add file + Code

Deployment commit · 11 hours ago · 2 commits

Deployment commit · 11 hours ago

Deployment commit · 11 hours ago

Deployment commit · 11 hours ago

Initialize project using Create React App · 2 months ago

Initialize project using Create React App · 2 months ago

Deployment commit · 11 hours ago

Deployment commit · 11 hours ago

README.md

Releases

No release published

Create a new release

Packages

No packages published

Get started with Create React App and Redux

This project was bootstrapped with [Create React App](#), using the [Redux](#) and [Redux Toolkit](#) template.

https://github.com/nabendu82/task-mern-frontend

Figure 13.9: New repository

In the next screen, we will give the repository a name and description and then click on the Create repository button.

New repository X +

[Create a new repository](#)

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner* **Repository name***

`task-mem-backend` / `task-mem-backend`

`task-mem-backend` is available.

Great repository names are short and memorable. Need inspiration? How about `curly-train`?

Description (optional)

`Backend for Task Mem`

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with

Add a README file
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

`.gitignore template` *

Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license

`LicenseNone` *

A license tells others what they can and can't do with your code. Learn more about licenses.

You are creating a public repository in your personal account.

Create repository

Figure 13.10: Backend repository

In the next screen, we will again get the set of commands to be given in the terminal. Note that here the path is different in the remote command with a unique link.

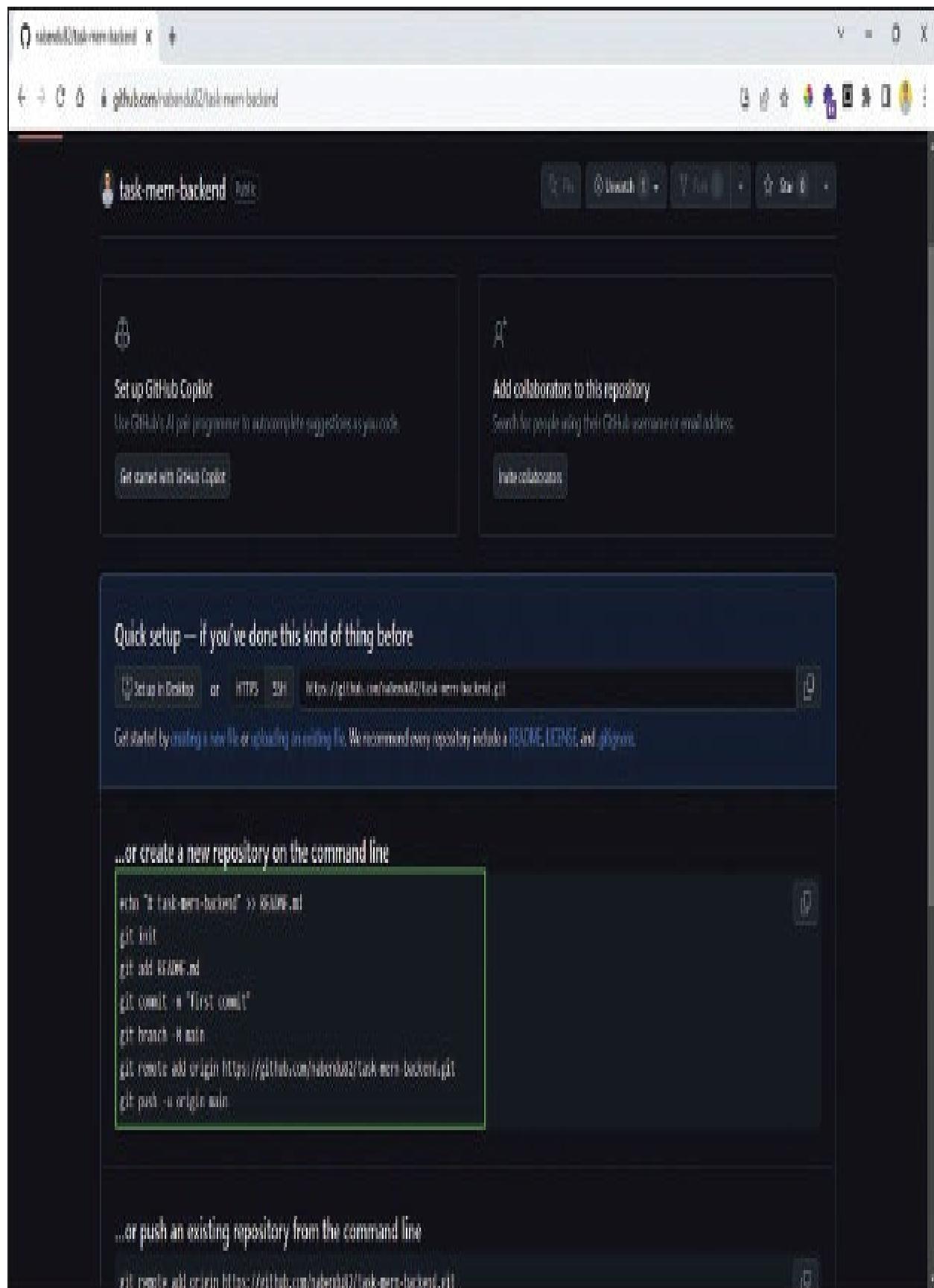


Figure 13.11: Backend repository commands

Again, we will enter the following three commands first. The first one is to initialize a Git repo, the second is to put the files in the staging area, and the third is to put the files in the commit area.

git init

git add.

git commit -m “Deployment commit”

```
MINGW64:/c/Users/naben/OneDrive/Desktop/Task-MERN/backend - X
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend
$ git init
Initialized empty Git repository in C:/Users/naben/OneDrive/Desktop/Task-MERN/backend/.git/
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (master)
$ git add .
warning: LF will be replaced by CRLF in jest.config.js.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in package-lock.json.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in package.json.
The file will have its original line endings in your working directory

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (master)
$ git commit -m "Deployment Commit"
[master (root-commit) ee5298c] Deployment Commit
16 files changed, 8784 insertions(+)
create mode 100644 .gitignore
create mode 100644 __tests__/taskController.test.js
create mode 100644 __tests__/userController.test.js
create mode 100644 connect/database.js
create mode 100644 controllers/taskController.js
create mode 100644 controllers/userController.js
create mode 100644 jest.config.js
create mode 100644 middleware/authMiddleware.js
create mode 100644 middleware/errorMiddleware.js
create mode 100644 models/taskModel.js
create mode 100644 models/userModel.js
create mode 100644 package-lock.json
create mode 100644 package.json
create mode 100644 routes/taskRoutes.js
create mode 100644 routes/userRoutes.js
create mode 100644 server.js

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (master)
$
```

Figure 13.12: git init, add and commit

The next three commands will be to create a branch, adding a remote path and pushing the code from the local computer to GitHub.

```
git branch -m main
```

```
git remote add origin https://github.com/nabendu82/task-mern-backend.git
```

```
git push -u origin main
```

```
MINGW64/c/Users/haben/OneDrive/Desktop/Task-MERN/backend - X
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (master)
$ git branch -M main

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)
$ git remote add origin https://github.com/nabendu82/task-mern-backend.git

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)
$ git push -u origin main
Enumerating objects: 24, done.
Counting objects: 100% (24/24), done.
Delta compression using up to 12 threads
Compressing objects: 100% (22/22), done.
Writing objects: 100% (24/24), 90.21 KiB | 6.44 MiB/s, done.
Total 24 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/nabendu82/task-mern-backend.git
 * [new branch]    main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)
$
```

Figure 13.13: Git push

Now, our backend code is also deployed successfully in GitHub.

nabendu2@task-mern-backend ~ +

github.com/nabendu2/task-mern-backend

nabendu2 / task-mern-backend

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

task-mern-backend Public

main · 1 branch · 0 tags

Go to file Add file + Code

File	Type	Commit
index.js	Deployment Commit	6e529fc · 2 minutes ago
node_modules	Deployment Commit	6e529fc · 2 minutes ago
public	Deployment Commit	6e529fc · 2 minutes ago
src	Deployment Commit	6e529fc · 2 minutes ago
controller	Deployment Commit	6e529fc · 2 minutes ago
model	Deployment Commit	6e529fc · 2 minutes ago
route	Deployment Commit	6e529fc · 2 minutes ago
util	Deployment Commit	6e529fc · 2 minutes ago
test	Deployment Commit	6e529fc · 2 minutes ago
config	Deployment Commit	6e529fc · 2 minutes ago
lock	Deployment Commit	6e529fc · 2 minutes ago
package	Deployment Commit	6e529fc · 2 minutes ago
server	Deployment Commit	6e529fc · 2 minutes ago

About

Backend for Task Mern

Activity

0 stars

1 watching

0 forks

Releases

No release published

Create a new release

Packages

No packages published

Publish your first package

Languages

Figure 13.14: Backend repository

Backend deployment in Back4app

We will first deploy our backend code to an awesome free service called Back4app. So, first, head over to <https://www.back4app.com/> and Sign up with Google.

The screenshot shows a web browser window with two tabs open: 'nabendu82/task-mem-backend' and 'Back4App - Low-code backend'. The main content area displays the 'back4app' website. At the top, there's a navigation bar with links for 'Products', 'Pricing', 'Docs', 'Support', 'Log in', and a prominent green 'Sign up' button. Below the navigation, a large heading reads 'Low-code backend to build modern apps'. A sub-section below it says 'Store and query relational data on the cloud. Make it accessible over GraphQL and REST with a scalable, open-source backend.' To the right, there's a section titled 'Build your app backend in minutes' with a note 'No credit card required' and two sign-up buttons: 'Sign up with Google' and 'Sign up with Github'. Further down, there are fields for 'E-mail' and 'Password', and a green 'Sign up with e-mail' button. At the bottom, a small note states 'By signing up you agree to our [Terms of Service](#) and [Privacy Policy](#)'. On the left side of the main content area, there are three performance metrics: '+3.5B/mo' (Processed Requests), '+100' (Countries Served), and '+190k' (Developers), each accompanied by a small icon.

back4app

Products Pricing Docs Support Log in Sign up

Low-code backend to build modern apps

Store and query relational data on the cloud. Make it accessible over GraphQL and REST with a scalable, open-source backend.

How it works

+3.5B/mo +100 +190k

Processed Requests Countries Served Developers

Build your app backend in minutes

No credit card required

Sign up with Google

Sign up with Github

E-mail

Password

Sign up with e-mail

By signing up you agree to our [Terms of Service](#) and [Privacy Policy](#)

Help

Figure 13.15: Back4app

Next, click on the NEW APP button in the top right corner.



Figure 13.16: New app

In the next screen, click on Containers as a Service banner.

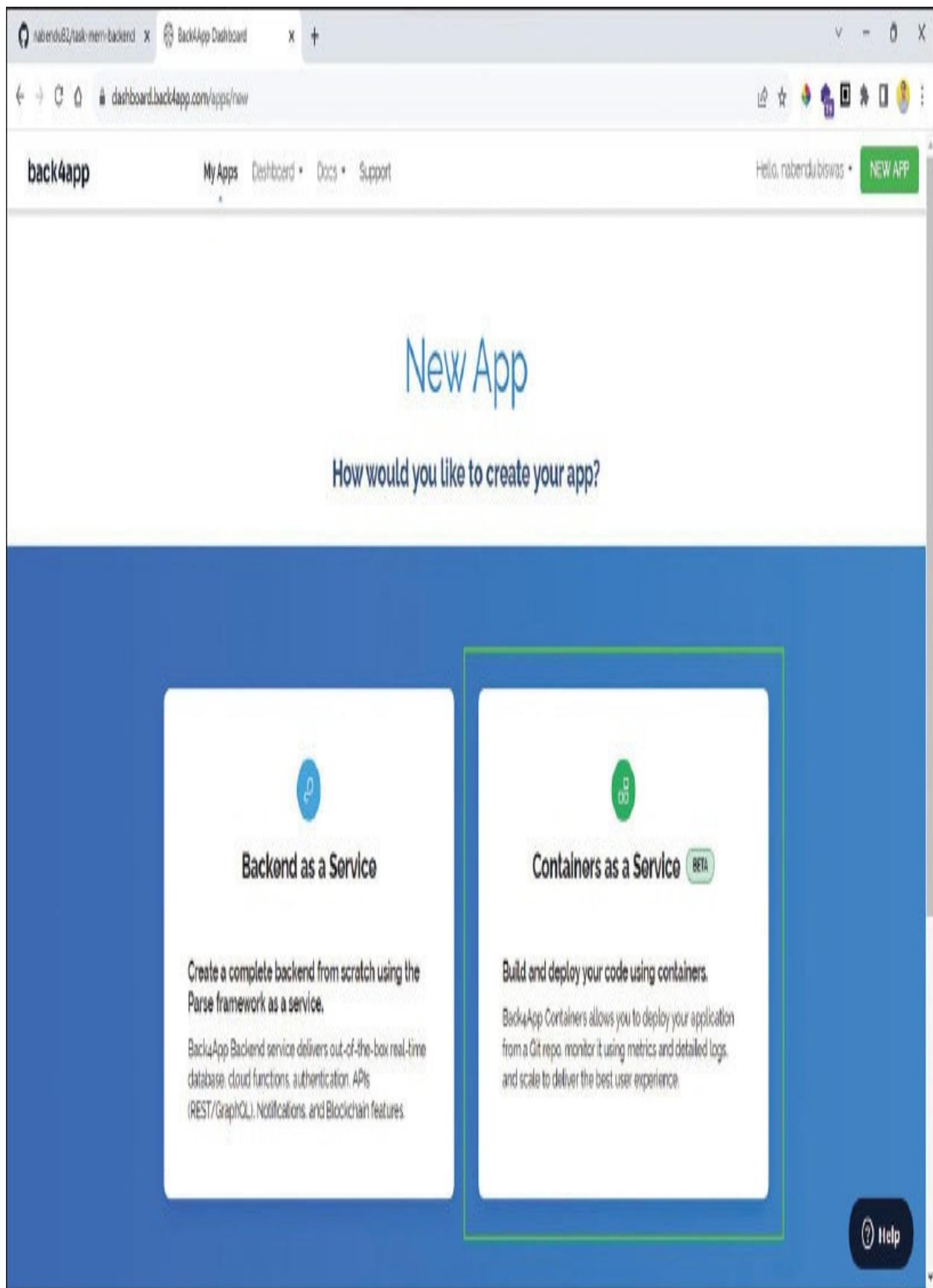


Figure 13.17: Containers as a Service

It will now ask us to select a Git repository containing our code.

nabendu82/task-mern-backend X BackApp Containers Dashboard X +

containers.back4app.com/new-container

back4app My Apps Dashboard Docs Support Hello, nabendu.biswas New app

New App from source code

To create a new Docker container App from source code, select an existing Git repository.

Select a Git repository

	Search	
 nabendu82/task-manager-night	11 days ago	Select
 nabendu82/Ankit-MERN	12 days ago	Select
 nabendu82/tinder-backend-ankit	12 days ago	Select
 nabendu82/MERN-Apr-23	12 days ago	Select
 nabendu82/MERN-Physical	13 days ago	Select
 nabendu82/task-managor-ankit	14 days ago	Select

Help

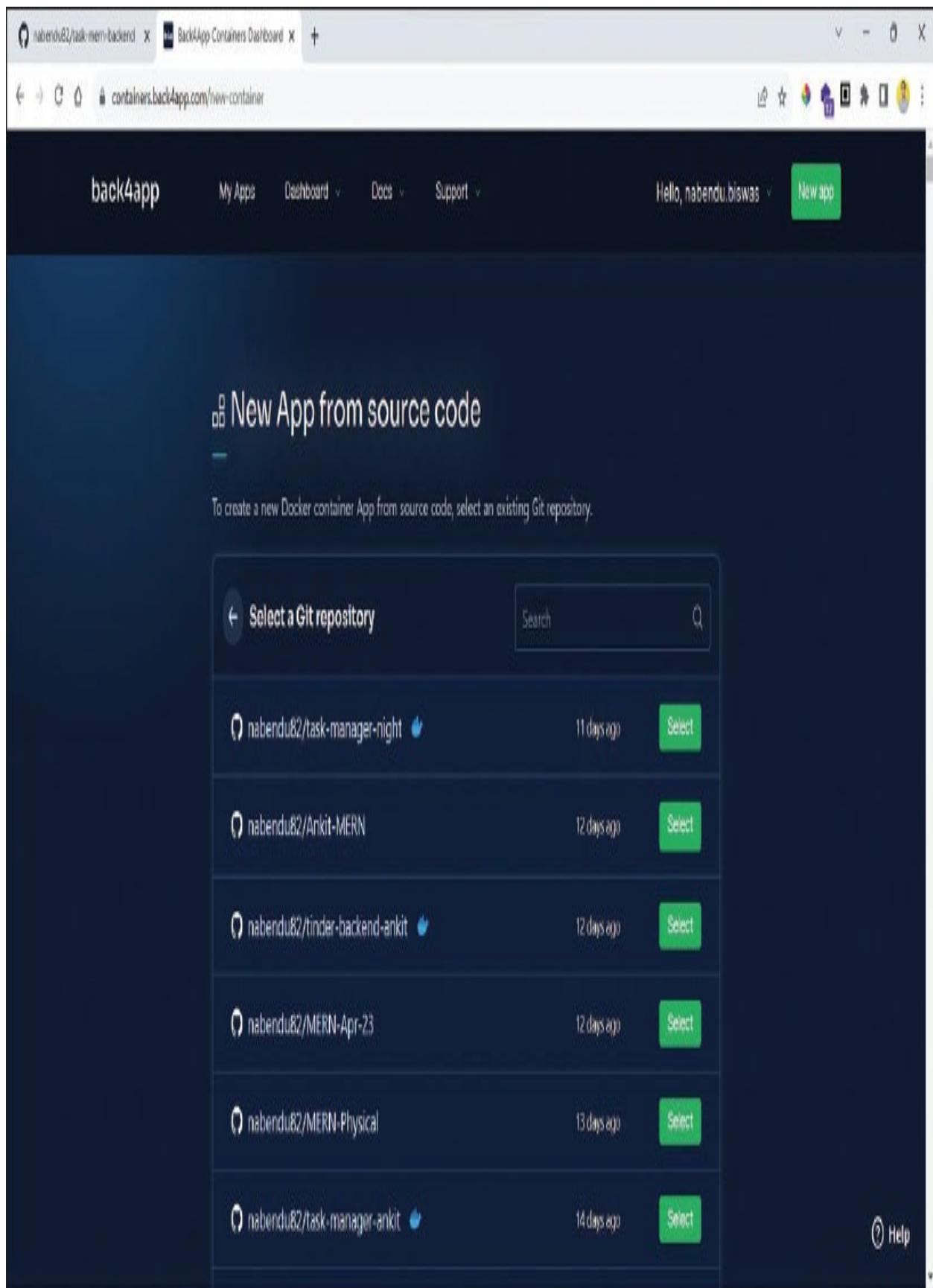


Figure 13.18: New repo addition

Now, we will search for our backend repository by typing task-mern. But since the permission is not given, we will not find it. Also, notice that a message of Dockerfile is mandatory is shown.

The screenshot shows a web browser window for the back4app platform. The URL in the address bar is `containers.back4app.com/new-container`. The page title is "New App from source code". The top navigation bar includes links for "My Apps", "Dashboard", "Docs", "Support", and a user profile with the name "Hello, nabandu.biswas". A green "New app" button is also visible.

The main content area displays a search bar with the placeholder text "Select a Git repository" and a dropdown menu containing the item "task-mem". Below the search bar, there is a link to "Edit GitHub Permissions". A note at the bottom of the form states: "A Dockerfile is mandatory for deploying your applications; please ensure it's included in your repo."

Figure 13.19: Searching for new repo

So, we need to first go back to our backend code and add a file Dockerfile in the root directory. Add the below content to it. Note that it should expose the correct port on which our backend code is running.

```
# Use the official Node.js image as the base image
FROM node:18

# Set the working directory in the container
WORKDIR /app

# Copy the application files into the working directory
COPY . /app

# Install the application dependencies
RUN npm install

EXPOSE 8000

# Define the entry point for the container
CMD ["npm", "start"]
```


A screenshot of the Visual Studio Code interface showing a file named "Dockerfile" open in the editor. The code is a Dockerfile for a Node.js application, defining a base image, working directory, copying application files, installing dependencies, exposing port 8000, and setting the entry point to "npm start". The "backend" folder in the project explorer also contains "server.js", ".env", and ".gitignore".

```
1 # Use the official Node.js image as the base image
2 FROM node:18
3
4 # Set the working directory in the container
5 WORKDIR /app
6
7 # Copy the application files into the working directory
8 COPY . /app
9
10 # Install the application dependencies
11 RUN npm install
12
13 EXPOSE 8000
14 # Define the entry point for the container
15 CMD ["npm", "start"]
```

Figure 13.20: Adding Dockerfile

Since we have added a new file in our backend code, we need to push again this changed code to our GitHub repo. So, again we need to give the add, commit and push commands.

git add.

git commit -m “Dockerfile added”

git push

```
MINGW64:/c/Users/naben/OneDrive/Desktop/Task-MERN/backend - X
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)
$ git add .
naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)
$ git commit -m "Dockerfile added"
[main 2e6e17c] Dockerfile added
1 file changed, 15 insertions(+)
create mode 100644 Dockerfile

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 492 bytes | 492.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/nabendu82/task-mern-backend.git
 ee5298c..2e6e17c main -> main

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)
$
```

Figure 13.21: git push

Now, in GitHub for our backend code, we will find the Dockerfile.

← → ⌂ ⌂ github.com/nabendu82/task-mern-backend

≡  nabendu82 / task-mern-backend

Type ⌘ to search

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

task-mern-backend Public

main · 1 branch · 0 tags

Go to file Add file ⌂ Code

nabendu82 Dockerfile added · 2 commits · 1 minute ago

tests Deployment Commit · 10 minutes ago

connect Deployment Commit · 10 minutes ago

controllers Deployment Commit · 10 minutes ago

middleware Deployment Commit · 10 minutes ago

models Deployment Commit · 10 minutes ago

routes Deployment Commit · 10 minutes ago

.gitignore Deployment Commit · 10 minutes ago

Dockerfile Dockerfile added · 1 minute ago

jest.config.js Deployment Commit · 10 minutes ago

package-lock.json Deployment Commit · 10 minutes ago

package.json Deployment Commit · 10 minutes ago

server.js Deployment Commit · 10 minutes ago

About Backend for Task Mern

Activity 0 stars 1 watching 0 forks

Releases No releases published Create a new release

Packages No packages published Publish your first package

Languages

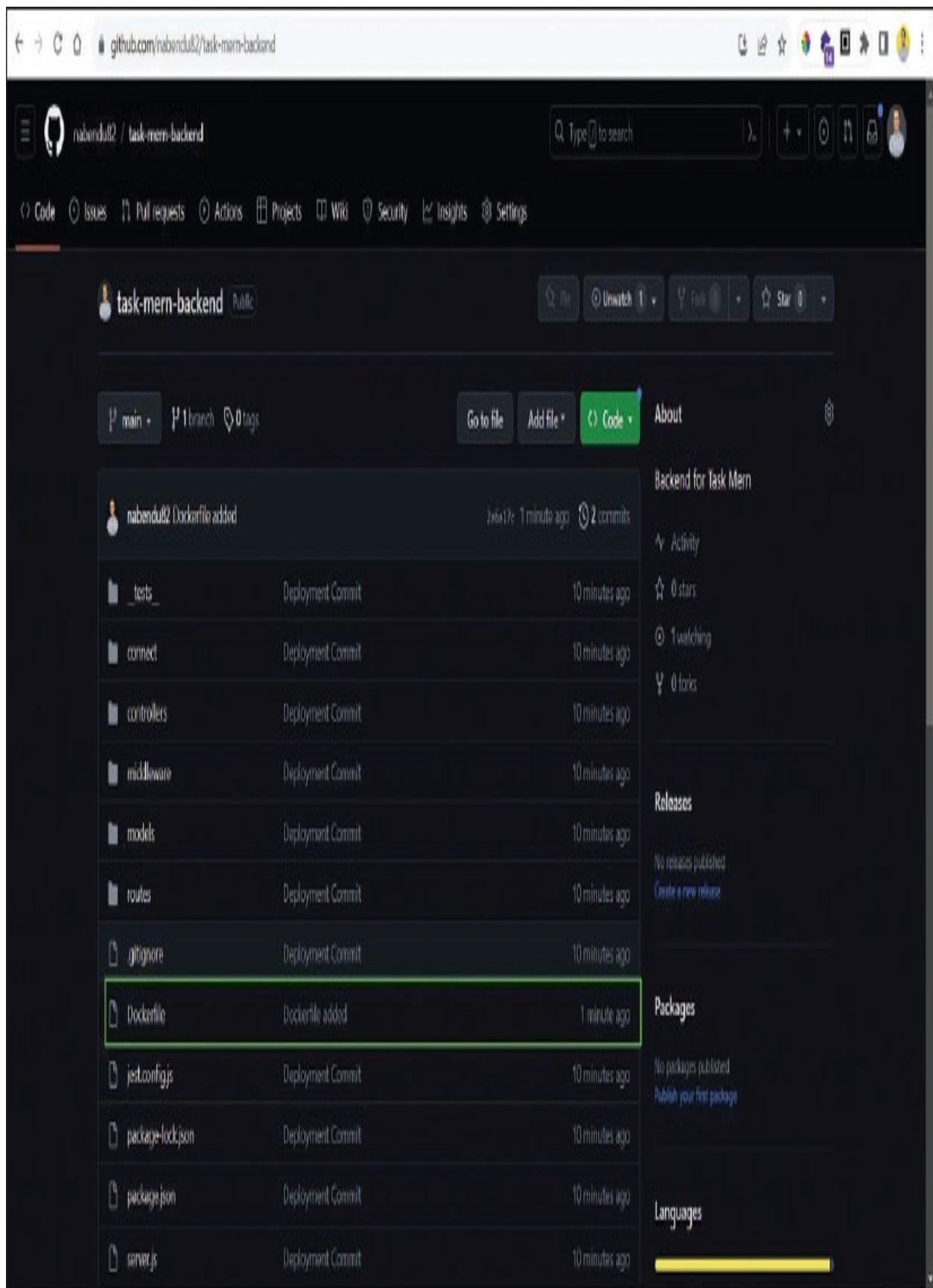


Figure 13.22: Dockerfile in GitHub

Back in Back4app, click on the Edit GitHub Permissions button.

The screenshot shows a web browser window with the URL `containers.back4app.com/new-container`. The page is titled "New App from source code". At the top, there is a navigation bar with links for "My Apps", "Dashboard", "Docs", "Support", and a user profile "Hello, nabendu.biswas". A green button labeled "New app" is visible on the right. Below the navigation, there is a search bar with the placeholder "task-mern" and a magnifying glass icon. A message says "Not seeing the repositories you expected here?". A blue button labeled "Edit GitHub Permissions" is highlighted with a green border. A note at the bottom states: "⚠ A Dockerfile is mandatory for deploying your applications, please ensure it's included in your repo." In the bottom right corner, there is a "Help" link.

containers.back4app.com/new-container

back4app

My Apps Dashboard Docs Support Hello, nabendu.biswas New app

New App from source code

To create a new Docker container App from source code, select an existing Git repository.

Select a Git repository task-mern

Not seeing the repositories you expected here?

Edit GitHub Permissions

⚠ A Dockerfile is mandatory for deploying your applications, please ensure it's included in your repo.

Help

Figure 13.23: Adding backend code

We will get a pop-up in which we need to select the right account.

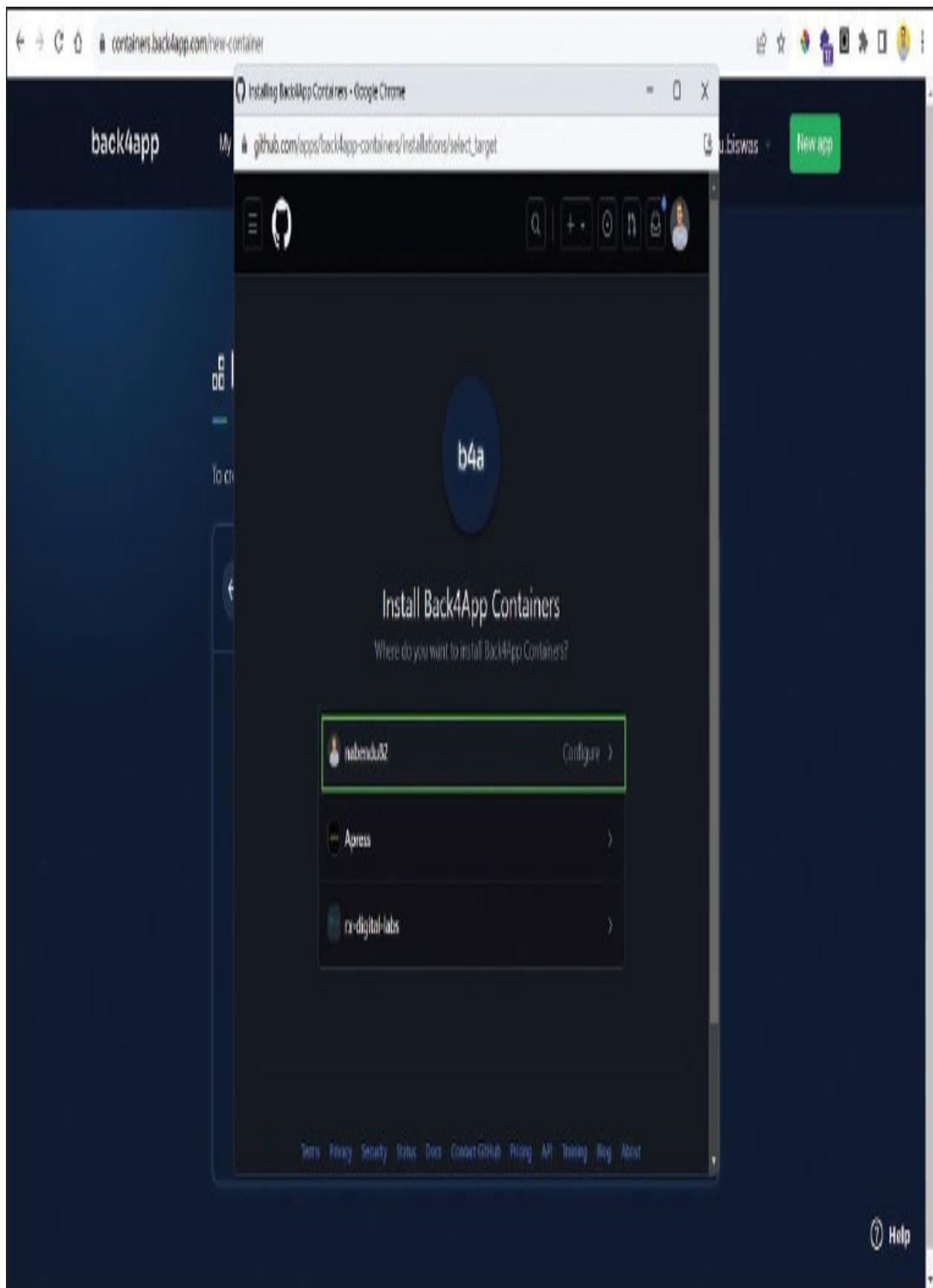


Figure 13.24: Selecting the GitHub account

In the next screen, we will select our backend repository of task-mern-backend.

Localed GitHub App - Back4App Containers - Google Chrome

github.com/settings/installations/36395990

Back4App Containers is requesting an update to its permissions

Review request

Emails

>Password and authentication

Sessions

SSH and GPG keys

Organizations

Enterprises

Moderation

Repository access

Code, planning, and automation

Repositories

Codespaces

Packages

Copilot

Pages

Saved replies

Security

Code security and analysis

Integrations

Applications

Scheduled reminders

Archives

Security log

Read access to code and metadata

✓ Read and write access to commit statuses

All repositories

This applies to all current and future repositories owned by the resource owner.
Also includes public repositories (read-only).

Only select repositories

Select at least one repository.
Also includes public repositories (read-only).

Select repositories

task-mem

nabenbuli/task-mem-backend
Backend for Task Mem

nabenbuli/task-mem-frontend
Frontend for Task Mem

Save

Danger zone

The screenshot shows the GitHub App permissions settings for the Back4App Containers app. It highlights the 'Repository access' section where the 'Only select repositories' option is selected. A modal window displays a list of repositories: 'task-mem' (selected), 'nabenbuli/task-mem-backend' (Backend for Task Mem), and 'nabenbuli/task-mem-frontend' (Frontend for Task Mem). The 'task-mem' repository is highlighted with a green border. At the bottom of the modal is a 'Save' button.

Figure 13.25: Adding backend repo

Next, click on the Save button.

Index of this App · BackApp Containers · Google Chrome

github.com/settings/installations/939399

BackApp Containers is requesting an update to its permissions

Review request

① (mail)

① Password and authentication

① Sessions ✓ Read access to code and metadata

② SSH and GPG key ✓ Read and write access to current database

② Organizations

② Enterprises

② Motivation

Repository access

Code, planning and automation

② Repositories

② Codepaces

② Packages

② Copilot

② Pages

① Saved replies

All repositories

The app has full control of all repositories owned by the selected owner.
Also includes public repositories (local only).

Only select repositories

Select at least one repository.
Also includes public repositories (local only).

Select repositories

Selected 1 repository.

huboard/02-task-man-backend

Save Cancel

Danger zone

Figure 13.26: Saving backend repo

We will get back to Back4app and now we will see our repository. Here, click on the Select button.

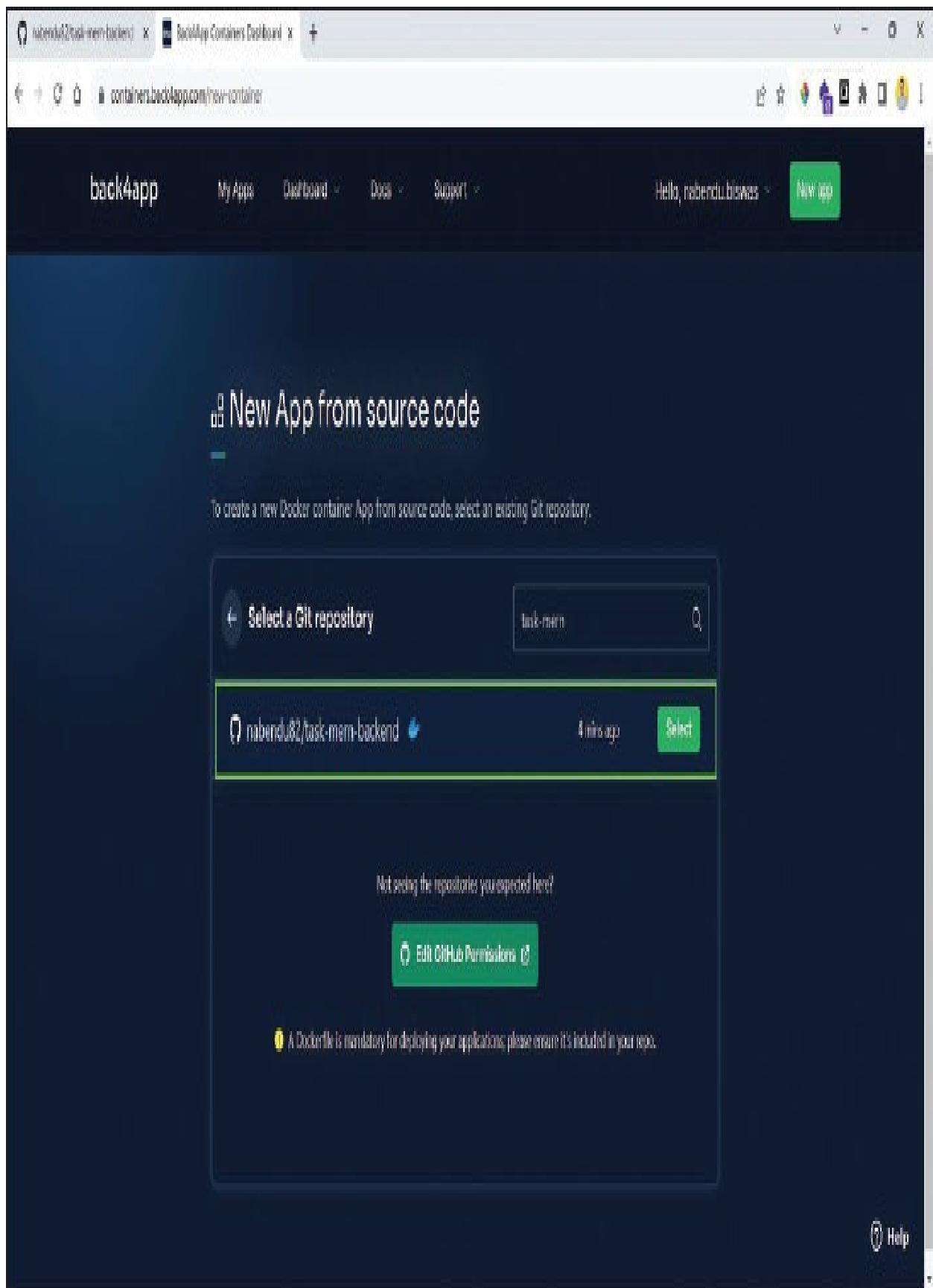


Figure 13.27: Selecting repo in Back4app

Since, our backend code also contains environment variables in the .env file, we need to give them in the next screen.

Screenshot of the Back4App Dashboard showing the configuration for a new container named "task-mern-backend".

The dashboard includes a header with tabs for "Dashboard" and "Container", and a top bar with user information and navigation icons.

The main area shows the "back4app" project with a "New app" button.

Configuration details:

- App Name:** task-mern-backend
- Build and Deploy:** (Collapsible section)
- Environment Variables (Optional):** (Collapsible section)
 - Name:** PORT **Value:** 3001
 - Name:** MONGO_URI **Value:** **REDACTED**
 - Name:** JWT_SECRET **Value:** **REDACTED**
- Add variable** button
- Help** link

Figure 13.28: Adding environment variables

After that, scroll down and click on the Create App button.

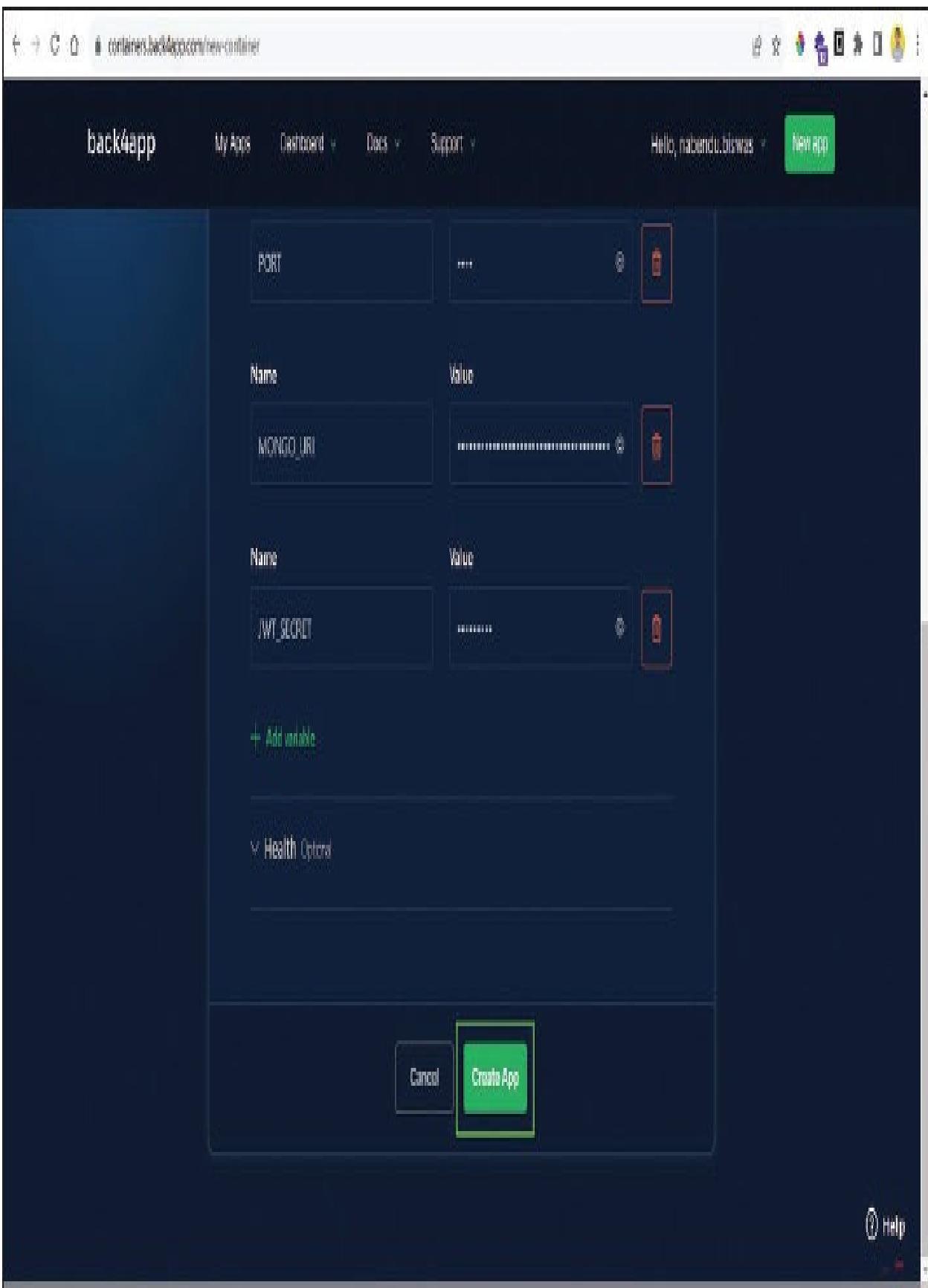


Figure 13.29: Create App

In the next screen, we will see all the logs when the deployment happens.

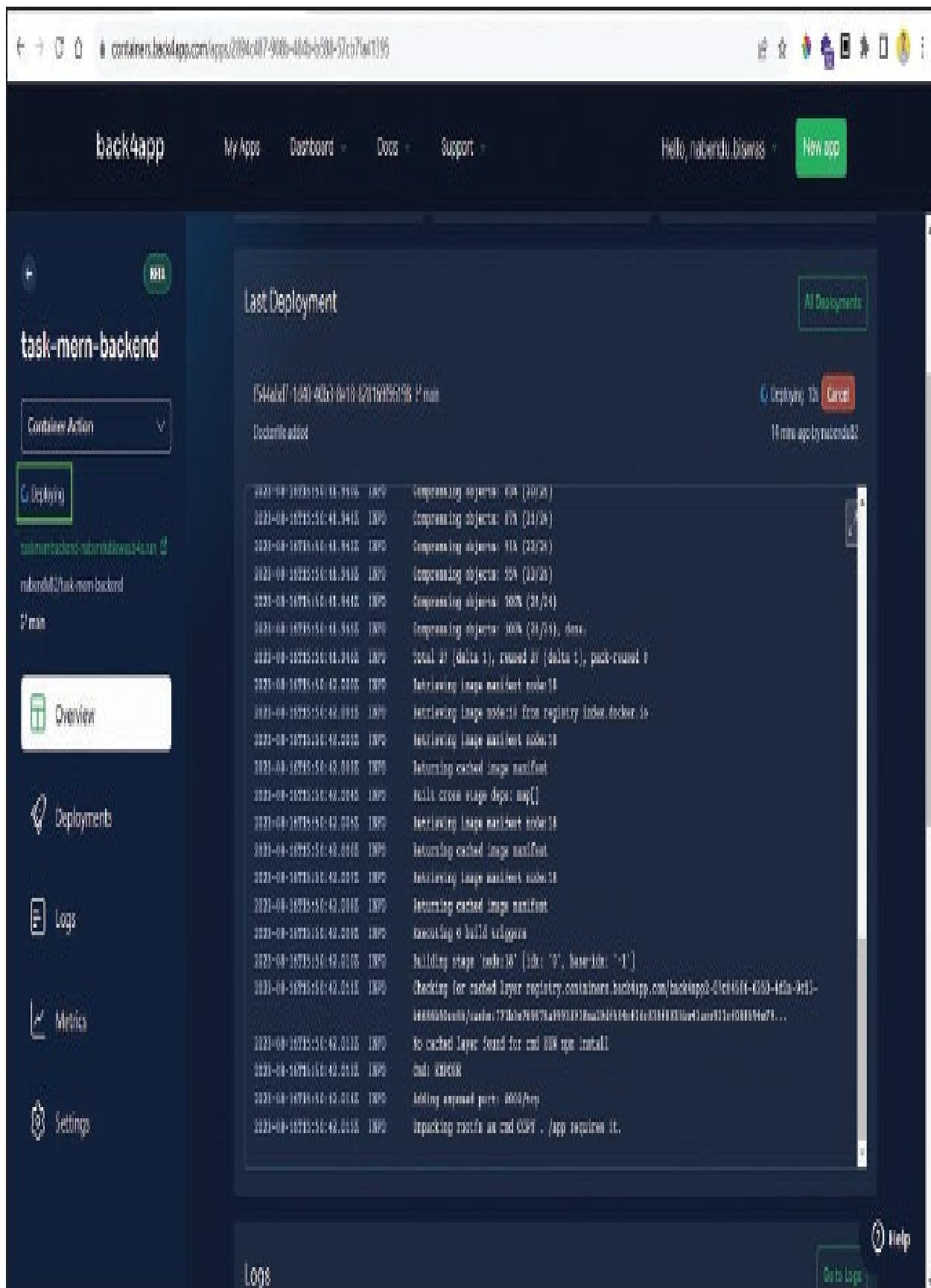


Figure 13.30: Deploying in Back4app

After some time, we will get the DEPLOYMENT READY message on the logs. We will also get the link to the deployment on the left side.

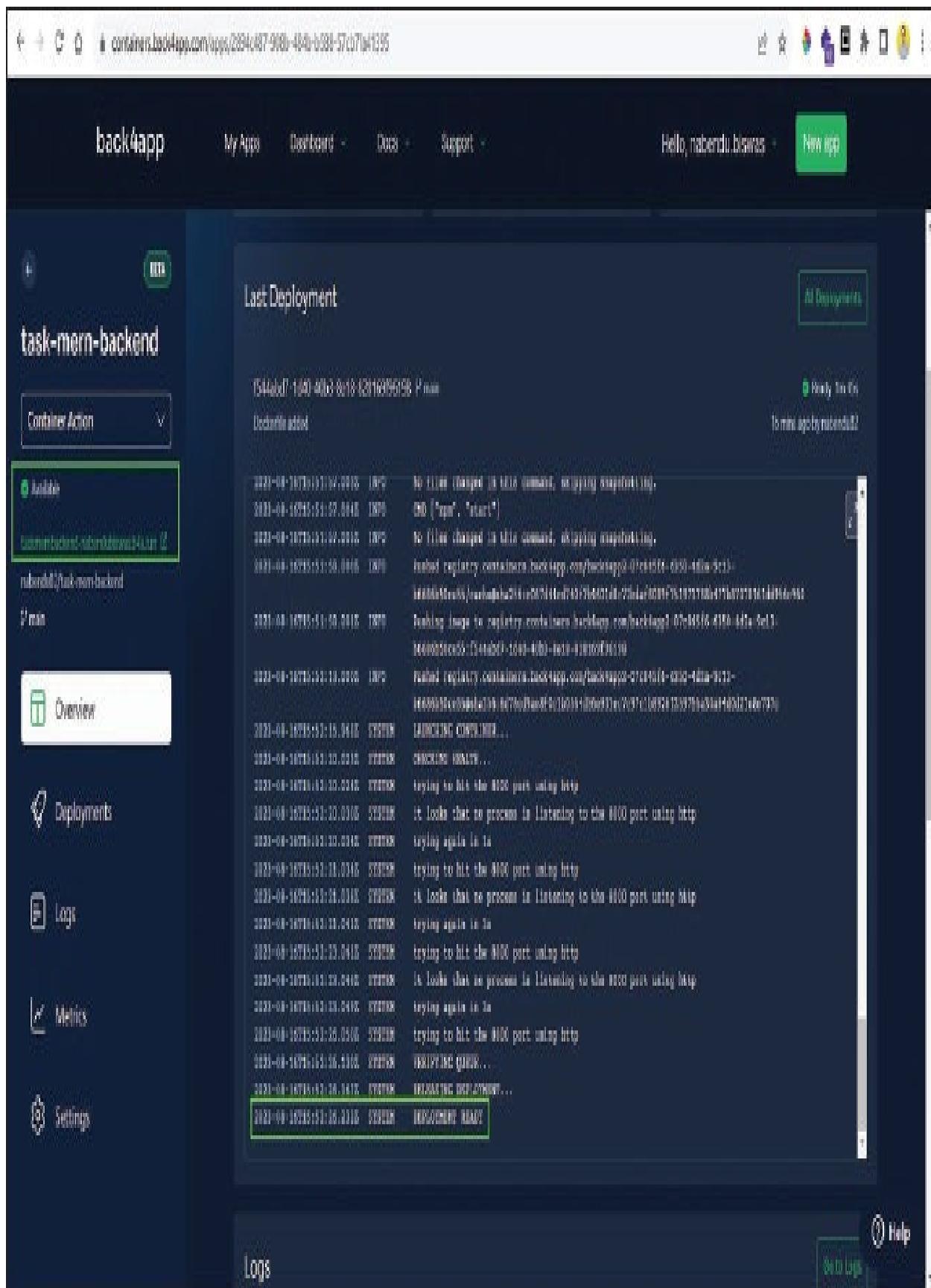


Figure 13.31: Deployment successful

Backend integration with frontend

We will now do the backend integration with the front end. But first, take the newly created backend endpoint and test the Register User route in Postman.

Here, we are able to use this route to create new users.

Home Workspaces API Network Explore

Search Postman

No Environment

MERN-Task | Register User

POST https://taskmenbackend-nabendubsws.b4s.run/api/users

Params Authorization Headers Body Pre-request Script Tests Settings Cookies

Body (Form-data)

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> name	Robin	
<input checked="" type="checkbox"/> email	robin@gmail.com	
<input checked="" type="checkbox"/> password	robin12345	

Body Cookies Headers (10) Test Results

Status: 201 Created Time: 2.82 s. Size: 673 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "_id": "64dcf31ab2d389f5a811bac1",
3   "name": "Robin",
4   "email": "robin@gmail.com",
5   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVC9.yJzC16IjYVQGhWzfIyJ2RjgS2jVh08xYnFjMS1sInIhdCfOMYSHjlaMrc1Sna1Z3mzjno1kjyijNekzU1fO.15eb-XMFYDWW0dc5YvUTsE$4VzA-5aPZNPVfc5b18c"
6 }
```

Online Find and Replace Console

Cookies Capture requests Beoncamp Runner Trash

The screenshot shows the Postman interface with a registered user. The 'Body' tab is selected, showing a form-data structure with three fields: 'name' (value: Robin), 'email' (value: robin@gmail.com), and 'password' (value: robin12345). The 'Pretty' tab is selected in the preview section, displaying the JSON response. The response body is a single object with properties: '_id' (value: 64dcf31ab2d389f5a811bac1), 'name' (value: Robin), 'email' (value: robin@gmail.com), and 'token' (value: a complex JWT token). The status bar at the bottom indicates the request took 2.82 seconds and had a size of 673 bytes.

Figure 13.32: Register user in Postman

We are also able to log in with this newly created user in the login route.

Home Workspaces API Network Explore

Search Postman

No Environment

MERN-Task / Login User

POST https://taskmembackend-nabendubiswas.b4a.run/api/users/login

Params Authorization Headers Body Pre-request Script Tests Settings Cookies

Body (Form-data)

KEY	VALUE	DESCRIPTION	Bulk Edit
email	robin@gmail.com		
password	robin12345		
Key	Value	Description	

Status 200 OK Time: 1078 ms Size: 668 B Save Response

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "_id": "64dcf31bb2028915a811b0e1",
3   "name": "Robin",
4   "email": "robin@gmail.com",
5   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJyZWFyaXplMjM1IiwidXNlcm5hbWUiOiJsb2dpbiIsImV4cCI6MTYwNjQwOTk3LCJleHAiOjE2ODQwMDIwLCJlbWFpbCI6ImFkb2JlLmNvbS5jb20ifQ.eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.CVGATR6gDAmHEOBzVehJLe9s_3Mniz2-E0CZuGe9x0T1"
```

Online Find and Replace Console

Cookies Capture requests Bootcamp Runner Trash

Figure 13.33: Login user in Postman

Now, in the package.json file in proxy, we will change the earlier localhost route with this backend route. Again, don't give / at the end.

File Edit Selection View Go ... package.json - Task-MERN - Visual Studio Code

EXPLORER ... package.json M X

frontend > package.json > proxy

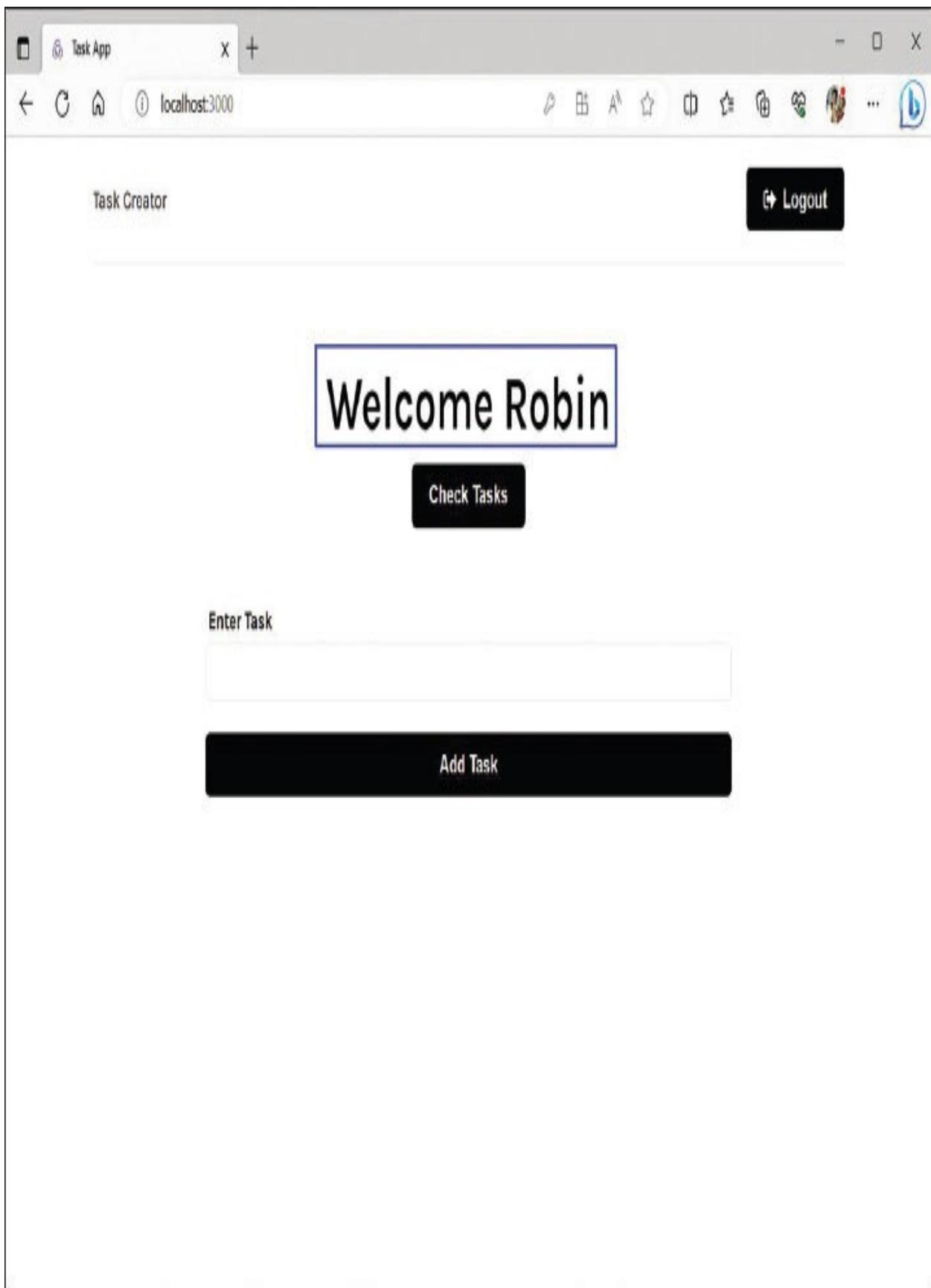
```
1 {
2   "name": "frontend",
3   "version": "0.1.0",
4   "proxy": "https://taskmernbackend-nabendubiswas.b4a.run",
5   "private": true,
6   "dependencies": {
7     "-": "^0.0.1",
8     "@reduxjs/toolkit": "^1.9.5",
9     "@testing-library/jest-dom": "^5.16.5",
10    "@testing-library/react": "^13.4.0",
11    "@testing-library/user-event": "^14.4.3",
12    "axios": "^1.4.0",
13    "axios-mock-adapter": "^1.21.5",
14    "D": "^1.0.0",
15    "history": "^5.3.0",
16    "react": "^18.2.0",
17    "react-dom": "^18.2.0",
18    "react-icons": "^4.9.0",
19    "react-redux": "^8.1.0",
20    "react-router-dom": "^6.14.1",
21    "react-scripts": "5.0.1",
22    "react-toastify": "^9.1.3",
23    "redux-mock-store": "^1.5.4",
24    "web-vitals": "^2.1.4"
}
```

coverage
node_modules
public
src
 app
 components
 features
 App.js
 index.css
 index.js
 index.test.js
 setupTests.js
.babelrc
.gitignore
package-lock.json
package.json M
README.md
OUTLINE
TIMELINE

main* tabnine starter ↻ Ln 4, Col 58 Spaces: 2 UTF-8 LF JSON Go Live ⚡ ⚡

Figure 13.34: Proxy in package.json

Now in `http://localhost:3000/`, we will log in with the newly created user and it will work properly.



Task Creator

Logout

Welcome Robin

Check Tasks

Enter Task

Add Task

Figure 13.35: Login to frontend

Since we have made changes in the frontend code, we will stage, commit, and push this code to GitHub.

git add.

git commit -m “Backend integrated”

git push



MINGW64:c/Users/naben/OneDrive/Desktop/Task-MERN/frontend

- X

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$ git add .

warning: LF will be replaced by CRLF in package-lock.json.

The file will have its original line endings in your working directory

warning: LF will be replaced by CRLF in package.json.

The file will have its original line endings in your working directory

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$ git commit -m "Backend integrated"

[main 82cbf22] Backend integrated

1 file changed, 1 insertion(+), 1 deletion(-)

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$ git push

Enumerating objects: 5, done.

Counting objects: 100% (5/5), done.

Delta compression using up to 12 threads

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.

Total 3 (delta 2), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (2/2), completed with 2 local objects.

To https://github.com/nabendu82/task-mern-frontend.git

b1d4626..82cbf22 main -> main

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$

Figure 13.36: git push

Frontend deployment in Netlify

We will now deploy our project in Netlify. So, open Netlify at <https://www.netlify.com/> and click on the Sign up button, if you don't have an account.

Join us at Compose Conference, October 18-19. Register now!

netlify - Platform Solutions Integrations Start Building Docs Pricing

Contact Log in Sign up

Connect everything.

Build anything.

Netlify is the modern web development platform for Enterprises to realize the full potential of a scalable, customizable web architecture.

Learn more Request demo

Figure 13.37: Netlify

Netlify requires you to Sign up or log in with any Git cloud account like GitHub, Gitlab and Bitbucket. Login with GitHub, since our code is in GitHub.



Figure 13.38: Netlify login

Since we have a lot of deployed projects, it will show all the sites. If you don't have any, it will show blank.

← → C ⌘ app.netlify.com/teams/nabendu82/overview

Nabendu's team v Search anything... Labs News Support

Team overview Sites Builds Integrations Domains Members Audit log Billing Team settings

+ Try Pro free for 1 month and get background functions, password-protected sites, email support, and more. Not now Start Pro trial

Nabendu's team Starter Current usage period (Aug 15 to Sep 15)

Bandwidth	Build minutes	Concurrent builds	Team members
262 MB/100 GB	0/300	0/1	1

Import environment variables with netlify env:import .env.

Sites Add new site ▾

thewebdev.tech Deploy from GitHub

amazinghampi.com Deploy from GitHub

Builds

restaurant-bangalore Failed Deploy Preview #2...

recipes-homemade Failed Deploy Preview #2...

ccdgatsby Failed Deploy Preview #2...

[Upgrade](#)

<https://app.netlify.com/teams/nabendu82/sites>

Figure 13.39: Netlify dashboard

Now, click on Sites first and then the Add new site button on the top-right. Here, from the drop-down menu, click on Import an existing project.

abendu2/task-npm-hacken | BackApp Containers Dashboard | Sites | abendu2 | Netlify | +

app.netlify.com/teams/nabendu2/sites

Nabendu's team

Search anything...

Labs News Support

Team overview

Sites

Add new site

Import an existing project

Start from a template

Deploy manually

Builds

Integrations

Domains

Members

Audit log

Billing

Team settings

Upgrade

thewebdev.tech

Owned by Nabendu's team
Published on Sep 6, 2022 (a year ago)

Deployed from GitHub

amazinghampi.com

Owned by Nabendu's team
Published on Feb 11, 2020 (4 years ago)

Deployed from GitHub

restaurant-bangalore

Owned by Nabendu's team
Published on Oct 26, 2020 (3 years ago)

Deployed from GitHub

recipes-homemade

Owned by Nabendu's team
Published on May 23, 2020 (3 years ago)

Deployed from GitHub

ccd-gatsby

Owned by Nabendu's team
Published on Apr 21, 2020 (3 years ago)

Deployed from GitHub

twd.codes

Owned by Nabendu's team
Published on May 1 (4 months ago)

Deployed from GitHub

dmz-twd.com

Owned by Nabendu's team

Figure 13.40: Import existing project

In the next screen click on Deploy with GitHub button.

app.netlify.com/start

Nabendu's team

Search anything...

Labs News Support

Team overview

Sites 1. Connect to GitHub provider

Builds 2. Select repository

Integrations 3. Configure site and deploy

Domains

Members Deploy with GitHub Deploy with GitLab

Audit log Deploy with Bitbucket Deploy Azure DevOps

Billing

Team settings

Let's deploy your project.

Get started another way

Try Netlify Drop or Choose a template

Upgrade

Figure 13.41: Deploy with GitHub

Now, in the GitHub access screen, we will select the task-mern-frontend after clicking on the Only select repository.

Installed GitHub App - Netlify - Google Chrome

github.com/settings/installations/750874

Security log

Sponsorship log

Developer settings

Permissions

Netlify is requesting an update to its permissions.

Review request

✓ Read access to code and metadata

✓ Read and write access to checks, commit statuses, and pull requests

Repository access

All repositories

This applies to all current and future repositories owned by the resource owner.
Also includes public repositories (read-only).

Only select repositories

Select at least one repository.
Also includes public repositories (read-only).

Select repositories ▾

task-mern

nabendu82/task-mern-backend
Backend for Task Mern

nabendu82/task-mern-frontend
Frontend for Task Mern

Save

Danger zone

Figure 13.42: Selecting in GitHub

After selecting the repository, click on the Save button.

Installed GitHub App - Netlify - Google Chrome

github.com/settings/installations/750874

Security log

Sponsorship log

Developer settings

Permissions

Netlify is requesting an update to its permissions.

Review request

✓ Read access to code and metadata

✓ Read and write access to checks, commit statuses, and pull requests

Repository access

All repositories

This applies to all current and future repositories owned by the resource owner.
Also includes public repositories (read-only).

Only select repositories

Select at least one repository.
Also includes public repositories (read-only).

Select repositories •

Selected 1 repository.

nabendi02/task-mern-frontend

Save Cancel

Figure 13.43: Access in GitHub

Again, it will ask the permission to install the repository. Click on the Install button next.

Installing Netlify - Google Chrome

github.com/settings/installations/750874/update

Install on your personal account Nabendu

All repositories

This applies to all current and future repositories owned by the resource owner.
Also includes public repositories (read-only).

Only select repositories

Select at least one repository.
Also includes public repositories (read-only).

Select repositories 

Selected 1 repository.

 nabendu82/task-mern-frontend 

with these permissions:

Read access to code and metadata

Read and write access to checks, commit statuses, issues, and pull requests

User permissions

Netlify can also request users' permission to the following resources. These permissions will be requested and authorized on an individual user basis.

Read access to email addresses

 Install  Cancel

Figure 13.44: Install in GitHub

We will now get back to the Netlify screen. Here, we will now see the new repository. Click on the new repository.

← → ⌂ ⌂ app.netlify.com/start/nabendu82/repos

Nabendu's team

Search anything...

Labs News Support

Team overview

Sites 1. Connect to Git provider 2. Select repository 3. Configure site and deploy

Builds

Integrations

Domains

Members

Audit log

Billing

Team settings

Let's deploy your project.

task-mern-frontend
JavaScript 13h ago Public >

task-me

Can't see your repo here? [Configure the Netlify app on GitHub](#)

Upgrade

<https://app.netlify.com/start/repos/nabendu82%2Ftask-mern-frontend>

Figure 13.45: Selecting GitHub in Netlify

In the next screen, we will see the branch for deploying the project. Here, make sure it's the main branch.

← → ⌂ ⌂ 🔍 app.netlify.com/start/repos/nabendu82%2Flask-mern-frontend

Nabendu's team Search anything... Labs News Support

Team overview

Sites 1. Connect to Git provider

BUILDS 2. Select repository

Integrations 3. Configure site and deploy

Domains

Members Review configuration for task-mern-frontend

Audit log Deploy as nabendu82 on Nabendu's team team from main branch using npm run build command and publishing to build

Billing

Team settings Team Nabendu's team

Branch to deploy main Deploying main

Build settings Build command npm run build Published to build

Learn more in the docs /

Upgrade

Base directory

Figure 13.46: Branch of GitHub

On scrolling down on the same screen we will find the Build command and the publish directory. Both of them are right, so click on the Deploy task-mean-frontend button.

nabendu/task-mern-frontend X back4app Containers Dashboard X Create a new site from git Netlify X +

← → C ⌂ app.netlify.com/start/repos/nabendu%2Ftask-mern-frontend

Nabendu's team Search anything... Labs News Support

Team overview The directory where Netlify installs dependencies and runs your build command.

Builds Build command `npm run build`

Examples: jekyll build, gulp build, make all

Integrations Git repository task-mern-frontend

Domains Deploying main

Members Publish directory build command

Audit log Examples: _site, dist, public

Billing Functions directory Published to build

Team settings Add environment variables

Upgrade Deploy task-mern-frontend

Figure 13.47: Built commands

In the next screen, we will see the site deploy in progress.

← → ⌂ ⌂ app.netlify.com/sites/adorable-sunflower-1fc397/overview

N Nabendu's team • adorable-sunflower-1fc397 • Search anything... Labs News Support

Site overview

adorable-sunflower-1fc397

Site deploy in progress

Deploys from GitHub.

Created at 9:57 PM.

Site configuration Domain settings Favorite site

Set up your site

1 Deploying your site Netlify's robots are busy building and deploying your site to our CDN.

2 Set up a custom domain Buy a new domain or set up a domain you already own.

3 Secure your site with HTTPS Your site is secured automatically with a Let's Encrypt certificate.

Upgrade

With Netlify CLI, you can [share your development server](#) over HTTPS.

Figure 13.48: Deploy in progress

Once the site deployment is successful, click on the Open production deploy button.

← → C O app.netlify.com/sites/adorable-sunflower-1fc397/deploy/64dcf8d714b036da396684

Nabendu's team / adorable-sunflower-1fc397 • Search anything... Labs News Support

Site overview Site configuration Deploy Functions Edge Functions Integrations Analytics Forms Split Testing Large Media Upgrade

Published deploy for adorable-sunflower-1fc397 [Permalink](#)

Today at 9:57 PM Production: main@HEAD

[Open production deploy](#) Lock to stop auto publishing Options Play game

Fancy a game? This deploy is done, but hey, you can still play if you want to!

Deploy summary

6 new files uploaded
1 generated page and 5 assets changed.

No redirect rules processed
This deploy did not include any redirect rules. [Learn more about redirects](#)

No header rules processed
This deploy did not include any header rules. [Learn more about headers](#)

All linked resources are secure
Congratulations! No insecure mixed content found in your files.

No functions deployed
This deploy did not include any functions. [Learn more about Netlify Functions](#)

Figure 13.49: Deployment successful

Fixing CORS errors and final deployments

We will now try to log in with the credentials created earlier in the deployed site.



Task Creator

[Login](#) [Register](#)

Login

Login and start creating tasks

username@gmail.com

Submit

Figure 13.50: Login to the deployed site

But we got the error of Request failed with status code 404.



Figure 13.51: Failure to Login

On checking the network tab, we found out it was due to a CORS error. This error comes when our backend and frontend are on different domains and they are not able to communicate with each other, due to strict internet CORS(Cross Origin Request Service) policies.

Figure 13.52: CORS error

To fix this issue, we need to first install the Cors package in the backend folder. We are doing the same by giving the npm i cors command in the integrated terminal.

We also need to import and use it in the server.js file.

```
const Cors = require('cors');

app.use(Cors());
```

File Edit Selection View Go ... server.js - Task-MERN - Visual Studio Code

package.json authService.js taskService.js server.js M X

backend > JS server.js > ...

```
4 const connectDB = require('./connect/database');
5 const Cors = require('cors');
6 const port = process.env.PORT || 5000;
7
8 connectDB();
9 const app = express();
10
11 app.use(express.json());
12 app.use(express.urlencoded({ extended: false }));
13 app.use(Cors());
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)

```
$ npm i cors
```

added 2 packages, and audited 398 packages in 3s

42 packages are looking for funding
run 'npm fund' for details

4 vulnerabilities (3 moderate, 1 critical)

main* 0 △ 0 tabnine starter 🌟 Ln 10, Col 1 Spaces: 4 UTF-8 CRLF () JavaScript ⚡ Go Live 🔍 🔍 🔍

Figure 13.53: Adding CORS in the backend

Since we made changes in the backend code, we need to add, commit, and push the code through the familiar git commands.

git add.

git commit -m “Cors Added”

git push

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

bash + v ⌂ ⌂ ... ^ x

naben@LAPTOP-LG5LF91M MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)

```
$ git add .
```

Warning: LF will be replaced by CRLF in package-lock.json.

The file will have its original line endings in your working directory

Warning: LF will be replaced by CRLF in package.json.

The file will have its original line endings in your working directory

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/backend (main)

```
$ git commit -m "Cors Added"
```

[main a58797f] Cors Added

3 files changed, 38 insertions(+)

naben@LAPTOP-LGSLF9IM MINGW64 ~ /OneDrive/Desktop/Task-MERN/backend (main)

```
$ git push
```

Enumerating objects: 9, done.

Counting objects: 100% (9/9), done.

Delta compression using up to 12 threads

Figure 13.54: git push

We also need to delete the proxy code from the package.json file of the client.

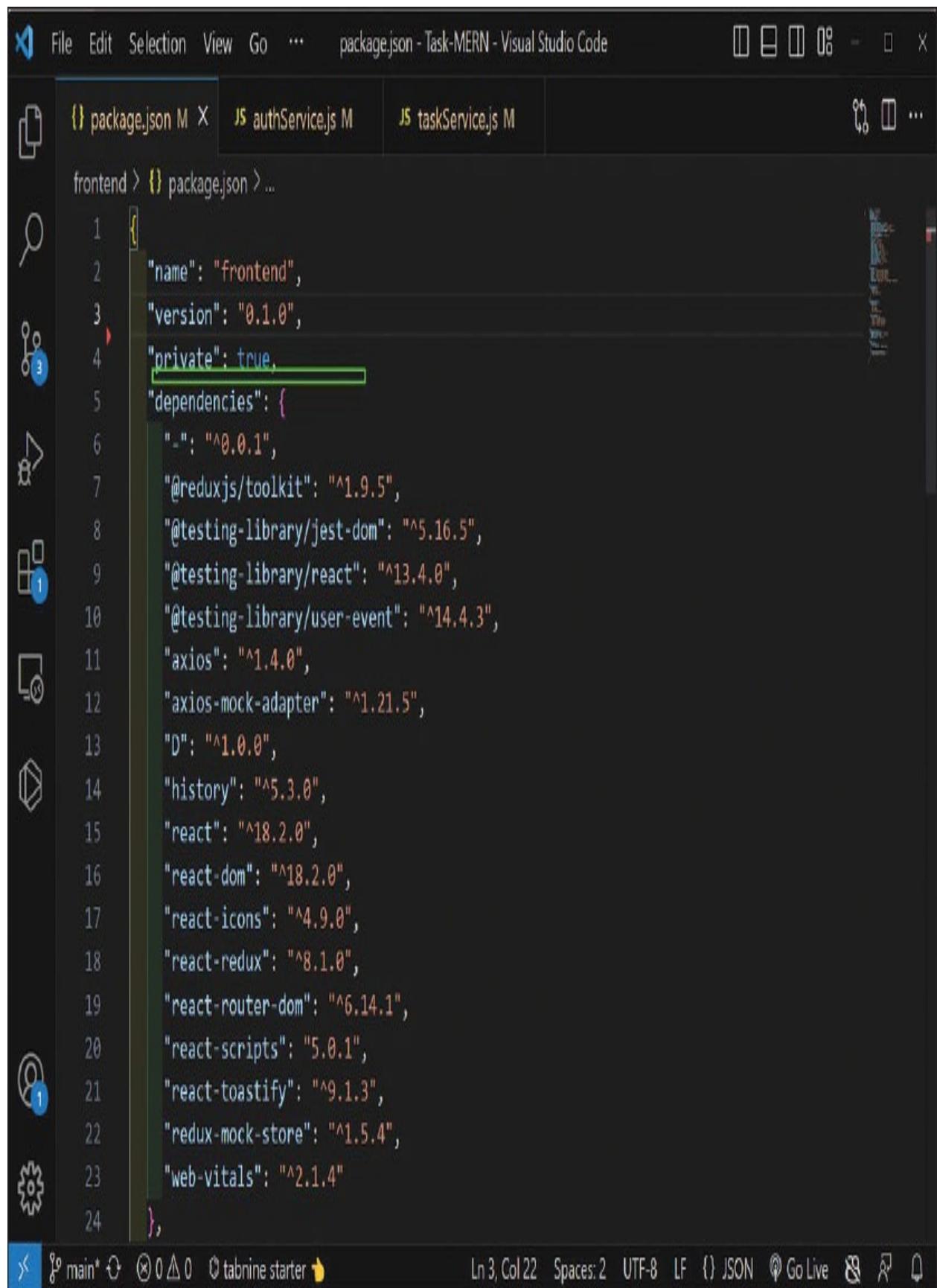


Figure 13.55: Deleting proxy

Next, we will update our authService.js file to contain the full backend URL.

File Edit Selection View Go authService.js - Task-MERN - Visual Studio Code

{ package.json M JS authService.js M X JS taskService.js M ...

frontend > src > features > auth > JS authService.js > ...

```
1 import axios from 'axios'
2 const API_URL = 'https://taskmernbackend-nabendubiswas.b4a.run/api/users/'
3
4 const register = async (userData) => {
5   const response = await axios.post(API_URL, userData)
6   if (response.data) {
7     localStorage.setItem('user', JSON.stringify(response.data))
8   }
9   return response.data
10 }
11
12 const login = async (userData) => {
13   const response = await axios.post(API_URL + 'login', userData)
14   if (response.data) {
15     localStorage.setItem('user', JSON.stringify(response.data))
16   }
17   return response.data
18 }
19
20
21 const logout = () => localStorage.removeItem('user')
22
23 const authService = { register, logout, login }
```

main* tabnine starter ↴ Ln 11, Col 1 Spaces: 2 UTF-8 CRLF () JavaScript Go Live ⌂ ⌂ ⌂

Figure 13.56: Deployed link in authService.js

We will also do the same in the taskService.js file.

File Edit Selection View Go ... taskService.js - Task-MERN - Visual Studio Code

{ package.json M JS authService.js M JS taskService.js M X

frontend > src > features > tasks > JS taskService.js > [o] API_URL

```
1 import axios from 'axios'
2
3 const API_URL = 'https://taskmernbackend-nabendubiswas.b4a.run/api/tasks/'
4
5 const createTask = async (taskData, token) => {
6   const config = {
7     headers: {
8       Authorization: `Bearer ${token}`,
9     }
10   }
11   const response = await axios.post(API_URL, taskData, config)
12   return response.data
13 }
14
15 const getTasks = async token => {
16   const config = {
17     headers: {
18       Authorization: `Bearer ${token}`,
19     }
20   }
21   const response = await axios.get(API_URL, config)
22   return response.data
23 }
```

main* Ø tabnine starter ↴ Ln 3, Col 63 Spaces: 2 UTF-8 CRLF {} JavaScript Go Live ⚡

Figure 13.57: Deployed link in taskService.js

Since we made changes in the frontend code, we need to add, commit, and push the code through the familiar git commands.

git add.

git commit -m “Cors Added”

git push

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$ git add .

warning: LF will be replaced by CRLF in package.json.

The file will have its original line endings in your working directory

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$ git commit -m "Cors added"

[main e66debb] Cors added

3 files changed, 2 insertions(+), 3 deletions(-)

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$ git push

Enumerating objects: 17, done.

Counting objects: 100% (17/17), done.

Delta compression using up to 12 threads

Compressing objects: 100% (9/9), done.

Writing objects: 100% (9/9), 757 bytes | 757.00 KiB/s, done.

Total 9 (delta 6), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (6/6), completed with 6 local objects.

To https://github.com/nabendu82/task-mern-frontend.git

82cbf22..e66debb main -> main

naben@LAPTOP-LGSLF9IM MINGW64 ~/OneDrive/Desktop/Task-MERN/frontend (main)

\$

Figure 13.58: git push

Once we push the code to GitHub, it will automatically deploy to the front end. Now, when we go to our deployed app, everything will work perfectly.



Task Creator

Logout

Welcome Robin

Check Tasks

Enter Task

Add Task

Figure 13.59: Successful deployment

Conclusion

In this final chapter, we have learned to deploy both the frontend and backend code. We deployed the frontend code using the service of Netlify. And deployed the backend code using the service of Back4app.

Points to remember

Uploading code from the local machine to GitHub using the command line

Deploying backend code in Back4app

Deploying frontend code in Netlify

Solving CORS issue which occurs due to frontend and backend on different URLs

[Index](#)

A

API_URL variable 178

Async Thunk

used, for creating task 176-182

Auth middleware

creating 76-78

auth service 129-133

auth slice 129-133

B

Back4app

backend deployment 251-260

backend code repository

creating 249-251

backend deployment

in Back4app 251-260

backend integration
with frontend 260-263

backend testing
setting up 223-225

boilerplate files
deleting 95-108

C

components
creating 111

Header component, creating 112, 113

controllers
creating 28-31

CORS errors and deployments
fixing 269-274

D

dashboard logic

changing 162-164

DELETE API 11

DELETE route 56, 57, 84-89

E

error handling 33-36

Express

about 3, 4

using 20-22

F

frontend

creating, with ReactJS 93-95

used, for integrating backend 260-263

frontend app

checking 220, 221

configuring 220, 221

frontend code repository

creating 244-248

frontend deployment

in Netlify 263-269

frontend testing

setting up 202-206

G

GET API 9

GET route 81-84

getTasks controller

testing 232-237

getTaskSpy variable 211

H

hashed password

used, for registering user 65-68

Header component

creating 112-114

J

Jest

used, for testing task service 206-214

used, for testing task slice 206-214

JSON

using 31-33

JSON Web Token (JWT)

about 71

reference link 71

using 72-74

L

login form 154-157

login functionality

implementing 151-153

Login page

creating 119-122

logout functionality

implementing 143-150

M

MERN project

overview 12

model

creating 50-52

MongoDB 2, 3

MongoDB database

about 38, 39

basic setup 40-44

collection URL, adding 45, 46

connection URL, adding 47

MongoDB, Express, React, and NodeJS (MERN)

about 1, 2

stacks 7

Mongoose

connecting through 48-50

N

Netlify

frontend deployment 263-269

reference link 263

NodeJS 6

NodeJS application

package installation 15-17

project folder structure 13-15

project, running 17, 18

setting up 12, 13

P

pages

creating 111

Login page, creating 119-122

Register page, creating 115-119

POST API 8

Postman

reference link 27

used, for testing Routes 27

POST route 52-54, 81-84

PUT API 10, 11

PUT route 54, 55, 84-89

R

ReactJS 4, 5

used, for creating frontend 93-95

react router

setting up 109-111

React testing library

testing 214-219

Redux toolkit

setting up 124-128

Register page

creating 115-119

registerUser controller

testing 226-232

registration form 134-138

REST API

alternatives 11, 12

REST APIs

about 7, 8

DELETE 11

GET 9

POST 8

PUT 10, 11

RESTful endpoints 7

Routes

about 22

all routes 24-26

creating 52

DELETE route 56, 57

initial routes 23, 24

POST route 52-54

protect function, using 79, 80

PUT route 54, 55

testing, through Postman 27, 28

S

server

used, for fetching task 182-186

setTask controller

testing 232-237

T

task

creating, with Async Thunk 176-182

deleting 195-201

displaying 186-195

fetching, from server 182-186

task form

creating 168-172

task routes

DELETE route 84-89

GET route 81-84

POST route 81-84

protecting 81

PUT route 84-89

task service

testing, with Jest 206-214

task slice

creating 165-168

testing, with Jest 206-214

U

updateTask controller

testing 238-243

user login 69-71

testing 157-160

user model and controller 59-64

user registration

testing 139-142

with hashed password 65-68