

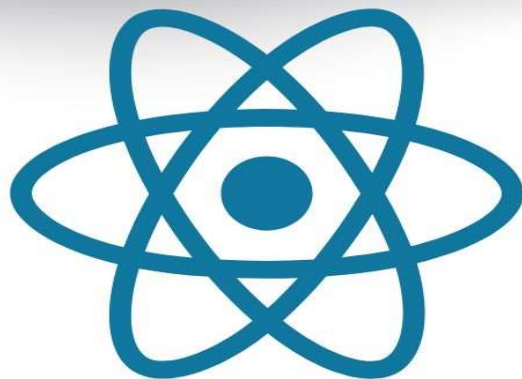
GREG LIM

NEXT.js 13

AND

REACT

CRASH COURSE



Next.js 13 and React Crash Course

Greg Lim

Copyright © 2023 Greg Lim All rights reserved.

COPYRIGHT © 2023 BY GREG LIM ALL RIGHTS RESERVED.
NO PART OF THIS BOOK MAY BE REPRODUCED IN ANY FORM
OR BY ANY ELECTRONIC OR MECHANICAL MEANS
INCLUDING INFORMATION STORAGE AND RETRIEVAL
SYSTEMS, WITHOUT PERMISSION IN WRITING FROM THE
AUTHOR. THE ONLY EXCEPTION IS BY A REVIEWER, WHO
MAY QUOTE SHORT EXCERPTS IN A REVIEW.

FIRST EDITION: AUGUST 2023

Table of Contents [PREFACE](#)

[CHAPTER 1: INTRODUCTION](#)

[CHAPTER 2: SETUP](#)

[CHAPTER 3: FILE STRUCTURE, HOME PAGE AND DAISY UI](#)

[CHAPTER 4: ROUTING SYSTEM AND NESTED ROUTES](#)

[CHAPTER 5: LAYOUTS](#)

[CHAPTER 6: METADATA API](#)

[CHAPTER 7: NAVIGATION BAR](#)

[CHAPTER 8: REACT SERVER COMPONENTS AND CLIENT COMPONENTS](#)

[CHAPTER 9: DATA FETCHING](#)

[CHAPTER 10: RENDERING GITHUB USERS](#)

[CHAPTER 11: DYNAMIC ROUTES AND PARAM PROPS](#)

[CHAPTER 12: 'REPOS' COMPONENT](#)

[CHAPTER 14: CACHING AND REVALIDATING](#)

[CHAPTER 15: API ROUTE HANDLERS](#)

[CHAPTER 16: SERVING 'BOOKS' DATA](#)

[CHAPTER 17: FETCHING 'BOOKS' DATA](#)

[CHAPTER 18: GETTING SEARCH PARAMS](#)

[CHAPTER 20: REFACTOR SERVER TO CLIENT COMPONENT](#)

[CHAPTER 21: SEARCH COMPONENT](#)

[CHAPTER 22: ADDBOOK COMPONENT](#)

[CHAPTER 23: CALLING THE ADDBOOK ENDPOINT](#)

[CHAPTER 24: REFRESHING BOOKS AFTER ADDING](#)

[CHAPTER 25: DELETING A BOOK](#)

CHAPTER 26: SETTING UP PRISMA DATABASE

CHAPTER 27: GETTING BOOKS FROM PRISMA DATABASE

CHAPTER 28: ADDING A BOOK INTO PRISMA

CHAPTER 29: DELETING FROM PRISMA

CHAPTER 30: SEARCHING IN PRISMA

ABOUT THE AUTHOR

PREFACE

About this book

In this book, we take you on a fun, hands-on and pragmatic journey to learning NextJS stack development. You'll start building your first NextJS stack app within minutes. Every chapter is written in a bite-sized manner and straight to the point as I don't want to waste your time (and most certainly mine) on the content you don't need.

In the course of this book, we will cover:

1. Introduction and Setup
2. File Structure, HomePage and Using Daisy UI
3. Routing System and Nested Routes
4. Layouts
5. Metadata API
6. Navigation Bar
7. React Server Components vs Client Components
8. Data Fetching
9. Rendering GitHub Users

10. Dynamic Routes and Params Prop

11 Repos Component

- 12. Loading Page
- 13. Caching and Revalidating 14. API Route Handlers
- 15. Serving Books Data
- 16. Fetching Books Data
- 17. Slight Adjustment
- 18. Getting Search Params
- 19. Adding a New Book
- 20. Refactor Server to Client Component 21. Search Component
- 22. AddBook Component
- 23. Calling the Add Book End Point 24. Refreshing Books After Adding 25. Deleting a Book
- 26. Setting up Prisma Database 27. Getting Books from Prisma 28. Adding a book into Prisma DB
- 29. Deleting from Prisma Database

30. Searching from Prisma Database

The goal of this book is to teach you NextJS stack development in a manageable way without overwhelming you. We focus only on the essentials and cover the material in a hands-on practice manner for you to code along.

Working Through This Book

This book is purposely broken down into short chapters where the development process of each chapter will center on different essential topics. The book takes a practical hands on approach to learning through practice. You learn best when you code along with the examples in the book.

Requirements

React knowledge is recommended. Contact me at support@i-ducate.com for a free Beginners React book if you need to get up to speed with React. No previous NextJS knowledge is required

Getting Book Updates

To receive updated versions of the book, subscribe to our mailing list by sending a mail to support@i-ducate.com. I try to update my books to use the latest version of software, libraries and will update the codes/content in this book. So, do subscribe to my list to receive updated copies!

Code Examples

You can obtain the source code of the completed project by contacting support@i-ducate.com

Online Course

If you are a more visual learner & learn better from absorbing this book's content through an online course, you can access the book's online course by contacting support@i-ducate.com and providing a proof of purchase. The course content is the same as this book. So, if learning through books is your preferred way of learning, skip this. But if you prefer to learn from videos (and you want to hear my voice), contact me.

CHAPTER 1: INTRODUCTION

Hello, and welcome to my Next.js 13 course. I'm delighted that you've decided to pick up this book.

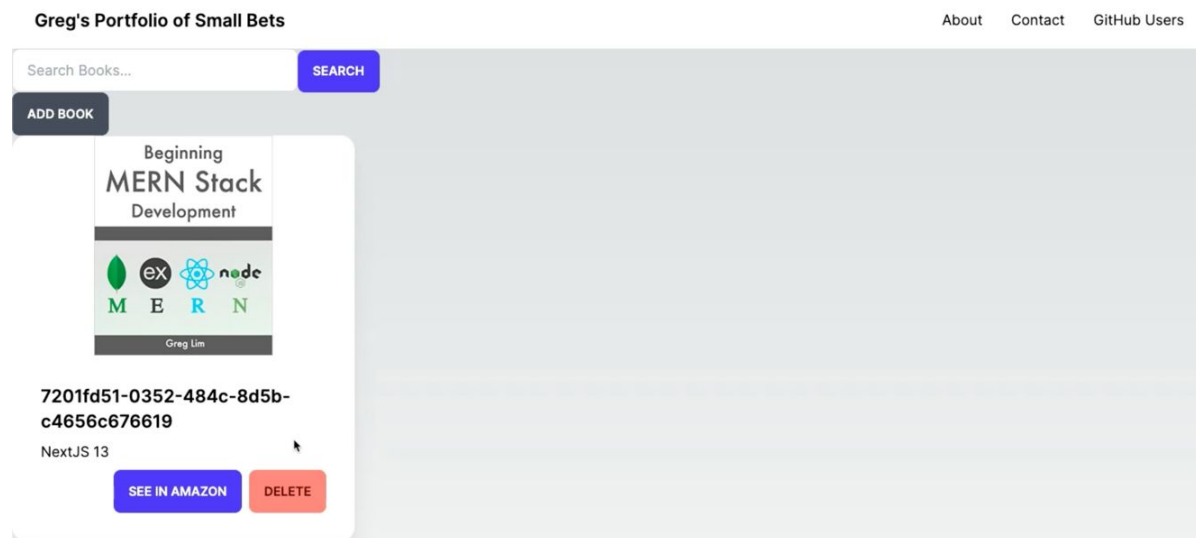
You don't need any previous experience with Next.js, but you do need some basic experience with React. If you're already familiar with React, that's perfect. However, if you're not, don't worry. Please contact me at support@i-ducate.com, and I will provide you with a free beginner's React book to get you started.

In this book, I'll bring you up to speed with all the latest features that Next.js 13 has to offer. We'll explore the new features such as the app directory and how to use it for customized routes. We'll examine React server components, layouts, data fetching from React server components, and how to create API route handlers. These topics will essentially allow us to use React full-stack, from the server to the client.

We'll also delve into how to connect to a backend Prisma database.

Throughout the book, we'll apply these new features in a practical project – a personal portfolio of digital assets, such as books.


In this application, you'll be able to add new books, search for specific books, and delete them:



If I add a new book, it will appear in the list:

Greg's Portfolio of Small Bets

Beginning
MERN Stack
Development



Greg Lim

7201fd51-0352-484c-8d5b-c4656c676619


NextJS 13

Add New Book

If I search for "React," it will show me a book related to React:

Greg's Portfolio of Small Bets

Beginning
React
with Hooks



Greg Lim

3

Beginning React Hooks

Clicking on ‘ See in Amazon ’ will redirect you to the actual Amazon link:



The project also includes navigating to different routes, for example, to an







About Contact GitHub Users

"About" page or a "Contact" page:

Moreover, we'll discuss server components and data fetching from external APIs:

Greg's Portfolio of Small Bets

About Contact GitHub Users

NAME	URL	REPOS
 greg 1658846	VIEW ON GITHUB	GO TO REPOS
 gregkh 14953	VIEW ON GITHUB	GO TO REPOS
 greggman 234804	VIEW ON GITHUB	GO TO REPOS
 gdb 211857	VIEW ON GITHUB	GO TO REPOS
 gregberge 266302	VIEW ON GITHUB	GO TO REPOS
 gregoryyoung 381274	VIEW ON GITHUB	GO TO REPOS

In essence, this course is all about building a small application while learning about and applying the new features of Next.js 13. Along the way, we'll explore some other interesting aspects. I look forward to our journey together in mastering Next.js 13.

CHAPTER 2: SETUP

Now, we will use the command 'Create Next App' to initiate our Next.js project. If you're familiar with React, this is very similar to the 'Create React App' command.

First, in your Terminal, navigate to the folder where you wish to create the project. For me, I've created a dedicated NextJS folder. Before we proceed any further, we need to ensure that Node.js is installed. To check, type 'node --version'. As you can see, I have version 16.17:

```
MacBook-Air-2:NextJS user$ node --version
v16.17.0
MacBook-Air-2:NextJS user$
```

Once that's confirmed, we run:

```
'npx create-next-app@latest'
```

which will install the latest version of Next.js.

After running the command, you'll be asked a series of questions, such as your project's name. It provides a default name - 'my-app'. Since I already have a project named 'my-app', I'll call mine 'my-app-2'. Feel free to name your project as you please:

```
MacBook-Air-2:NextJS user$ npx create-next-app@latest
✓ What is your project named? ... my-app2
```

In this setup process, we won't be using TypeScript or ESLint. However, we will be using Tailwind CSS (see below).

```
MacBook-Air-2:NextJS user$ npx create-next-app@latest
✓ What is your project named? ... my-app2
✓ Would you like to use TypeScript with this project? ... No / Yes
✓ Would you like to use ESLint with this project? ... No / Yes
✓ Would you like to use Tailwind CSS with this project? ... No / Yes
✓ Would you like to use `src/` directory with this project? ... No / Yes
✓ Use App Router (recommended)? ... No / Yes
✓ Would you like to customize the default import alias? ... No / Yes
Creating a new Next.js app in /Users/user/Documents/NextJS/my-app2.
```

Using npm.

```
Initializing project with template: app-tw
```

We won't be utilizing the source directory to keep things simple. We will be using the app router. We don't need to customize the default import alias at this point: The setup will then proceed to install the dependencies:

Installing dependencies:

- react
- react-dom
- next
- tailwindcss
- postcss
- autoprefixer

added 109 packages, and audited 110 packages in 13s

21 packages are looking for funding
run `npm fund` for details

found 0 vulnerabilities

Success! Created my-app2 at /Users/user/Documents/NextJS/my-app2

Once the setup process is complete, it will have created a new project named 'my-app2'.

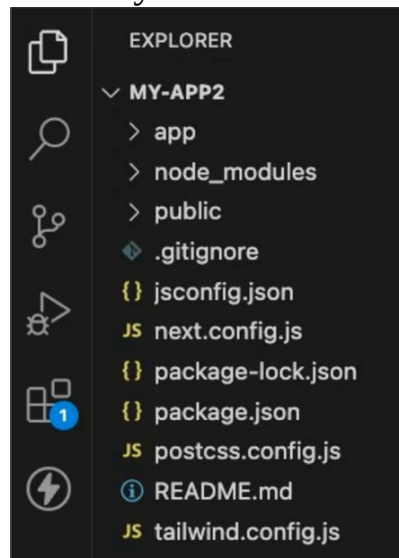
Navigate into this directory with:

```
cd my-app2
```

and open it with VS Code.

You can use your preferred IDE, but I highly recommend VS Code as we'll be utilizing a useful extension called 'Thunder Client' for testing our API endpoints later on.

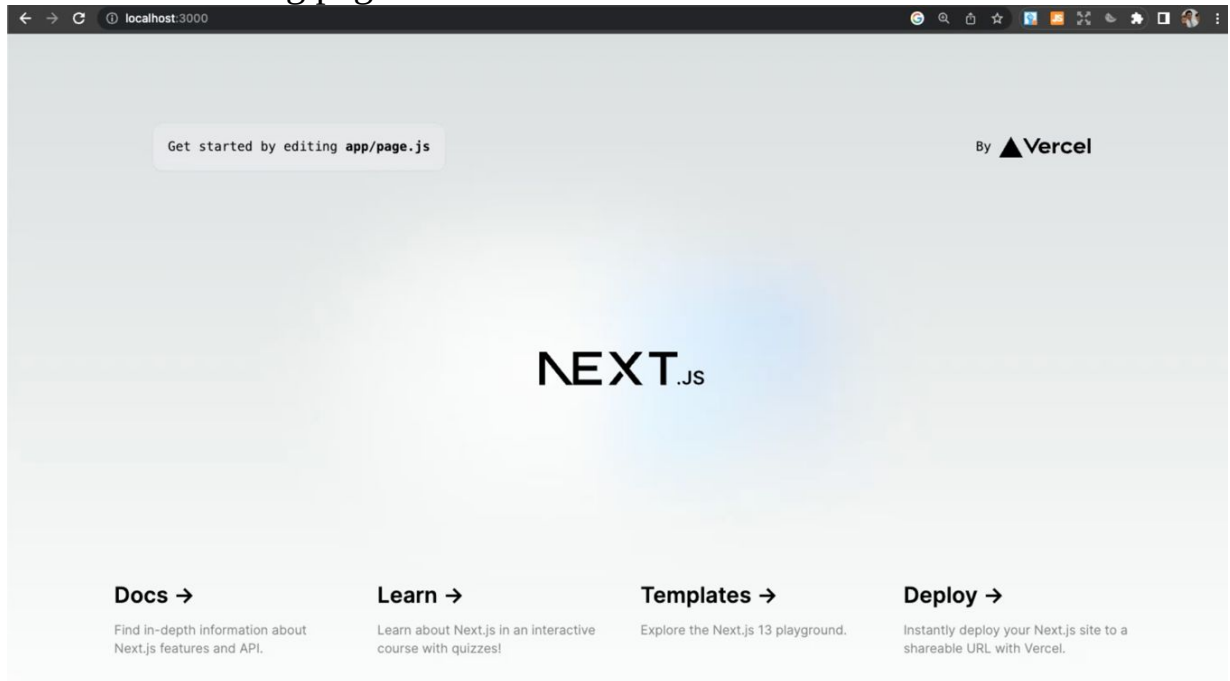
Once VS Code is open, we'll open the project folder. You'll see the complete directory structure on the left.



Our next step is to run the development server from the terminal. To do this, execute the command 'npm run dev'.

```
MacBook-Air-2:NextJS user$ cd my-app2
MacBook-Air-2:my-app2 user$ npm run dev
```

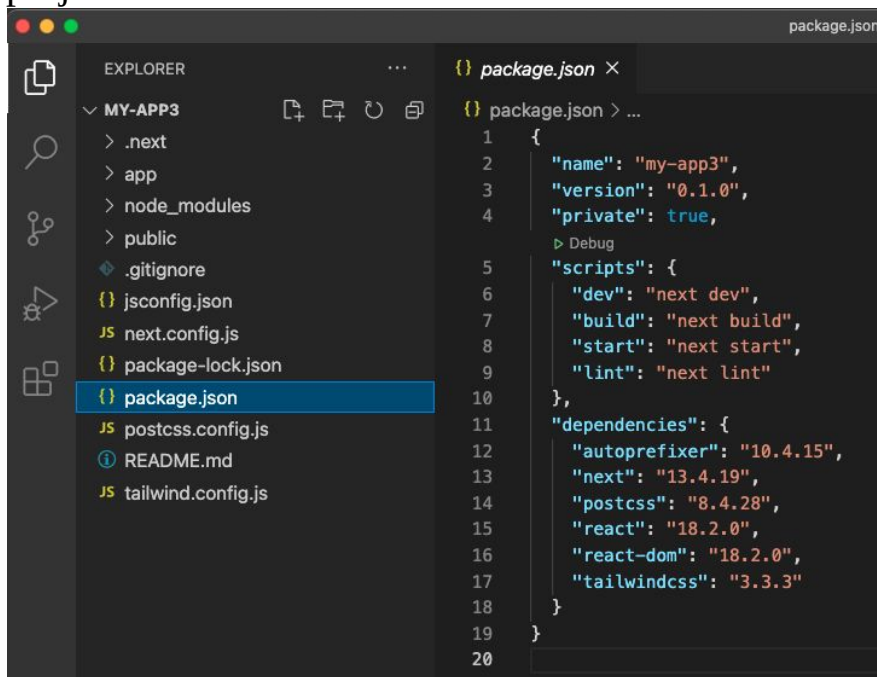
Upon navigating to 'localhost:3000' in your web browser, you should see the default landing page.



CHAPTER 3: FILE STRUCTURE, HOMEPAGE AND DAISY UI

Let's explore the file structure.

In our package.json, under 'dependencies', we have Next.js, React, React-DOM, and Tailwind CSS, which we chose to install when we created the project.



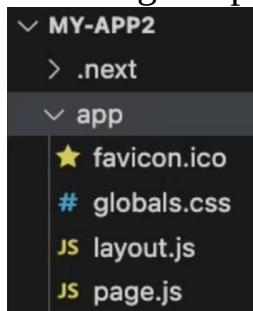
The 'scripts' section includes our 'dev' script, which we just executed. If we wish to build our project for production, we can run 'npm run build', and to test the production build, we can run 'npm start'.

```
"scripts": {  
  "dev": "next dev",  
  "build": "next build",  
  "start": "next start",  
  "lint": "next lint"  
},
```

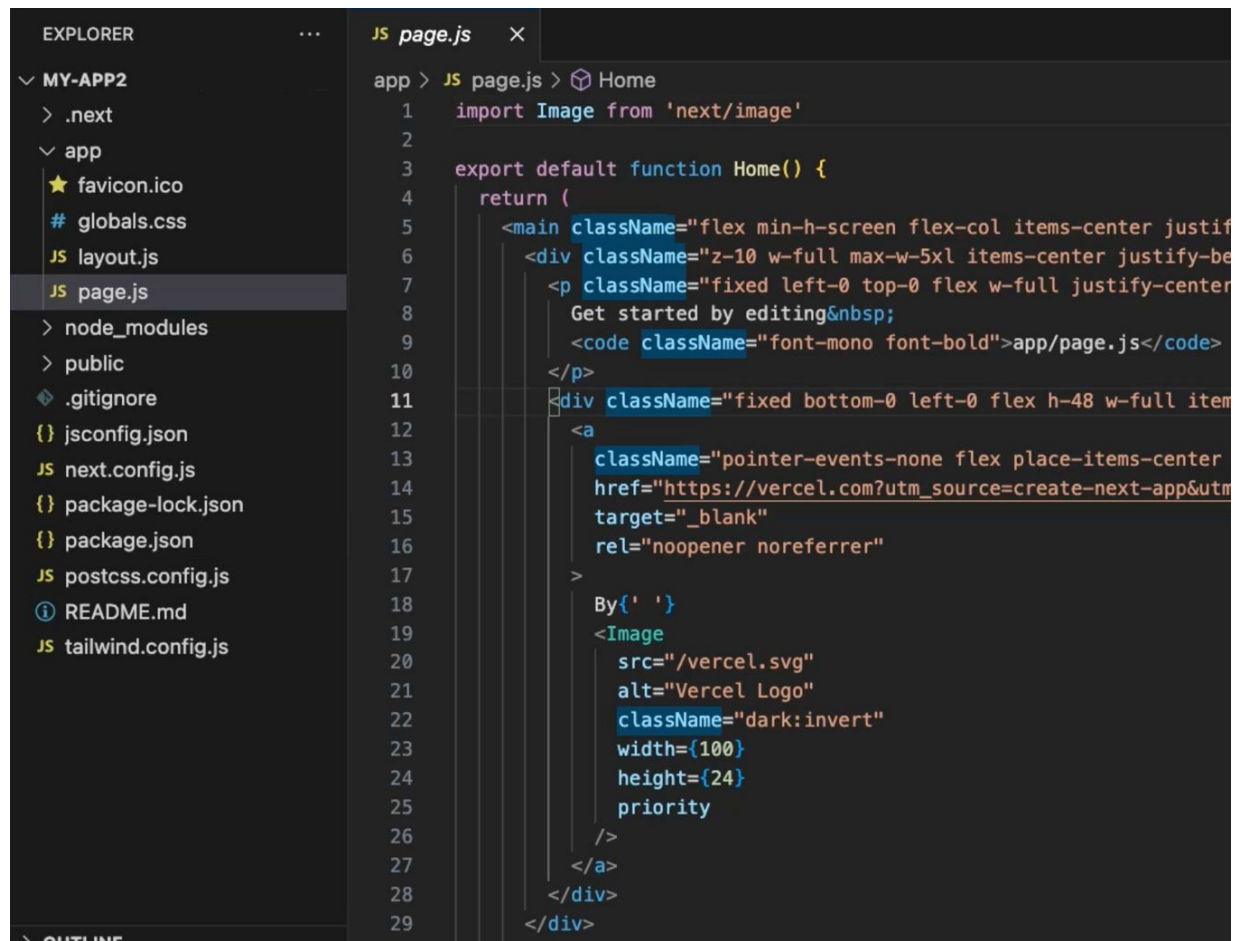
Going back to dependencies, we also have Autoprefixer, a tool that automatically adds vendor prefixes to CSS code to ensure compatibility across all browsers, and PostCSS, which transforms our CSS code.

```
"dependencies": {  
  "autoprefixer": "10.4.15", "next": "13.4.19",  
  "postcss": "8.4.28",  
  "react": "18.2.0",  
  "react-dom": "18.2.0",  
  "tailwindcss": "3.3.3"  
}
```

Next, we'll examine the 'app' folder where everything we create goes, including our pages, components, and API route handlers.



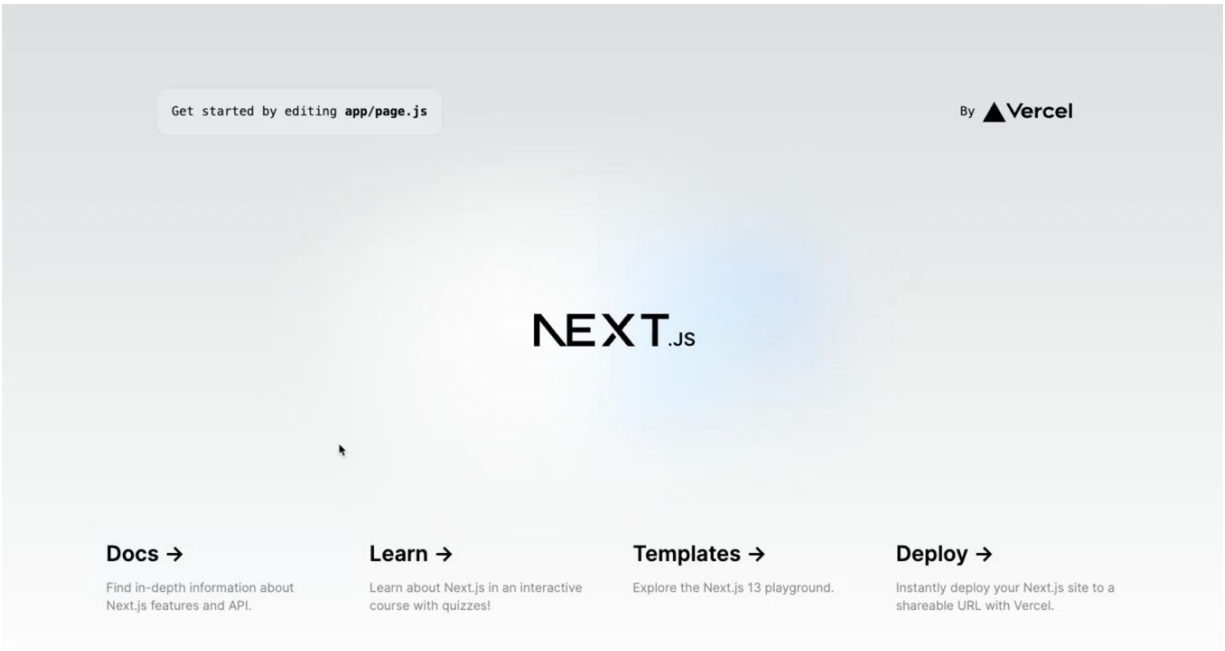
'page.js' houses our homepage.



The image shows a code editor interface with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'MY-APP2' with a file tree containing: .next, app (with favicon.ico, globals.css, layout.js, page.js, node_modules, public, .gitignore, jsconfig.json, next.config.js, package-lock.json, package.json, postcss.config.js, README.md, and tailwind.config.js), and an OUTLINE section. The code editor shows the content of 'page.js' with the following code:

```
1 import Image from 'next/image'
2
3 export default function Home() {
4   return (
5     <main className="flex min-h-screen flex-col items-center justify-between">
6       <div className="z-10 w-full max-w-5xl items-center justify-between">
7         <p className="fixed left-0 top-0 flex w-full justify-center">
8           Get started by editing app/page.js
9         </p>
10       </div>
11       <div className="fixed bottom-0 left-0 flex h-48 w-full items-center">
12         <a
13           className="pointer-events-none flex place-items-center"
14           href="https://vercel.com?utm_source=create-next-app&utm_medium=default-template&utm_campaign=vercel-docs"
15           target="_blank"
16           rel="noopener noreferrer"
17         >
18           By{' '}
19           <Image
20             src="/vercel.svg"
21             alt="Vercel Logo"
22             className="dark:invert"
23             width={100}
24             height={24}
25             priority
26           />
27         </a>
28       </div>
29     </div>
```

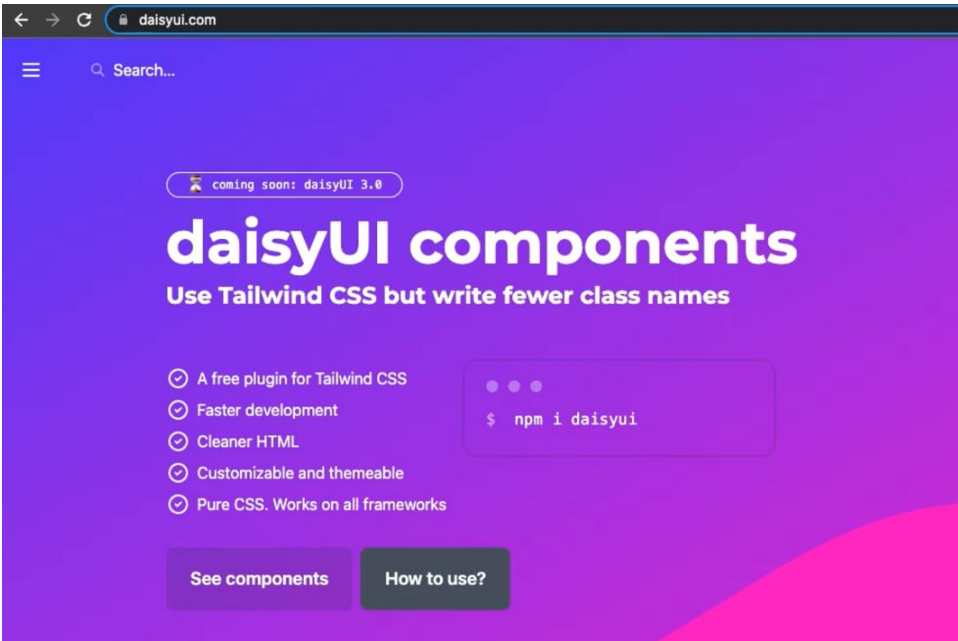
If you navigate to 'localhost:3000', the dashboard you see originates from 'page.js'.



To simplify, we'll remove everything in 'page.js' and instead create a functional 'HomePage' component by filling in the below codes:

```
const HomePage = () => {  
  return (  
    <div>  
      <h1>Home Page</h1> </div>  
    )  
  };  
  export default HomePage;
```

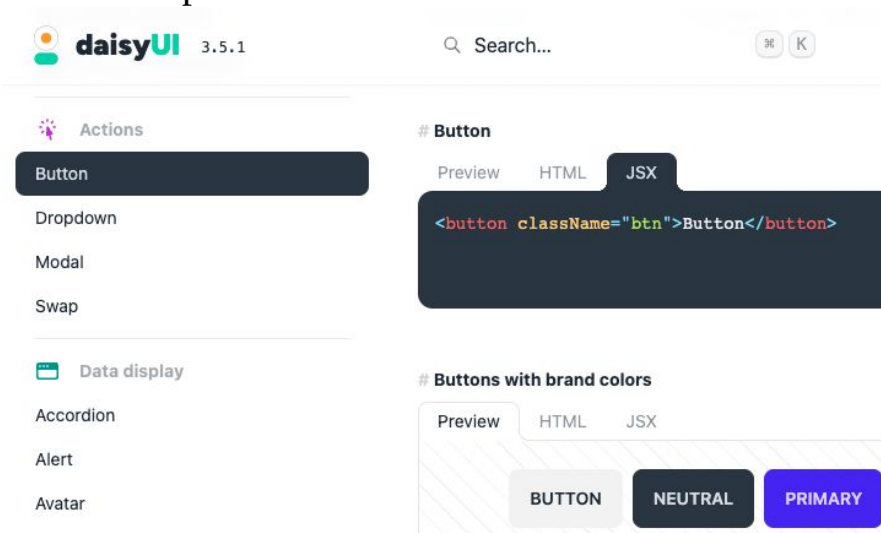
Now, regarding styling, we use Tailwind CSS in combination with Daisy UI, which enhances our usage of Tailwind CSS. Daisy UI provides numerous pre-built templates and components, making it easier and faster to build our interface.



In a new Terminal, install Daisy UI and add it to our project by running:
`npm i daisyui`

```
MacBook-Air-2:my-app2 user$ npm i daisyui
( [REDACTED] ) " idealTree:my-app2: sill idealTree buildDeps
```

Now, to ensure Daisy UI has been installed and is usable, let's import a button component.

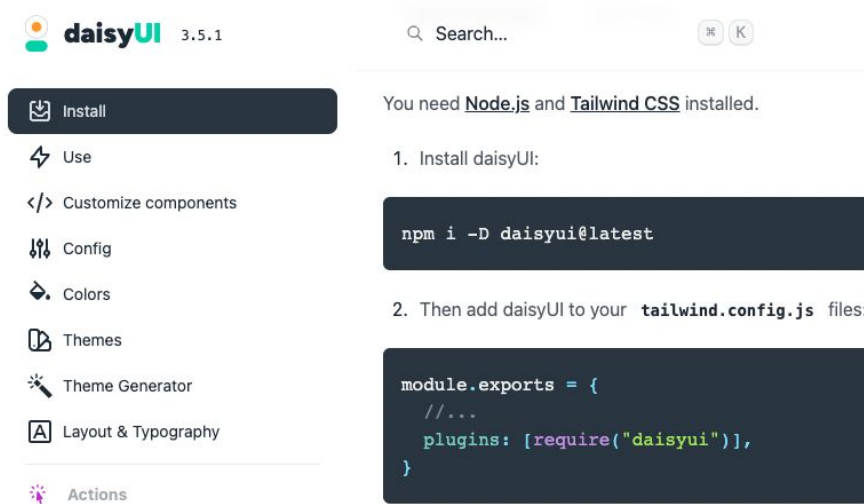


Copy the JSX markup from DaisyUI and add it to our 'HomePage' component. Apply the 'primary' style to make it more noticeable. It should

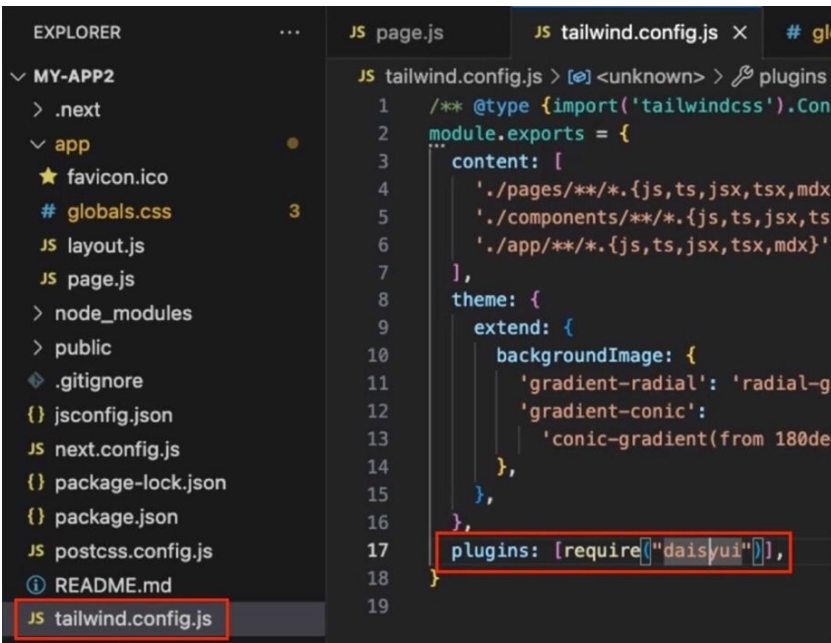
look like the code in **bold**:

```
const HomePage = () => {  
  return (  
    <div>  
      <h1>Home Page</h1> <button className="btn btn-primary">Button</button> </div>  
    )  
  };  
  export default HomePage;
```

Before we proceed, we also need to add Daisy UI to our Tailwind CSS configuration file.

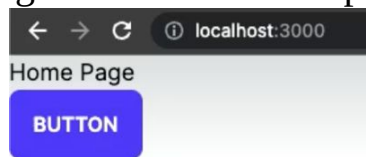


In tailwind.config.js, under 'plugins', make sure to include 'require('daisyui')'.



```
1  /** @type {import('tailwindcss').Config} */
2  module.exports = {
3    content: [
4      './pages/**/*.js,ts,jsx,tsx,mdx',
5      './components/**/*.js,ts,jsx,tsx,mdx',
6      './app/**/*.js,ts,jsx,tsx,mdx'
7    ],
8    theme: {
9      extend: {
10        backgroundImage: {
11          'gradient-radial': 'radial-gradient(circle, #f06292 1px, transparent 1px)',
12          'gradient-conic': 'conic-gradient(from 180deg at 50% 50%, #f06292 0deg, #f06292 360deg)',
13        },
14      },
15    },
16    plugins: [require('daisyui')],
17  }
```

With that done, if we go back to our homepage, we should see the button



displaying correctly:

And because we are using DaisyUI, we can remove the existing CSS styles in /app/globals.css and have just the tailwind essentials:

```
@tailwind base;
@tailwind components;
@tailwind utilities;

:root {
  ...
}

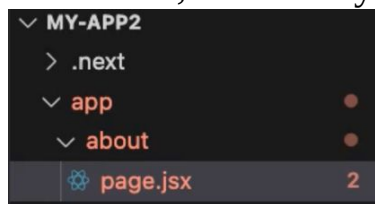
@media (prefers-color-scheme: dark) {
  ...
}

body {
  ...
}
```

CHAPTER 4: ROUTING SYSTEM AND NESTED ROUTES

Now, let's discuss routing using Next.js. Let's say we want to create an 'About' page. How do we do this?

Firstly, under the 'app' directory, create a new folder named 'about'. Inside this folder, create a file called 'page.jsx'. You could use either '.js' or '.jsx' extensions; it's entirely up to you.



Inside this file, we'll create a simple component named 'AboutPage' with the below codes:

```
const AboutPage = () => {  
  return <div>About Page</div>;  
};  
export default AboutPage;
```

To demonstrate the functioning of this new page and its route, let's go back to our 'HomePage' component in ' /app/page.js '. We'll import the 'Link' component from 'next/link', and use this to create a couple of navigation links. Add the codes in **bold**:

/app/pages.js

```
import Link from "next/link";  
const HomePage = () => {  
  return (  
    <div>  
      <h1>Home Page</h1> <button className="btn btn-primary">Button</button> <ul>  
        <li><Link href="/">Home</Link></li> <li><Link href="/about">About</Link></li>  
      </ul>  
    </div>  
  )  
};
```

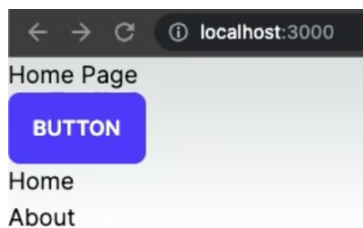
```
export default HomePage;
```

For the 'About' page, we'll use '/about', and for the 'Home' page, just '/'.

Note that with Next.js 13, we no longer need the ' <a> ' tag in the 'Link' component: ...

```
<li><Link href="/"><a>Home</a></Link></li> ...
```

After saving these changes, if we navigate to our homepage, we'll see the 'About' and 'Home' links.



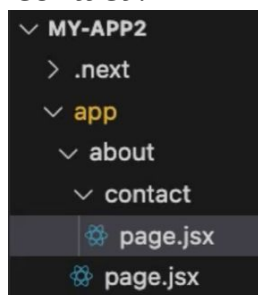
Clicking on 'About' will take us to the 'About' page, while 'Home' keeps us on the homepage.

Next.js also supports nested routes. Let's say we want a 'Contact' page nested within the 'About' page. Add:

/app/page.jsx

```
<li><Link href="/">Home</Link></li> <li><Link href="/about">About</Link></li> <li><Link href="/about/contact">Contact</Link></li>
```

To achieve this, within the 'about' folder, we create a nested folder called 'contact'.

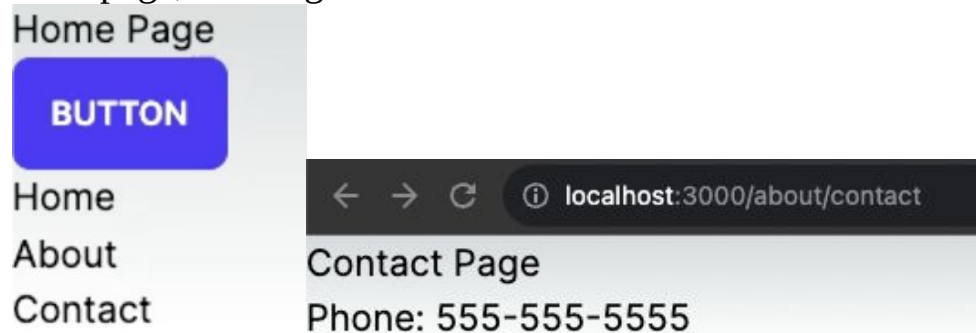


Every custom route we create corresponds to a folder name. Inside the 'contact' folder, we'll create a 'page.jsx' file and a 'ContactPage' component.

/app/about/contact/page.jsx

```
const ContactPage = () => {  
  return (  
    <div> <h1>Contact Page</h1> <p>Phone: 555-555-5555</p> </div> );  
  }  
  export default ContactPage;
```

With this nested route, we now have '/about/contact'. Back on our homepage, clicking on 'Contact' takes us to the nested 'Contact' page.



If you require further nested routes, simply create more nested folders. For example, if you wanted a '/about/contact/greg' route, you'd have an 'about' folder, with a nested 'contact' folder, which then contains a nested 'greg' folder.

CHAPTER 5: LAYOUTS

Next, we'll explore layouts in Next.js. If you look at '/app/layout.js', it wraps everything in our app.

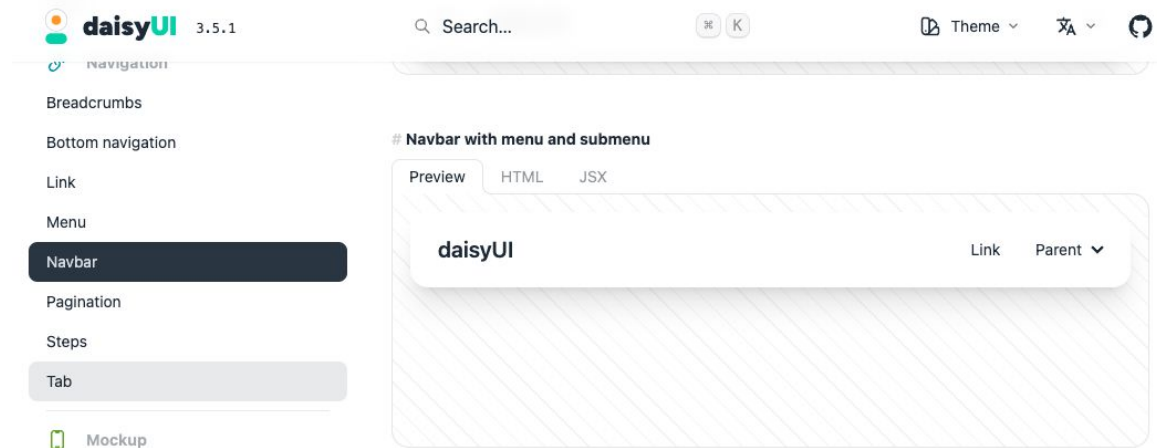
```
...
const inter = Inter({ subsets: ['latin'] })

export const metadata = {
  ...
}

export default function RootLayout({ children }) {
  return (
    <html lang="en"> <body className={inter.className}>{children}</body> </html> )
  )
}
```

This file includes a component called 'RootLayout' which receives 'children' as a prop. These 'children' could be the routes or pages we create. This file also contains the HTML and body tags. Therefore, if you want something to appear on every page, such as a header or footer, it can be placed here.

For instance, let's say we want to include a navigation bar. We could find a suitable one on DaisyUI,



Copy the JSX and paste it above {children} in the main layout.

```
export default function RootLayout({ children }) {
```

```

return (
  <html lang="en"> <body className={inter.className}> Add here...
    {children}
  </body>
</html>
)
}

```

Now, this navigation bar will appear on top of all pages and routes in our



If you need a specific layout for a certain page, such as a 'Login' or 'Signup' page, where you don't want the general header to appear, you can create a unique 'layout.js' file within their respective folders.

We'll revisit layouts and headers later on, but for now, this should provide a basic understanding of how layouts work in Next.js.

CHAPTER 6: METADATA API

In the 'app/layout.js' file, we have a metadata object which contains key information like the page title and description.

```
import './globals.css'
import { Inter } from 'next/font/google'

const inter = Inter({ subsets: ['latin'] })

export const metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

export default function RootLayout({ children }) {
  ...
```

You can also add additional meta tags or keywords as needed. For instance, the title could be "Greg's Portfolio of Small Bets", and the description might be "Tech Courses and Books" (see below).

```
...
const inter = Inter({ subsets: ['latin'] })

export const metadata = {
  title: 'Greg\'s Portfolio of Small Bets',
  description: 'Tech Courses and Books',
  keywords: 'passive income, small bets, tech courses, tech books, tech tutorials'
}
...
```

You could also specify keywords such as "Passive Income".

When you save these changes and refresh your website, you'll notice that the page title has updated. This process can significantly aid in search engine optimization (SEO), helping your website to rank higher on Google.

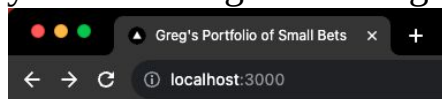
However, you might want to customize the metadata on each of your different pages to optimize each one for SEO. To do this, copy the metadata object from 'layout.js' and paste it into each individual folder's page.jsx.

For instance, in the '/app/about/page.jsx', you could add "About" to the title, and customize the description and keywords accordingly:

```
export const metadata = {  
  title: 'About Greg's Portfolio of Small Bets',  
  description: '...',  
  keywords: '...' };
```

```
const AboutPage = () => {  
  return <div>About Page</div>;  
};  
export default AboutPage;
```

Once saved, you'll notice the page title changes depending on the page you're viewing - it's "Greg's Portfolio of Small Bets" on the homepage,



daisyUI

Home Page

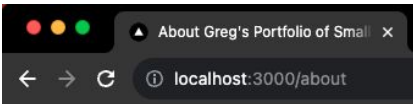
BUTTON

Home

About

Contact

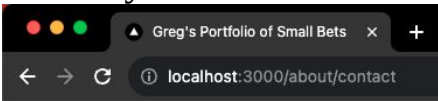
but changes to "About ... " when you navigate to the 'About' page.



daisyUI

About Page

And these changes don't trickle down to nested pages. For instance, if you navigate to the 'Contact' page, the title won't change because the metadata was only set for the 'About' page.



daisyUI

Contact Page

Phone: 555-555-5555

CHAPTER 7: NAVIGATION BAR

Let's now turn our attention back to our navigation bar (header). We used a copy-and-paste template from DaisyUI, so we'll need to customize it further to suit our needs.

In `/app/layout.js`, the first step is to import 'Link' from 'next/link'.

```
Import './globals.css'
import { Inter } from 'next/font/google'
import Link from 'next/link';
...
```

We don't really need the submenu Parent item from our original template, so we can remove that.

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <div className="navbar bg-base-100">
          <div className="flex-1">
            <a className="btn btn-ghost normal-case">
          </div>
          <div className="flex-none">
            <ul className="menu menu-horizontal px-1">
              <li><a>Link</a></li>
              <li>
                <details>
                  <summary>
                    Parent
                  </summary>
                  <ul className="p-2 bg-base-100">
                    <li><a>Link 1</a></li>
                    <li><a>Link 2</a></li>
                  </ul>
                </details>
              </li>
            </ul>
          </div>
        </div>
      </body>
    </html>
  );
}
```

Let's customize these further with the changes in **bold**:

```
...
...
export default function RootLayout({ children }) {
  return (
```

```

<html lang="en">
<body className={inter.className}>
<div className="navbar bg-base-100"> <div className="flex-1">
  <Link href="/" className="btn btn-ghost normal-case text-xl"> Greg's Portfolio of Small
Bets </Link>
</div>
...

```

Instead of 'Daisy', we want a link that directs users to the homepage. To achieve this, we can replace the 'a' tag with 'Link', and set the 'href' attribute to '/'. Then, instead of 'Daisy', we can insert the name of the website, eg. "Greg's Portfolio of Small Bets".

We add the Links to the 'About' and 'Contact' pages respectively. Make the changes in **bold**: ...

```

<div className="flex-1">
  <Link href="/" className="btn btn-ghost normal-case text-xl"> Greg's Portfolio of Small Bets
</Link>
</div>
<div className="flex-none"> <ul className="menu menu-horizontal px-1"> <li><Link
href="/about">About</Link></li> <li><Link href="/about/contact">Contact</Link></li>
<li><Link href="/githubusers">GitHub Users</Link></li> </ul>
</div>
...

```

We also add a third link for searching GitHub users. We label this 'GitHub Users' and set the route to '/githubusers'. We will implement this later.

While it's possible to refactor this code further by putting it into its own 'Header' component, for the sake of simplicity, we won't do that right now.

Once these changes have been saved, we can see that the navigation bar is working fine on the homepage:

The links to the 'About' and 'Contact' pages are working as expected, although the 'GitHub Users' link won't work just yet, as we haven't created that page.

CHAPTER 8: REACT SERVER COMPONENTS AND CLIENT COMPONENTS

Let's now take a moment to discuss React server components and React client components.

So far, all the pages we've created, such as the 'About', 'Contact', and 'Home' pages, are all React server components by default.

/app/about/contact/page.jsx

```
const ContactPage = () => {  
  return (  
    <div>  
      <h1>Contact Page</h1> <p>Phone: 555-555-5555</p> </div>  
    );  
  }  
}  
  
export default ContactPage;
```

In Next.js, unless specifically stated otherwise, all components are server components, meaning they are rendered on the server.

Server components come with a few notable benefits.

First, they offer faster initial page loads, because there's no need to wait for JavaScript to load. This also results in a smaller client-side JavaScript bundle size, since many components aren't included on the client-side.

Second, React server components are SEO-friendly, which is essential, given that we can define our metadata in these components (as shown earlier).

Third, server components provide access to resources usually inaccessible to the client. For instance, you can access backend resources like databases

directly from the server component and safeguard sensitive data like access tokens and API keys on the server.

Overall, using server components often enhances the developer experience and simplifies work processes.

However, server components do have their drawbacks, especially regarding interactivity. They don't support event listeners like *onClick* or *onChange*, component states, or *useEffect*. If your application requires these kinds of interactive features, then server components aren't the right choice.

For example, if you try to import *useState* into a server component. Eg.:
/app/about/page.jsx

```
import { useState } from 'react';
export const metadata = {
  ...
}

const AboutPage = () => {
  return <div>About Page</div>;
}
export default AboutPage;
```

You'll encounter an error message stating that *useState* only works in a client component and the current component is not marked with 'useClient'.

Failed to compile

```
./app/about/page.jsx
ReactServerComponentsError:

You're importing a component that needs useState. It only works in a Client Component but
none of its parents are marked with "use client", so they're Server Components by default.

    ,-[/Users/user/Documents/NextJS/my-app2/app/about/page.jsx:1:1]
  1 | import { useState } from 'react';
    |               ^^^^^^^
  2 |
  3 | export const metadata = {
  4 |   title: 'About Greg\'s Portfolio of Small Bets',
    |   -----
    |
    |

Maybe one of these should be marked as a client entry with "use client":
./app/about/page.jsx
```

This error occurred during the build process and can only be dismissed by fixing the error.

If you want to transform a server component into a client component, you must mark it with 'useClient'.

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

export const metadata = {
  ...
}
...
export default AboutPage;
```

However, doing so would mean you can't include metadata in the component, which triggers another error message.

Failed to compile

```
./app/about/page.jsx
ReactServerComponentsError:

You are attempting to export "metadata" from a component marked with "use client", which is
disallowed. Either remove the export, or the "use client" directive. Read more:
https://nextjs.org/docs/getting-started/react-essentials#the-use-client-directive

- [Users/user/Documents/NextJS/my-app2/app/about/page.jsx:1:1]
1 | 'use client';
2 | import { useState } from 'react';
3 |
4 | export const metadata = {
  |      ^^^^^^^
5 |   title: 'About Greg\'s Portfolio of Small Bets',
6 |   description: 'Tech Courses and Books',
7 |   keywords: 'passive income, small bets, tech courses, tech books, tech tutorials'
  |   ^^^^^
  |   _____

File path:
./app/about/page.jsx
```

To make it work as a client component, it will be something like:

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

const AboutPage = () => {
  return <div>About Page</div>;
}
export default AboutPage;
```

The above 'About' page is currently functioning as a typical React client component. Although it could *useState* or *useEffect*, we're opting not to, since the page doesn't require any interactivity. Instead, it primarily serves to present information and needs to be SEO-optimized. For these reasons, we'll continue treating it as a React server component.

So ensure you revert the changes back, ie.:

```
export const metadata = {
  title: 'About Greg\'s Portfolio of Small Bets', description: 'Tech Courses and Books', keywords:
'passive income, small bets, tech courses, tech books, tech tutorials'
}

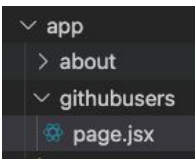
const AboutPage = () => {
  return <div>About Page</div>;
}
```

```
export default AboutPage;
```

CHAPTER 9: DATA FETCHING

Let's discuss data fetching in a React server component. Usually, in a React client component, we would utilize *useEffect* for fetching data. However, in this case, we need to fetch GitHub user data in a server component.

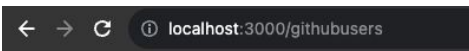
Our goal here is to retrieve a list of GitHub users when the 'GitHub Users' link is clicked. To do this, we'll need to create a route for 'GitHub Users'. Under 'app', we'll create a new folder named 'githubusers' and within that, a file named 'page.jsx'.



We then create a 'GitHubUsersPage' component with the following:

```
const GitHubUsersPage = () => {  
  return(  
    <div>  
      <h1>GitHub Users Page</h1> </div>  
    )  
  }  
}  
export default GitHubUsersPage;
```

Once saved, you should be directed to the appropriate page upon clicking the 'GitHub Users' link.

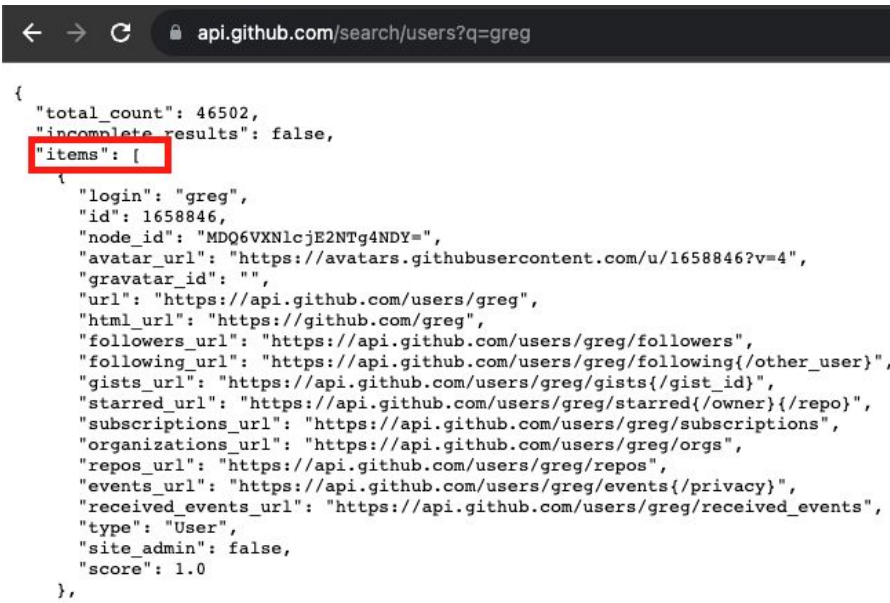


Greg's Portfolio of Small Bets

GitHub Users Page

The next step involves fetching the GitHub users. This can be achieved by accessing the GitHub API via 'api.github.com/search/users' and appending a username to this URL eg.: *http://api.github.com/search/users?q=greg*

The API returns a list of results, including an 'items' array which houses the array of GitHub users.



```
{
  "total_count": 46502,
  "incomplete_results": false,
  "items": [
    {
      "login": "greg",
      "id": 1658846,
      "node_id": "MDQ6VXNlcjE2NTg4NDY=",
      "avatar_url": "https://avatars.githubusercontent.com/u/1658846?v=4",
      "gravatar_id": "",
      "url": "https://api.github.com/users/greg",
      "html_url": "https://github.com/greg",
      "followers_url": "https://api.github.com/users/greg/followers",
      "following_url": "https://api.github.com/users/greg/following{/other_user}",
      "gists_url": "https://api.github.com/users/greg/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/greg/starred{/owner}/{repo}",
      "subscriptions_url": "https://api.github.com/users/greg/subscriptions",
      "organizations_url": "https://api.github.com/users/greg/orgs",
      "repos_url": "https://api.github.com/users/greg/repos",
      "events_url": "https://api.github.com/users/greg/events{/privacy}",
      "received_events_url": "https://api.github.com/users/greg/received_events",
      "type": "User",
      "site_admin": false,
      "score": 1.0
    }
  ]
}
```

To fetch this data, we will create a function called 'fetchGitHubUsers'. Add the code in **bold**:
/app/githubusers/page.jsx

```
async function fetchGitHubUsers () {
  const res = await fetch("https://api.github.com/search/users?q=greg"); const json = await
res.json () ; return json;
}
```

```
const GitHubUsersPage = async() => {
  const users = await fetchGitHubUsers () ; console.log (users) ;
  return(
    <div>
      <h1>GitHub Users Page</h1> </div>
    )
}
export default GitHubUsersPage;
```

Code Explanation

```

async function fetchGitHubUsers ( ){
  const res = await fetch("https://api.github.com/search/users?q=greg"); const json = await
res.json() ; return json;
}

```

Given we'll be using 'await', the function must be labeled as 'async'. The function fetches from the GitHub Users link. For the sake of this example, we'll hardcode 'Greg' into the search.

```

const GitHubUsersPage = async() => {
  const users = await fetchGitHubUsers () ; console.log (users) ;
  return(
    <div>
      <h1>GitHub Users Page</h1> </div>
    )
}
export default GitHubUsersPage;

```

We then call *fetchGitHubUsers* in the *GitHubUsersPage* component and mark it as asynchronous ('async') because we 'await' the completion of *fetchGitHubUsers*. To verify we're receiving the correct data, we'll log the output to the console.

Running our App

After saving the changes and navigating to 'GitHub Users', in the Terminal server logs, you'll see that all required data is retrieved. However, we're currently receiving the entire JSON response when we're only interested in the 'items' array. So, in the 'fetchGitHubUsers' function, we should ensure only 'items' are returned. Add the following:

/app/githubusers/page.jsx

```

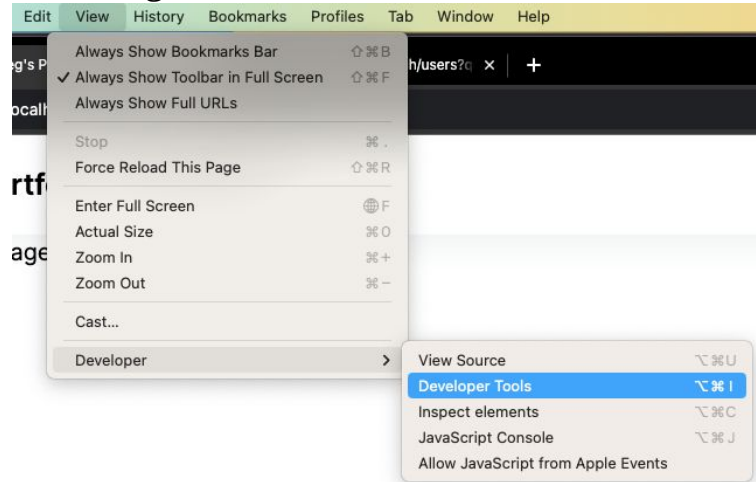
async function fetchGitHubUsers ( ){
  const res = await fetch("https://api.github.com/search/users?q=greg"); const json = await
res.json() ;
  return json.items; }

```



```
const GitHubUsersPage = async() => {  
  ...  
}  
export default GitHubUsersPage;
```

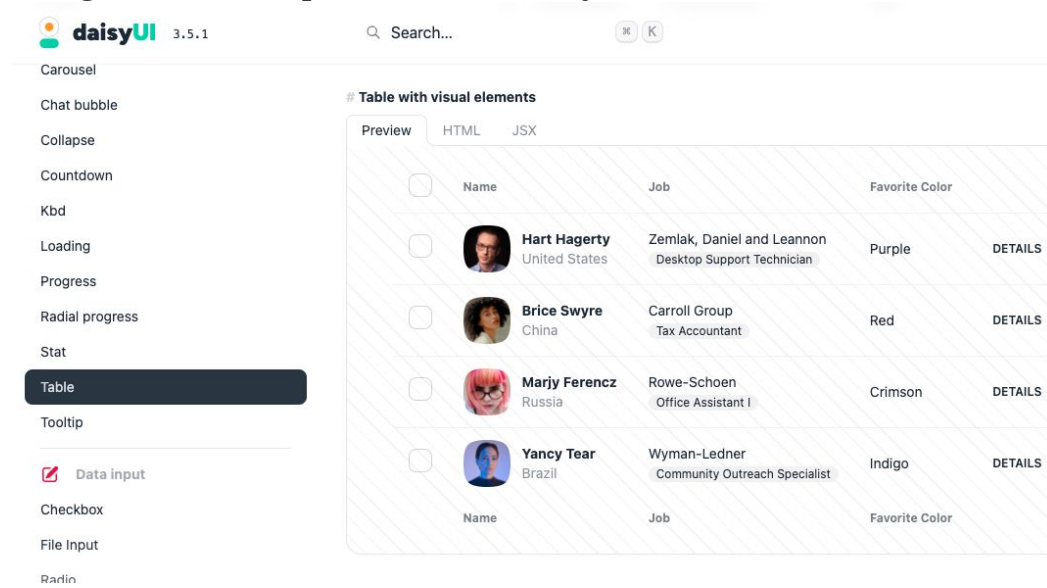
It's important to note that since this is a server component, nothing will be visible in the browser's developer tools logs as all the operations are occurring on the server-side.



Consequently, we need to check the server logs in the terminal. However, if 'useClient' is specified at the top of our component, the console log will be visible in the browser.

CHAPTER 10: RENDERING GITHUB USERS

Next, let's examine how to render and output the fetched users onto our page, instead of merely logging the data to the console. For this, I'll be using a Table component from DaisyUI.



The Table component will present GitHub users' profile pictures, their names, GitHub URLs, and their repositories.

Copy the Table component markup from DaisyUI,

Table with visual elements

Preview HTML JSX

```
<div className="overflow-x-auto">
  <table className="table">
    { /* head */ }
    <thead>
      <tr>
        <th>
          <label>
            <input type="checkbox" className="checkbox" />
          </label>
        </th>
        <th>Name</th>
        <th>Job</th>
        <th>Favorite Color</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      { /* row 1 */ }
      <tr>
        <th>
```

then paste it into the GitHubUsersPage Component (/app/githubusers/page.jsx) in *return*. Remove unnecessary dummy rows and footings from the markup to simplify the overall layout.

Here 's the simplified markup (refer to the source codes if you prefer to avoid typing it all out – contact support@i-ducate.com):

import Link from "next/link";

...

```
const GitHubUsersPage = async() => {
  const users = await fetchGitHubUsers();





  return (
    <div className="overflow-x-auto"> <table className="table"> <thead>
      <tr> <th>Name</th> <th>URL</th> <th>Repos</th> <th></th> </tr> </thead>
      <tbody>
        {users.map((user) => (
          <tr key={user.id}> <td> <div className="flex items-center space-x-3"> <div
            className="avatar"> <div className="mask mask-squircle w-12 h-12"> <img src=
            {user.avatar_url} /> </div> </div> <div> <div className="font-bold"> {user.login}
            </div>
            <div className="text-sm opacity-50"> {user.id}
          </div>
            <div> </div> </td> <td> <Link href={user.html_url} className="btn btn-link">
            View on GitHub
          </Link>
            </td> <th> Go to Repos </th> </tr> )}}
        </tbody> </table>
      </div>

  );
```

```
}
export default GitHubUsersPage;
```

The table will look something like:

localhost:3000/githubusers

Greg's Portfolio of Small Bets			About	Contact	GitHub Users
Name	URL	Repos			
 greg 1658846	VIEW ON GITHUB	Go to Repos			
 gregkh 14953	VIEW ON GITHUB	Go to Repos			
 greggman 234804	VIEW ON GITHUB	Go to Repos			
 gdb 211857	VIEW ON GITHUB	Go to Repos			

Code Explanation

```
...
<tbody>
  {users.map((user) => (
    <tr key={user.id}> ...
  </tr>
  ))}
</tbody>
...

```

We iterate through our 'users' array and display each user. This is done with the 'map' function in JavaScript. We'll map each user to a row in our table. Additionally, we specify the 'key' property to ensure each element in the list has a unique ID. This is a good practice in React. For this, use the 'id' property from the user object.

```
...
    {users.map((user) => (
      <tr key={user.id}> <td>
        <div className="flex items-center spacex-3"> <div className="avatar"> <div
className="mask mask-squircle w-12 h-12"> <img src={user.avatar_url} /> </div> </div>
```

The first column displays the user's profile picture, fetched from the 'avatar_url' property in the API response:

```
{
  "total_count": 46502,
  "incomplete_results": false,
  "items": [
    {
      "login": "greg",
      "id": 1658846,
      "node_id": "MDQ6VXNlcjE2NTg4NDY=",
      "avatar_url": "https://avatars.githubusercontent.com/u/1658846?v=4",
      "gravatar_id": "",
      "url": "https://api.github.com/users/greg",

```

```
...
    <div>
      <div className="font-bold"> {user.login}
    </div>
    <div className="text-sm opacity-50"> {user.id}
  </div>
  </div> </div>
...

```

The 'login' property will give us the user's name.

```
{
  "total_count": 46502,
  "incomplete_results": false,
  "items": [
    {
      "login": "greg", "id": 1658846, ...

```

User ID is obtained from the 'id' property.

Next, we need the URL to link to each user's GitHub profile. In the API response, the 'html_url' property holds this information.

To make this a clickable link to redirect the user to the GitHub profile when clicked, we import 'Link' at the top:

```
import Link from 'next/link';
```

```
async function fetchGitHubUsers(){
  ...
  ...

```

and replace the text with 'View on GitHub'. Thus, we have:

```
    <td> <Link href={user.html_url} className="btn btn-link">
View on GitHub
  </Link>
    </td>

```

To ensure a more aesthetic appeal, we apply the class ‘ btn-link ’ to style the link.

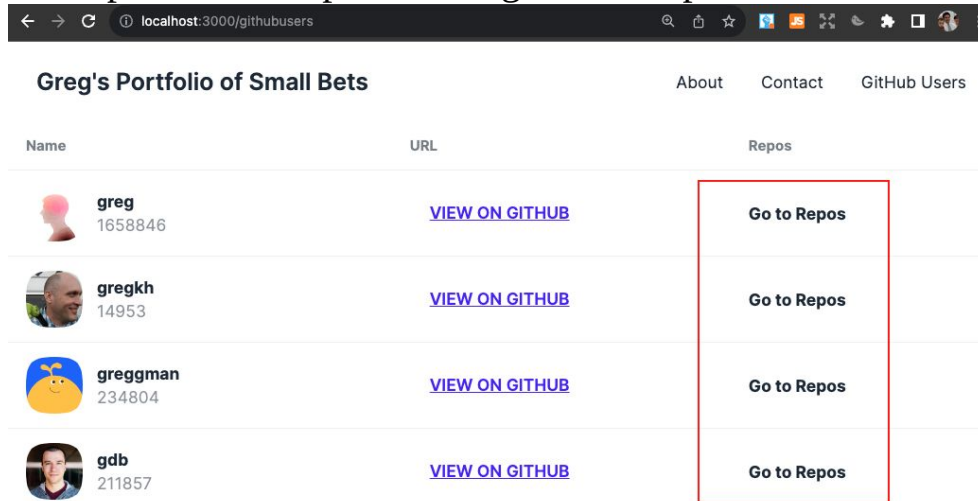
Lastly, we add a column for 'Repos', intending to implement a link to each user's repositories later. For now, we just display ‘ Go to Repos ’ .

```
<th> Go to Repos </th>
```

If you get lost at any point in this chapter, refer to the source codes or the online lecture (contact support@i-ducate.com).

CHAPTER 11: DYNAMIC ROUTES AND PARAM PROPS

The next objective is to navigate to a page in our app listing all repositories for a specific user upon clicking 'Go to Repos'.



To achieve this, we will utilize a 'Link' component that references a route structured as '/github_users/<username>'. Add the code:

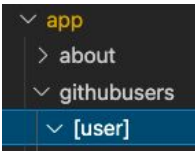
/app/githubusers/page.jsx ...

```
<tbody>
{users.map((user) => (
  <tr key={user.id}> ...
    <th>
      <Link href={`/${githubusers}/${user.login}`} className="btn btn-link"> Go to Repos
    </Link> </th>
  </tr>
))}
</tbody>
...

```

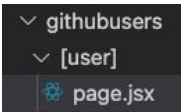
This way, upon saving and clicking on the link, it redirects you to a URL like 'localhost:3000/github_users/<username>'. Here, 'username' is dynamically obtained from the user's GitHub login.

Since the route is '/github_users/<username> ', we need to create a new folder named '[user]' under the 'githubusers' directory.



The square brackets denote a dynamic route parameter. The naming of this dynamic parameter (in this case, 'user') can be 'id', 'name', or anything else depending on your application.

In the '[user]' folder, create another file named 'page.jsx'.



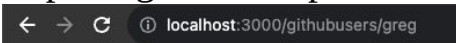
It's important to note that in this setup, the actual name for a route's page is always 'page.jsx'. We don't use specific names like 'about.js', 'repos.js', 'user.js' – the routing is based on folder names, and the corresponding file is always 'page.jsx'.

In 'page.jsx', we will define a component called 'UserReposPage' with the following:

```
const UserReposPage = () => {  
  return <div>Users Repo Page</div> }  
export default UserReposPage;
```

Initially, we won't provide any arguments to this component. Later we will fetch the user's GitHub username from the URL parameters. As a placeholder, we label the page as 'User's Repo Page'.

If we save the file and click on 'Go to Repos', we navigate to the 'User's Repo Page' of the specific user. Eg: <http://localhost:3000/githubusers/greg>



Greg's Portfolio of Small Bets

Users Repo Page

To extract the GitHub username (e.g., 'greg') from the URL, we will use a 'params' prop. Add the following:

```
const UserReposPage = ({params: {user}}) => {  
  return <div>Users Repo Page</div> };  
export default UserReposPage;
```

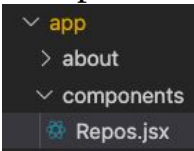
From this 'params' object, we can extract the username.

Next, we want to fetch the repositories for the particular user.

CHAPTER 12: ‘ REPOS ’ COMPONENT

Our next task is to create a component that retrieves and lists a user's repositories. Instead of including this functionality directly in a page, it's more suitable to delegate it to a separate component.

In our 'app' directory, we'll create a new 'components' folder. Within 'components', we create a new file 'Repos.jsx'.



This 'Repos' component will handle fetching and listing all repositories for a specific user.

We'll start by defining the 'Repos' component. Fill in the following code in Repos.jsx:

```
async function fetchRepos(user) {
  const res = await fetch(`https://api.github.com/users/${user}/repos`); const json = await res.json();
  return json;
}

const Repos = async ({user}) => {
  const repos = await fetchRepos(user);
  console.log(repos);
  return (
    <div>
      <h1>Repos</h1>
    </div>
  )
}
export default Repos;
```

Code Explanation

```
const Repos = async ({user}) => {
  const repos = await fetchRepos(user);
```

We take the user's GitHub username as a prop to retrieve their repositories. We will fetch the repositories using an asynchronous 'fetchRepos' function. We pass the user's GitHub username as an argument to the 'fetchRepos' function.

```
async function fetchRepos(user) {  
  const res = await fetch(`https://api.github.com/users/${user}/repos`); const json = await res.json();  
  return json;  
}
```

The 'fetchRepos' function will use the GitHub API to fetch repositories for a specific user. When you replace the 'username' in the API's URL with a valid GitHub username and open it in a web browser, you receive a list of repositories for that user.

Eg. <https://api.github.com/users/greg/repos>

After fetching the repositories, the function will return the repositories as JSON.

Testing our App




To verify that our 'Repos' component is correctly fetching the repositories, we'll log the repositories in the console. To test this, we import the 'Repos' component in `/app/githubusers/[user]/page.jsx` and pass the username as a prop. Add the following:

```
import Repos from "../../components/Repos";  
const UserReposPage = ({params: {user}}) => {  
  return (  
    <div>  
      <Repos user={user} />  
    </div>  
  )  
};  
export default UserReposPage;
```

Now once we navigate to the 'Repos' page,

Greg's Portfolio of Small Bets

[About](#)
[Contact](#)
[GitHub Users](#)

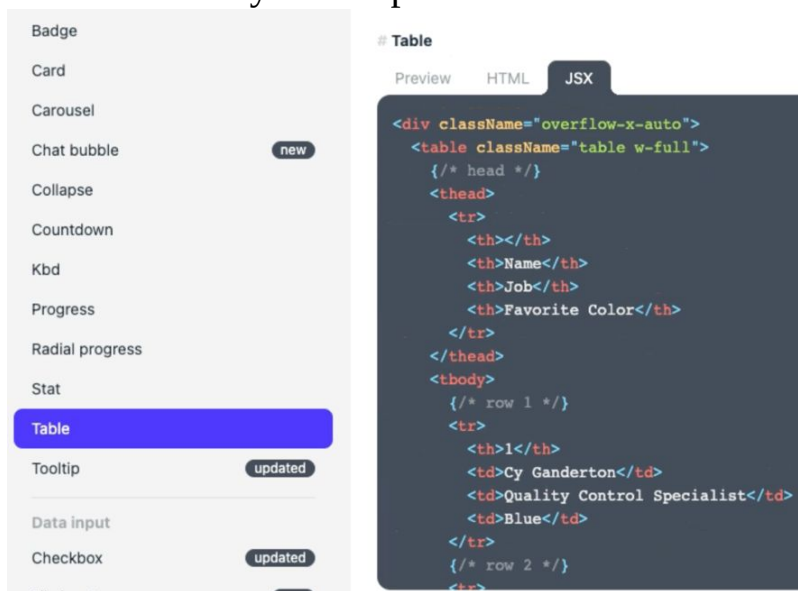
Name	URL	Repos
 <div> <div>greg</div> <div>1658846</div> </div>	VIEW ON GITHUB	GO TO REPOS
 <div> <div>gregkh</div> <div>14953</div> </div>	VIEW ON GITHUB	GO TO REPOS
 <div> <div>greggman</div> <div>234804</div> </div>	VIEW ON GITHUB	GO TO REPOS

we should see the user's repositories logged in the console.

```
- GET /githubusers/greg 200 in 1067ms
  GET https://api.github.com/users/greg/repos 200 in 900ms (cache: MISS)
```

After confirming the 'Repos' component is functioning correctly, we can remove the console log and start displaying the repositories.

We'll use a table DaisyUI component for this. We will use the first table



there:

Copy the markup into Repos.jsx. It should something like:

```
const Repos = async ({user}) => {
  const repos = await fetchRepos(user);
```

```

return (
  <div>
    <h1>{user}'s Repos</h1> <div className="overflow-x-auto"> <table className="table w-
full"> { /* head */}
    <thead>
      <tr>
        <th>Repo Name</th> <th>Description</th> </tr>
      </thead>
      <tbody>
        {repos.map((repo) => (
          <tr> <td>{repo.name}</td> <td>{repo.description}</td> </tr> ))}
        </tbody>
      </table>
    </div>
  </div>
)
}
export default Repos;

```

The table's header have columns for the repository's name and description. In the table's body, we map over the repositories and display each repository's name and description.

The repository's name and description are properties available in the JSON object that the GitHub API returns.

```

{
  "id": 169835898,
  "node_id": "MDEwOlJlcG9zaXRvcnR5cXNj4MzU4OTg=",
  "name": "CodableInterception",
  "full_name": "greg/CodableInterception",
  "private": false,
  "owner": {
    "login": "greg",
    "id": 1658846,
    "node_id": "MDQ6VXNlcjE2NTg4NDY=",
    "avatar_url": "https://avatars.githubusercontent.com/u/1658846?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/greg",
    "html_url": "https://github.com/greg",
    "followers_url": "https://api.github.com/users/greg/followers",
    "following_url": "https://api.github.com/users/greg/following{/other_user}",
    "gists_url": "https://api.github.com/users/greg/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/greg/starred{/owner}/{repo}",
    "subscriptions_url": "https://api.github.com/users/greg/subscriptions",
    "organizations_url": "https://api.github.com/users/greg/orgs",
    "repos_url": "https://api.github.com/users/greg/repos",
    "events_url": "https://api.github.com/users/greg/events{/privacy}",
    "received_events_url": "https://api.github.com/users/greg/received_events",
    "type": "User",
    "site_admin": false
  },
  "html_url": "https://github.com/greg/CodableInterception",
  "description": "A generalised library for intercepting & customising the encoding/decoding process for Codable types",
  "fork": false,
  "url": "https://api.github.com/repos/greg/CodableInterception",
  "forks_url": "https://api.github.com/repos/greg/CodableInterception/forks",
  "keys_url": "https://api.github.com/repos/greg/CodableInterception/keys{/key_id}",
  "collaborators_url": "https://api.github.com/repos/greg/CodableInterception/collaborators{/collaborator}",
  "teams_url": "https://api.github.com/repos/greg/CodableInterception/teams",
  "hooks_url": "https://api.github.com/repos/greg/CodableInterception/hooks",
  "issue_events_url": "https://api.github.com/repos/greg/CodableInterception/issues/events{/number}",

```

After adding the repository's name and description to the table and refreshing the page, we can see the table populated with the user's repositories.

Greg's Portfolio of Small Bets		About	Contact	GitHub Users
greg's Repos				
Repo Name	Description			
CodableInterception	A generalised library for intercepting & customising the encoding/decoding process for Codable types			
corebluetooth-rainbow	Demo app that uses CoreBluetooth to sync up screen colours of connected devices			
CyclicCoding	Encodes & decodes Codable object graphs containing duplicates and cycles			
dotfiles				
fb-mac-messenger	⚡ Mac app wrapping Facebook's Messenger for desktop			
goofy	OS X client for Facebook Messenger			
ICInputAccessory	A customized token text field used in the iCook app.			
Inconsolata-LGC	Inconsolata LGC extension			

With that, we successfully created a component that fetches and lists a user's GitHub repositories.

Chapter 13: Loading Page

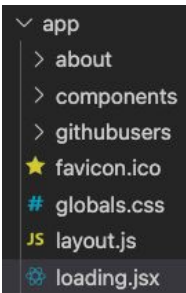
Let's explore how to implement a loading page, which is a helpful feature during data fetching. When the app is fetching data, we can display a

Greg's Portfolio of Small Bets

LOADING

spinner:

To accomplish this, we'll start by navigating to our *app* folder and creating a file named 'loading.jsx' (or 'loading.js').



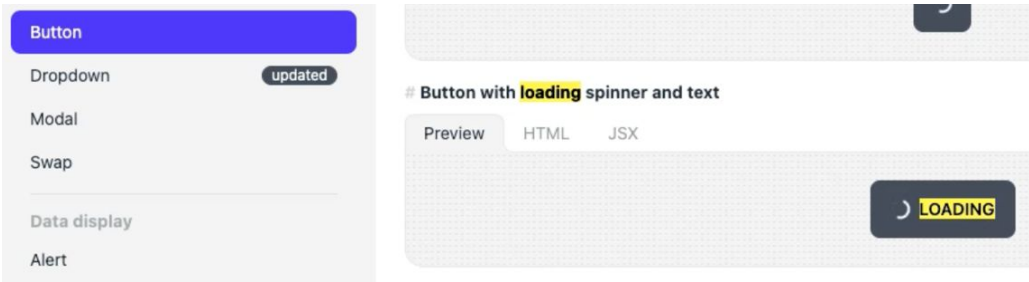
In this file, we'll define a 'LoadingPage' component that displays "Loading".

Fill it with the following: `const LoadingPage = () => {`

```
  return (
    <div>
      Loading
    </div>
  )
}
```

`export default LoadingPage;`

For a more aesthetically pleasing loading indicator, we can opt for a loading spinner with text. Let 's find a suitable loading indicator under the "button" category of the DaisyUI library.



We'll copy this loading spinner markup and insert it into our 'LoadingPage' component.



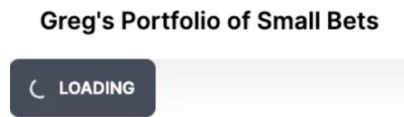
```
const LoadingPage = () => {
  return (
    <div>
      <button className="btn loading">loading</button> </div>
    )
  }
}
export default LoadingPage;
```

Remember that while the component's name can be anything you choose, the filename must be 'loading.js' or 'loading.jsx'.

With the loading page set up, you may notice that the spinner flashes quickly on the screen if your data loads very fast. To make the loading spinner more visible, we can deliberately introduce a delay in our data fetching. For example, in our 'GitHubUsers' component, we can add a delay of a few seconds before fetching the data. Add the following in **bold**:
import Link from 'next/link';

```
async function fetchGitHubUsers ( ){
  const res = await fetch("https://api.github.com/search/users?q=greg"); await new
Promise((resolve) => setTimeout(resolve, 5000)); const json = await res.json(); return json.items;
}
```


This promise resolves only after a set delay (e.g., 5 seconds). With this delay in place, when we navigate to our GitHub users page, we see the loading spinner appear before the data loads.



This loading indicator is implemented with the 'loading.jsx' file we created. We didn't have to manually manage the loading state with 'useState' or other such hooks. As long as we have a server-side component and a loading page (loading.jsx), the app will automatically display the loader while fetching data.

CHAPTER 14: CACHING AND REVALIDATING

In our current code, we use the `fetch` function to retrieve data, for instance in the 'GitHubUsers' component, we have:

```
async function fetchGitHubUsers () {  
  const res = await fetch("https://api.github.com/search/users?q=greg"); const json = await  
  res.json();  
  return json.items;  
}
```

By default, *fetch* caches everything in your production build, which is great for performance. However, it can cause issues if the data changes frequently.

In Next.js 13, there's an option called 'revalidate' that instructs Next.js how frequently to check for new data. We'll include this 'revalidate' option in all our `fetch` calls. To do this, we'll add an object after the endpoint in our `fetch` call, specifying *next* and then `'revalidate: 60'`. In `/app/githubusers/page.jsx`, add the following in **bold**:

```
async function fetchGitHubUsers () {  
  const res = await fetch("https://api.github.com/search/users?q=greg",{  
    next:{  
    revalidate: 60  
  })  
  }); const json = await res.json () ;  
  return json.items;  
}
```

Here, '60' represents the number of seconds to wait before refreshing the data. Essentially, it will cache the data for 60 seconds.

If you're dealing with data that changes frequently, you'll want to revalidate often. However, for data that doesn't change as much, like retrieving GitHub users, you might not need to revalidate as frequently. In some cases, you might even consider not using 'revalidate' at all, in which case your

application will behave more like a static website, fetching the data only once.

Similarly, in the 'Repos' component where we also use *fetch*, we'll include 'revalidate' as well. In /app/components/Repos.jsx, add in **bold**:

```
async function fetchRepos(user) {  
  const res = await fetch(`https://api.github.com/users/${user}/repos`, {  
    next: {  
    revalidate: 60  
    }  
  }); const json = await res.json();  
  return json;  
}
```

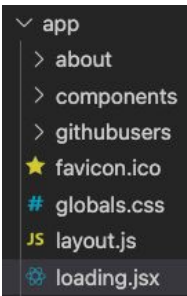
After adding 'revalidate', save the changes and ensure everything still works correctly. You can also adjust the delay in the loading page to a shorter duration for smoother user experience.

With 'revalidate' set, if a new repository is added for a user while we're on their page, it will be picked up and displayed here. If we don't use 'revalidate', then the new repositories won't be reflected in our fetched data.

CHAPTER 15: API ROUTE HANDLERS

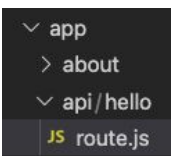
Now, let's discuss API route handlers. API route handlers are functions that are called or executed when a user requests an API route. They handle incoming HTTP requests for a specific route to respond with the necessary data.

Unlike typical full stack apps that have a separate Express API, we can actually incorporate the entire route handlers within our app structure. To demonstrate, in our 'app' directory, create a folder named 'api'.



While you can have route handlers in your page routes, placing them in the 'api' folder means they will be prefixed with '/api' in the endpoint.

For a simple example, let's create another folder within 'api' called 'hello'. Within 'hello', we'll create a file named 'route.js'.

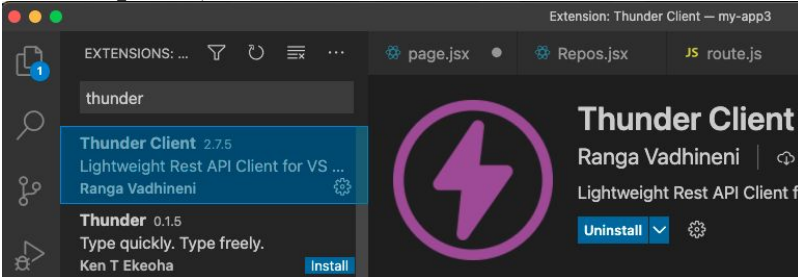


In '/app/api/hello/route.js', add in:

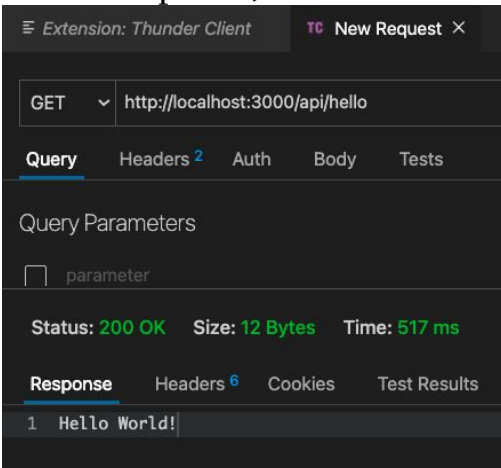
```
export async function GET(req) {  
  return new Response('Hello World!');  
}
```

We export an async function named 'GET' that returns a 'Hello, world' response.

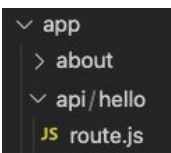
To test this, we can use Postman or, in my case, I use a Visual Studio Code plugin called Thunder Client, which acts like Postman but is conveniently integrated within Visual Studio Code (search 'Extensions' in Marketplace).



Sending a GET request to 'localhost:3000/api/hello' returns the 'Hello, world' response, which we defined earlier.



The name of the route is akin to how we structured our folders.



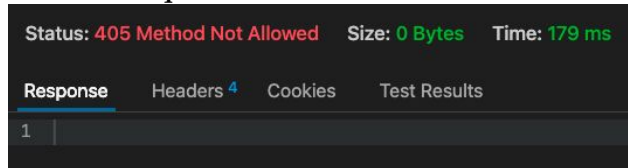
The endpoint is 'api', and under 'api', we have 'hello'. The route's name corresponds to our folder structure, and the actual code resides in the 'route.js' file.

In 'route.js', the function is named after the HTTP request method we want to handle. For instance, if we want to handle a GET request, we name the

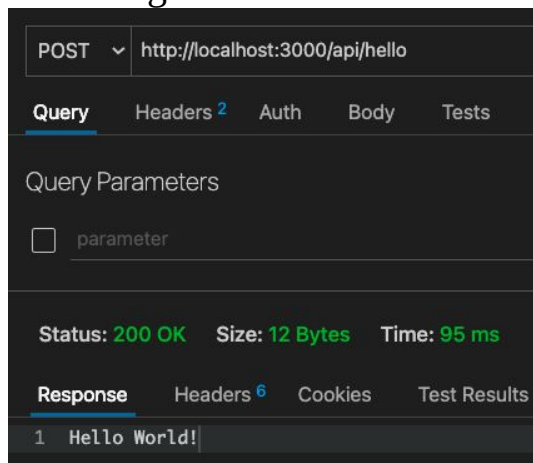
function 'GET'. If we were to change this to 'POST',

```
export async function POST(req) {  
  return new Response('Hello World!');  
}
```

a GET request would return 'Method Not Allowed'.



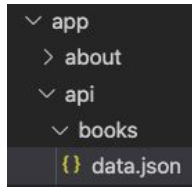
However, changing the request to POST would return a 200 status, indicating success.



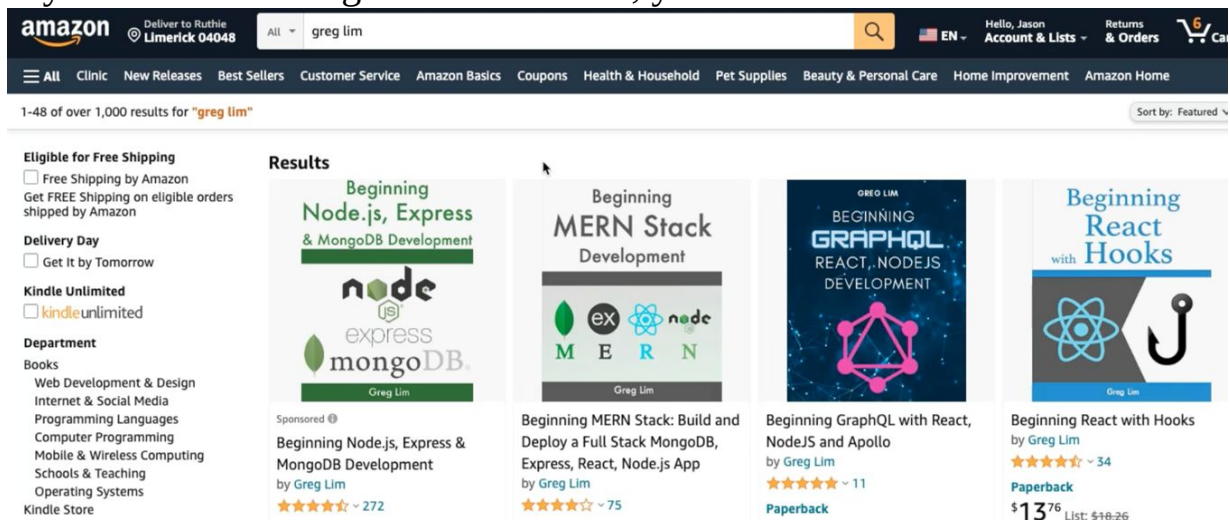
CHAPTER 16: SERVING ‘BOOKS’ DATA

Our next step involves using our route handlers to fetch some book data. In my case, as part of my portfolio, I've written several books available on Amazon, which contribute to my income stream. I want to use this API route handler to allow users to fetch the books I've authored. Initially, we will be accessing the data from a JSON file. Eventually, we will replace this with a database, specifically Prisma, but you can choose any database you prefer.

First, let's create a new folder under 'api' called 'books'. Inside 'books', we'll create 'data.json'.



If you search for Greg Lim on Amazon, you'll find the books I've written.



We'll represent some of these books in the JSON file.

In the JSON file, we'll start with attributes such as 'id', 'title', 'link' (the Amazon link), and 'image'. Here's the details for the first three books (to be filled into data.json).

```
[
  {
    "id": 1,
    "title": "MERN Stack",
    "link": "https://www.amazon.com/Beginning-MERN-Stack-MongoDB-Express/dp/B0979MGJ5J", "img": "https://m.media-amazon.com/images/I/41y8qC9RT0S._SX404_BO1,204,203,200_.jpg"
  },
  {
    "id": 2,
    "title": "Beginning GraphQL",
    "link": "https://www.amazon.com/Beginning-GraphQL-React-NodeJS-Apollo/dp/B0BXMRB5VF/", "img": "https://m.media-amazon.com/images/I/41+PG6uPdHL._SX404_BO1,204,203,200_.jpg"
  },
  {
    "id": 3,
    "title": "Beginning React Hooks",
    "link": "https://www.amazon.com/Beginning-React-Hooks-Greg-Lim/dp/B0892HRT3C/", "img": "https://m.media-amazon.com/images/I/41e9U1d9QIL._SX404_BO1,204,203,200_.jpg"
  }
]
```

If you wish, you can use your own data. This JSON file will be included in the source code (contact support@i-ducate.com) for your reference.

Next, under the 'books' folder, we'll create a 'route.js' file where we will define a route to return the books. Fill in the following:

```
import books from './data.json';
import { NextResponse } from 'next/server';

export async function GET(req) {
  return NextResponse.json(books);
}
```

Firstly, we import the books from the JSON file. We also need to import 'NextResponse' from 'next/server'. Our function is named 'GET' because we're handling a GET request, and inside this function, we return 'NextResponse.json(books)', where 'books' is the data from our JSON file.

Testing our App

Using Thunder Client, if we send a GET request to 'api/books', we should receive the book data in the response:

GET

http://localhost:3000/api/books

Query

Headers 2

Auth

Body

Tests

Query Parameters

☐

parameter

value

Status: 200 OK

Size: 607 Bytes

Time: 397 ms

Response

Headers 6

Cookies

Test Results

1

<

{

2

3

4

5

6

7

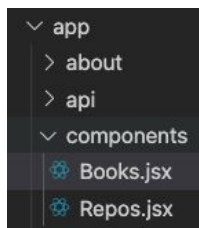
}

,

CHAPTER 17: FETCHING ‘ BOOKS ’ DATA

We'd like to display these books on our home page to showcase my Amazon books.

To accomplish this, in /app/components, I'll create a component named 'Books.jsx'.



Add in the following codes in Books.jsx:

```
import Link from "next/link";
```

```
async function getBooks() {  
  const res = await fetch("http://localhost:3000/api/books");  
  const json = await res.json();  
  return json;  
}
```

```
const Books = async () => {  
  const books = await getBooks();  
  return(  
    <div>  
      <h1>Books</h1>  
    </div>  
  )  
}
```

```
export default Books;
```

Code Explanation

In this component, we import 'Link' from 'next/link', as we want to provide a link to each book's Amazon page.

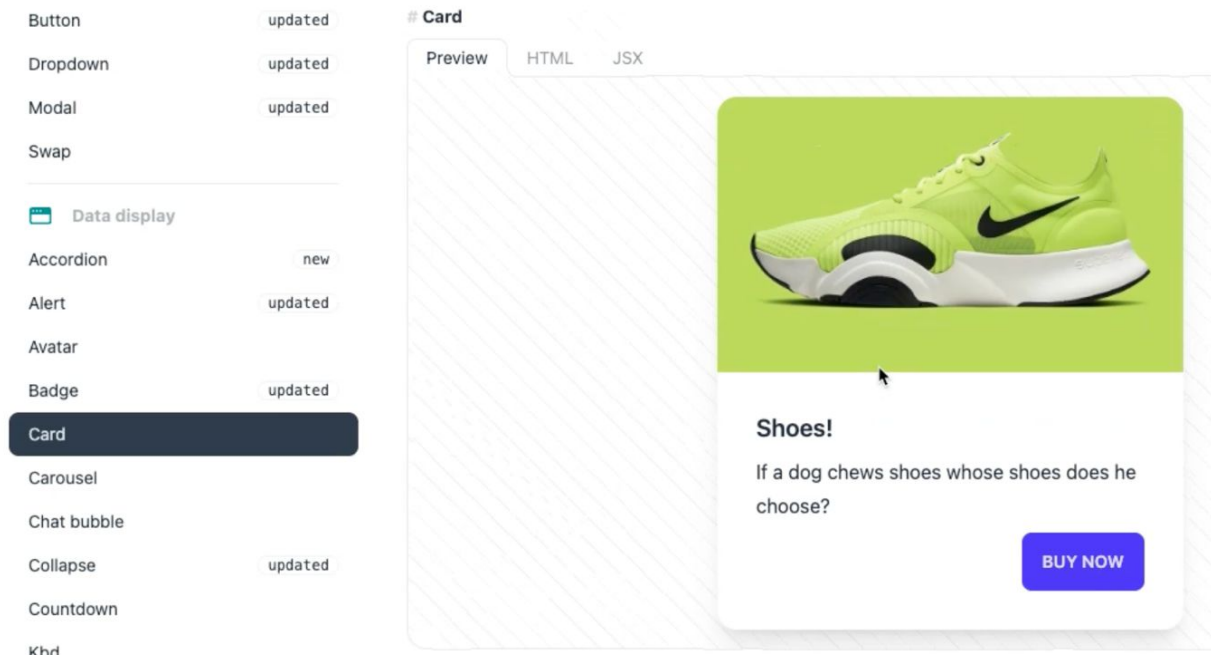
```
async function getBooks() {  
  const res = await fetch("http://localhost:3000/api/books");  
  const json = await res.json();  
  return json;  
}
```

Next, we'll define an async function *getBooks* to fetch the books. The endpoint we're fetching from is the one where we have our books, and we'll parse the response as JSON.

```
const Books = async () => {  
  const books = await getBooks();  
  return(  
    <div>  
      <h1>Books</h1>  
    </div>  
  )  
}
```

Inside the Books component, we'll call *getBooks* using 'await' since it's asynchronous. Consequently, this part of the code also needs to be labeled as 'async'.

Mapping through the books array We'll then map through the books array. Each book will be displayed using a 'Card' component from Daisy UI.



Add in the following in **bold**: ...

```
const Books = async () => {
```

```
const books = await getBooks () ;
```

```
return(
```

```
  <div>
```

```
    <h1>Books</h1>
```

```
    {books.map((book) => (
```

```
      <div key={book.id}> <div className="card w-96 bg-base-100 shadow-xl"> <figure> <img
src={book.img} width="200" height="150" /> </figure> <div className="card-body"> <h2
className="card-title">{book.id}</h2> <p>{book.title}</p> <div className="card-actions
justify-end"> <Link href={book.link} className="btn btn-primary">See in Amazon</Link>
<button className="btn btn-error"> Delete </button> </div> </div> </div> <br /> </div> )}
```

```
    </div>
```

```
)
```

```
}
```

(you can copy-paste above from the source code)

Code Explanation

Using the card component as a template, we'll iterate over the books array, rendering each book within its own card.

```
{books.map((book) => (
```

```

<div key={book.id}> <div className="card w-96 bg-base-100 shadow-xl"> <figure>
  <img src={book.img} width="200" height="150" /> </figure>
  <div className="card-body"> <h2 className="card-title">{book.id}</h2> <p>{book.title}</p>

```

We provide book id as the unique key for each card. We fill the card's properties, such as 'image source' and 'title', with the corresponding book data.

<Link href={book.link} className="btn btn-primary">See in Amazon</Link> For the button, we'll replace it with a link to the respective Amazon page.

```

  <button className="btn btn-error"> Delete
</button>

```

We also include a Delete button. We will implement it later.

Calling Books from HomePage Once our Books component is ready, we include it in the home page with the following code.

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

```

const HomePage = () => {
  return (
    <>
      <Books />
    </>
  );
}

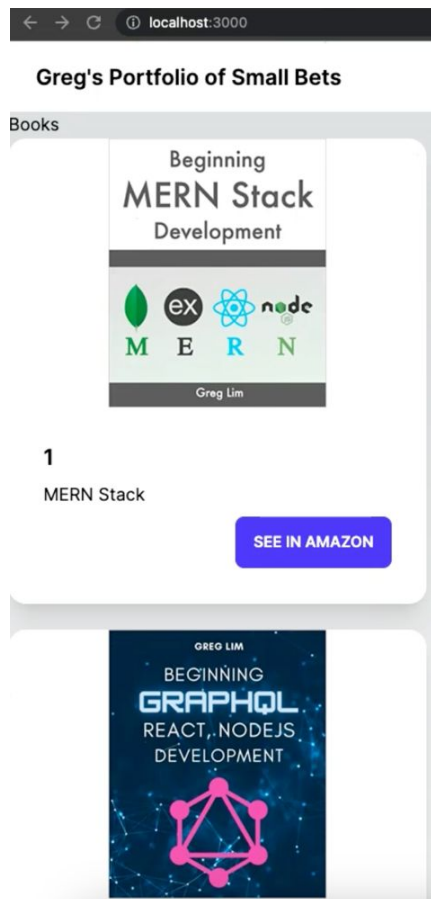
```

```
export default HomePage;
```

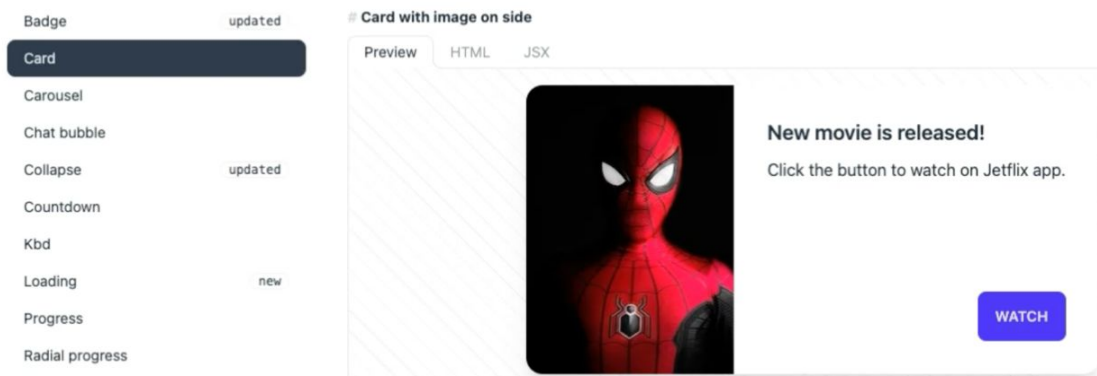
We import and add the Books component into the home page component and remove the extra links, as we already have the navigation bar with the necessary links.

Running our App

Upon visiting the website, you'll see the books displayed in their individual cards, complete with images, IDs, titles, and a 'See in Amazon' link that redirects you to the appropriate Amazon page.



We've created an API route handler that retrieves book data from our 'data.json' file. We plan to transition this to pulling data from a Prisma database later. Finally, if you'd like to change the look of the card, feel free to experiment with different card component styles, eg.



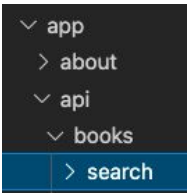
CHAPTER 18: GETTING SEARCH PARAMS

Next, we want to see how we can obtain search query parameters so that we can implement a book search function.

In our API endpoint, we're currently retrieving all the books. Suppose we want a route that contains a search query, e.g., 'localhost:3000/api/books/**search?query=graphql**'

We'll explore how to obtain 'graphql' so we can search through existing books.

First, we need to create the 'search' route. That means we need to create another folder named 'search' under the 'books' folder. Remember that the structure forms the route for a page.



Under 'search', we create the file 'route.js' and implement our endpoint route handler by filling in 'route.js' with the following code:

```
import { NextResponse } from "next/server";
import books from "../data.json";

export async function GET(req) {
  const { searchParams } = new URL(req.url);
  console.log(searchParams)

  return NextResponse.json(books);
}
```

Code Explanation

```
export async function GET(req) {
```

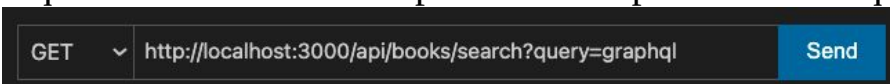
We create a function for handling GET requests.

```
const { searchParams } = new URL(req.url);  
console.log(searchParams)
```

We destructure *searchParams* from the new URL, passing in 'request.url'. *searchParams* will contain any query parameters that we include. Let's log these parameters to the console for now to check if they're correctly obtained.

Testing

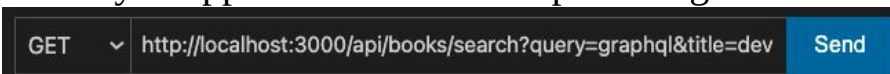
To test this, we can use Thunder client or similar software to send a GET request to our 'search' endpoint with a specific search query.



In the Terminal, you will see *URLSearchParams { 'query' => 'graphql' }* being logged:

```
- wait compiling /api/books/search/route (client and server)...  
- event compiled successfully in 195 ms (304 modules)  
URLSearchParams { 'query' => 'graphql' }
```

And if you append another search param e.g.:



You get:

```
URLSearchParams { 'query' => 'graphql', 'title' => 'dev' }
```

So *searchParams* will contain all the params we can access.

To get a particular param value, we pass in the parameter name eg 'searchParams.get('query')':


```
...
export async function GET(req) {
  const { searchParams } = new URL(req.url);
  console.log(searchParams.get('query')) // or searchParams.get('title')

  return NextResponse.json(books);
}
```

Filter Books based on Search Query

Finally, we want to filter our books based on the search query. Add the following in **bold**:

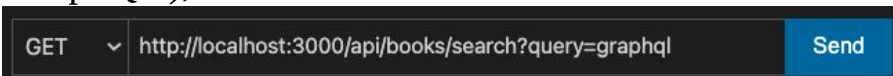
```
...
export async function GET(req) {
  const { searchParams } = new URL(req.url);
  const query = searchParams.get('query');

  const filteredBooks = books.filter((book) => {
    return book.title.toLowerCase().includes(query.toLowerCase());
});

  return NextResponse.json(filteredBooks);
}
```

We retrieve the query and store it in a constant. We then get our filtered books by looping through the books array, convert each book title to lowercase, and check if it includes the query (which is also converted to lowercase). This eliminates any case-sensitive differences. Lastly, we return the filtered books.

Now, when we save this and run a search for a specific query (e.g., 'GraphQL'),



A screenshot of a web browser's address bar. On the left, there is a dropdown menu showing 'GET'. To its right is the URL 'http://localhost:3000/api/books/search?query=graphql'. On the far right, there is a blue button labeled 'Send'.

it should return only the relevant GraphQL books.

```
Status: 200 OK   Size: 207 Bytes   Time: 100 ms

Response  Headers 6  Cookies  Test Results  {}

1 [
2   {
3     "id": 2,
4     "title": "Beginning GraphQL",
5     "link": "https://www.amazon.com/Beginning-GraphQL-React-NodeJS-Apollo
6     "img": "https://m.media-amazon.com/images/I/41+PG6uPdHL._SX404_B01,204
7   }
8 ]
```

If we change the query (e.g., to 'Mern'), we'll see only the Mern stack book.

```
GET http://localhost:3000/api/books/search?query=mern Send

Query  Headers 2  Auth  Body  Tests

Query Parameters

[✓] query mern
[] parameter value

Status: 200 OK   Size: 198 Bytes   Time: 128 ms

Response  Headers 6  Cookies  Test Results  {}

1 [
2   {
3     "id": 1,
4     "title": "MERN Stack",
5     "link": "https://www.amazon.com/Beginning-MERN-Stack-MongoDB-Express
6     "img": "https://m.media-amazon.com/images/I/41y8qC9RT0S._SX404_B01,204
7   }
8 ]
```

Chapter 19: Adding a New Book

We have a route handler that performs a GET request to retrieve filtered books. Suppose we want to add a new book, we'll probably need to access the request body as well. So how do we go about that?

We'll need to go back to `/app/api/books/route.js`, where we have an existing route handler for GET requests, we need to add one for POST requests as well. Add in the following in **bold**: `import books from './data.json';`

```
import { NextResponse } from 'next/server';

export async function GET(req) {
  return NextResponse.json(books); }

export async function POST(req) {
  const { title, link, img } = await req.json();
  const newBook = {
    id: books.length + 1, title,
    link,
    img
  };
  books.push(newBook); return NextResponse.json('Book added successfully'); }
```

Code Explanation

```
const { title, link, img } = await req.json();
```

First, we'll need to retrieve the body data from the `request.json`. We'll use `'await'` and then destructure `'title'`, `'link'`, and `'image'` from `'request.json'`. These will come from a form that we'll implement later and will be included in the body of the request.

```
const newBook = {
  id: books.length + 1,
  title,
  link,
  img
};
```

With these fields, we can proceed to create a new book object. We'll set the `'id'` to the number of books in the array plus one. This is a temporary

solution for generating IDs, and we'll replace this later when we have our Prisma database.

```
books.push(newBook);
```

Since we're not yet using a database, we'll add this new book directly to the 'books' array in memory.

```
return NextResponse.json('Book added successfully');
```

Finally, we'll return a successful response 'Book added successfully'. We're not maintaining any data persistence yet. We'll implement that later with a Prisma database. Later, we'll revisit this POST request and add a user interface for it.

CHAPTER 20: REFACTOR SERVER TO CLIENT COMPONENT

Let's now create the frontend that uses our search route we implemented earlier: `/app/api/books/search/route.js ...`

```
export async function GET(req) {
  const { searchParams } = new URL(req.url); const query = searchParams.get('query')
  const filteredBooks = books.filter((book) => {
    return book.title.toLowerCase().includes(query.toLowerCase());
  });
  return NextResponse.json(filteredBooks);
}
```

We will be incorporating the search feature in the Books component (`/app/components/Books.jsx`). We later also allow Books component to add and delete books. Given the interactivity involved, we will transition our Books server component into a client component. This will also help illustrate when it's preferable to use server components versus client components.

Firstly, we'll convert this to a client component because we'll be using *state* and the *useEffect* hook. We do this by adding 'useClient' at the top. In `Books.jsx`, add the below:

```
/app/components/Books.jsx
'use client'; import { useState, useEffect } from "react"; import Link from "next/link";
const Books = async() => {
  ...
  ...
}
```

Loading Page for Client Components

Remember the loading page? The automatic loading is only for server components when they are loading data. However, since we're fetching data from our client, this automatic loading doesn't work. But we can manually import it into our books client component. In our books component, we'll import the loading page:

```
'use client'; import { useState, useEffect } from "react"; import Link from "next/link"; import
LoadingPage from "../loading";
const Books = async () => {
  ...
}
```

...

Next, in our books component, we'll have a state for our books. Add: ...

...

```
const Books = async () => {  
  const [books, setBooks] = useState([]); ...
```

...

This book state will change based on the user's interaction. For instance, when the page first loads, it will contain all the books, but after a search, it will only contain the filtered books.

We also want to maintain a *loading* state to track whether the component is loading. Make the following code changes in **bold**: ...

...

```
const Books = async () => {  
  const [books, setBooks] = useState([]); const [loading, setLoading] = useState(true);  
useEffect() => {  
  getBooks().then((books) =>{  
    setBooks(books) ; setLoading(false) ; });  
}, []);
```

```
const books = await getBooks();
```

We'll also have a *useEffect* hook, inside which we'll call 'getBooks'. *useEffect* will make an API request to 'books', get the JSON data, set it in the state, and set loading to false to indicate loading has completed.

Lastly, in our render function, if the component is loading, we'll return the loading page. Add the following: **if (loading){ return <LoadingPage />}**

```
return(
```

...

...

The rendering of books should remain the same, so when we run this, we should still render all our books. The key difference now is that all of this is happening on the client side.

Here's the final code of Books.jsx in case you got lost at any point: 'use client';

```
import { useState, useEffect } from "react"; import Link from "next/link";
import LoadingPage from "../loading";
```

```
async function getBooks() {
const res = await fetch("http://localhost:3000/api/books"); const json = await res.json();
return json;
}
```

```
const Books = async () => {
const [books, setBooks] = useState([]);
const [loading, setLoading] = useState(true);
useEffect(() => {
  getBooks().then((books) =>{
    setBooks(books) ;
    setLoading(false) ;
  });
}, []);
```

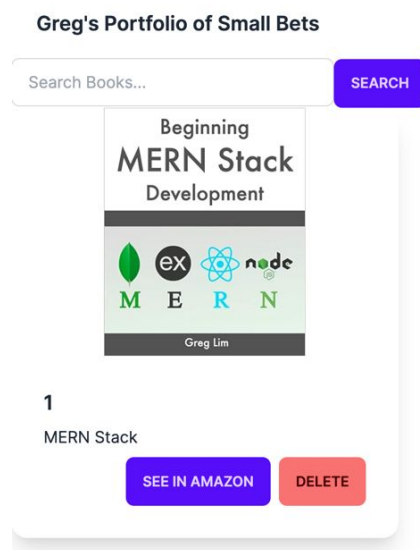
```
if (loading){ return <LoadingPage />}
```

```
return(
  <div>
    <h1>Books</h1>
    {books.map((book) => (
      <div key={book.id}>
        <div className="card w-96 bg-base-100 shadow-xl"> <figure>
          <img src={book.img} width="200" height="150" /> </figure>
          <div className="card-body"> <h2 className="card-title">{book.id}</h2> <p>{book.title}</p>
        <div className="card-actions justify-end"> <Link href={book.link} className="btn btn-
primary">See in Amazon</Link> <button className="btn btn-error"> Delete
          </button>
        </div>
      </div>
    )
    </div>
    <br />
  )
)}
```

```
    </div>
  )
}
export default Books;
```


CHAPTER 21: SEARCH COMPONENT

We are going to add a search form at the top, above the book listings to allow users to enter their search queries.



In Books.jsx, add:

...

```
const Books = async () => {  
  const [books, setBooks] = useState([]);  
  const [loading, setLoading] = useState(true); const [query, setQuery] = useState("");  
  useEffect(() => {  
    ...  
  }, []);
```

```
  if (loading){ return <LoadingPage />}
```

```
  const handleSubmit = async (e) => {  
  e.preventDefault(); }
```

```
  return (  
    <div>  
      <form onSubmit={handleSubmit}> <input type="text"
```

```

        placeholder="Search Books..."
        value={query}
        onChange={(e) => setQuery(e.target.value)} className="input
        input-bordered w-full max-w-xs"
    /> <button type="submit"
        className="btn btn-primary"> Search
    </button> </form>
    {books.map((book) => (
    <div key={book.id}>
        ...
        ...

    </div>
    )))}
    </div>
    )
}
export default Books;

```

Code Explanation `const handleSubmit = async (e) => {`

`e.preventDefault(); }`

`return (`

`<div>`

`<form onSubmit={handleSubmit}>`

`<input type="text"`

`placeholder="Search Books..."`

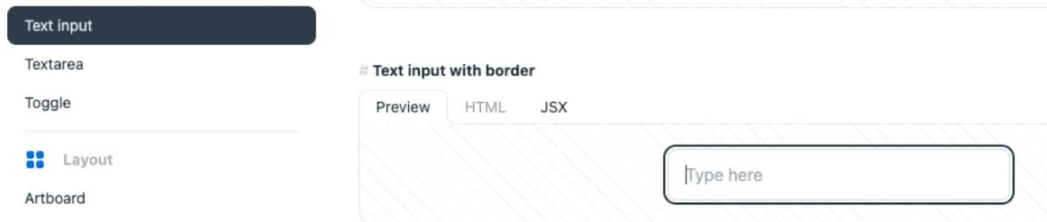
`value={query}`

`onChange={(e) => setQuery(e.target.value)} className="input`

`input-bordered w-full max-w-xs"`

`/>`

We add a form. When submitted, this form will trigger the *handleSubmit* function to handle the submission (will implement it fully later). Within this form, we use a simple input field with a border from DaisyUI.



We copy the markup and insert it in the form. The input type will be text, and the placeholder will be 'Search Books'.

```
const Books = () => {
  const [books, setBooks] = useState([]);
  const [loading, setLoading] = useState(true); const [query, setQuery] = useState("");
```

For its value, we link it to a state variable we just created called 'query'. The default value is an empty string.

```
onChange={(e) => setQuery(e.target.value)}
```

The 'onChange' event updates the 'query' state to reflect the user's input, i.e., 'e.target.value'.

```
<form onSubmit={handleSubmit}> <input type="text"
  ...
/>
<button type="submit"
  className="btn btn-primary"> Search
</button> </form>
```

Beneath the input field, we add a search button.

Testing our App Let's test our form to ensure it's functioning correctly. If you run your app, it seems that every state update triggers a server loading page because we have marked our component as *async*: `const Books = async () => {`

Now that our component is a client component, we get the following

warning:

```

Warning: Hooks are not supported inside an async component. This error is often caused by accidentally adding `use client` to a module that was originally written for the server.
  at app-index.js:31

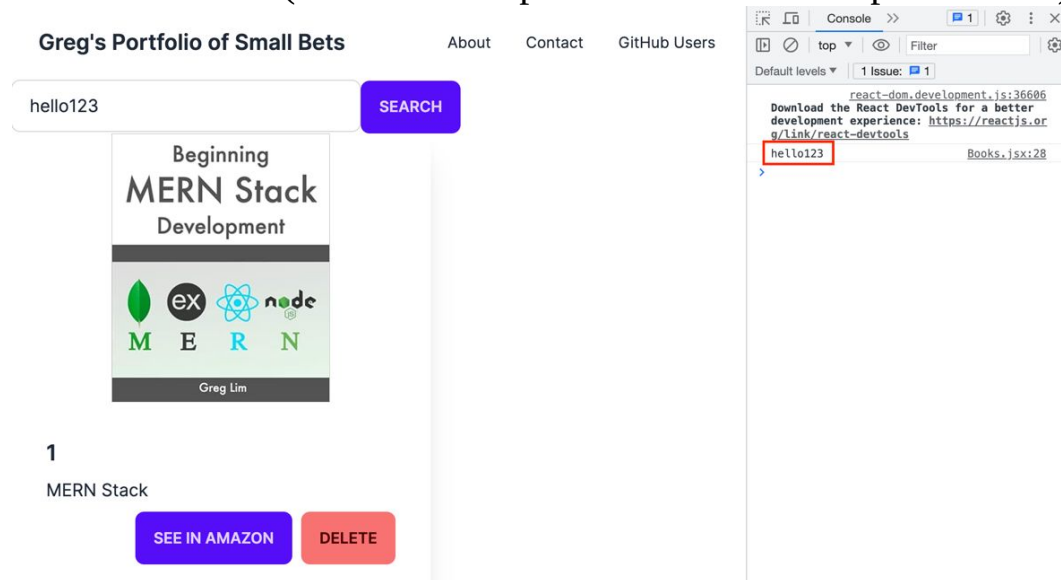
```

So we don't need *async* anymore and remove it: `const Books = async () => {`

Now, our form should function properly. To verify this, we can print *query* to the console:

```
const handleSubmit = async (e) => {  
  e.preventDefault();  
  console.log(query) }  
}
```

If everything is working correctly, we should see our input logged in the browser console (since our component is a client component now).



Fetching Data Using the Search Query

The final step is to fetch data from our search API route using the search query. In

handleSubmit, add: `const handleSubmit = async (e) => {`

```
  e.preventDefault();  
  setLoading(true);  
  const res = await fetch(`/api/books/search?query=${query}`); const books = await res.json();  
setBooks(books);  
  setLoading(false);  
}
```

Code Explanation `const res = await fetch(`/api/books/search?query=${query}`);` `const books = await res.json();`

After fetching, we'll store the result in a 'books' variable.

```
setBooks(books);
```

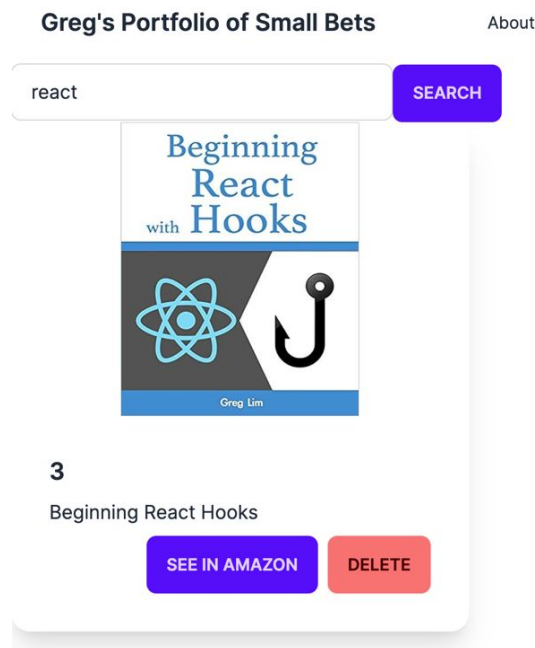
We'll then set 'books' as our state.

```
setLoading(true); const res = await fetch(`/api/books/search?query=${query}`); const books =  
await res.json();  
    setBooks(books);  
    setLoading(false);
```

Before fetching, we set 'loading' to true, and after fetching, we should set it back to false.

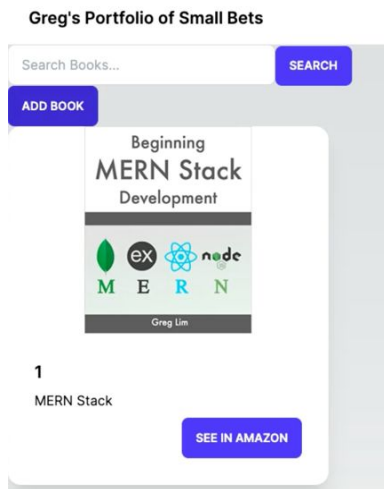
Running our App

After saving these changes, we can search for books, such as 'React' or 'MERN', and receive corresponding results.



A blank search should return all books.

CHAPTER 22: ADDBOOK COMPONENT



We now want to implement the frontend to create a new book. If you remember, under `/app/api/books/route.js`, we have a POST route for adding a new book.

```
export async function POST(req) {  
  const { title, link, img } = await req.json();  
  
  const newBook = {  
    id: books.length + 1,  
    title,  
    link,  
    img  
  };  
  
  books.push(newBook);  
  return NextResponse.json('Book added successfully');  
}
```

Our task now is to create the front-end to send a request to this POST route.

In our 'Books' component, below the submit form, we'll render a new 'AddBook' component by adding in **bold**: `/app/components/Books.jsx` ...

```
import LoadingPage from "../loading"; import AddBook from './AddBook';
```

```

const Books = () => {
  ...
  ...

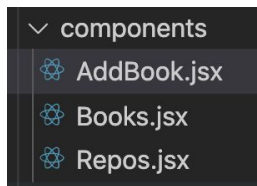
  return (
    <div>
      <form onSubmit={handleSubmit}> ...
    </form>
    <AddBook /> {books.map((book) => (
      <div key={book.id}> ...
    </div>
    ))}
  </div>
  )
}

export default Books;

```

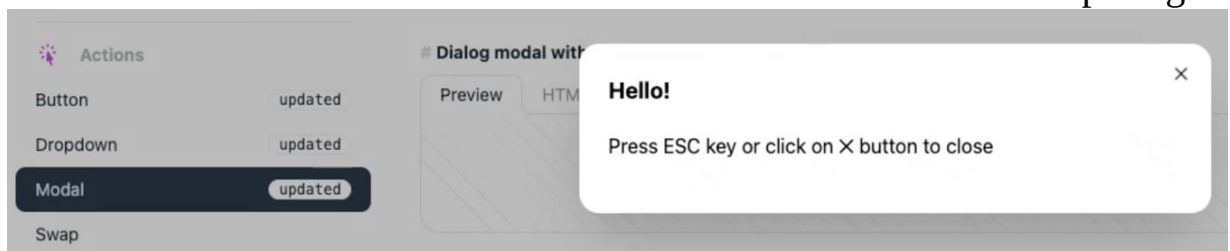
We start by importing 'AddBook', although we haven't created it yet.

Then, under the components directory, we'll create the 'AddBook.jsx' file.



The 'AddBook' component will be a client component as we'll be using 'useState' for form interactivity.

When users click 'Add book', we want a modal form to open, allowing them to enter the details for the new book. We'll utilize a modal from Daisy UI. We'll select a modal with a close button on the top right:



In AddBooks.jsx, fill in the following:

```

'use client';
import { useState } from 'react';

const AddBook = () => {
  const [modalOpen, setModalOpen] = useState(false);

```

```

return (
  <div>
    <button className="btn" onClick={()=>setModalOpen(true)}> Add Book
      </button> <dialog id="my_modal_3"
        className={`modal ${modalOpen ? "modal-open" : ""}`}> <form
        method="dialog" className="modal-box"> <button onClick=
        {()=>setModalOpen(false)}
          htmlFor="my-modal-3"
          className="btn btn-sm btn-circle btn-ghost absolute right-
            2 top-2"> ✕
        </button> <h3 className="font-bold text-lg">Add New Book</h3> </form>
      </dialog>
    </div>
  )
}
export default AddBook;

```

Code Explanation

```

const AddBook = () => {
  const [modalOpen, setModalOpen] = useState(false);

```

We handle the modal's open/close state by having a state variable 'modalOpen', which is initially set to false.

```

const AddBook = () => {
  ...
  return (
    <div>
      <button className="btn" onClick={()=>setModalOpen(true)}> Add Book
        </button> <dialog id="my_modal_3"
          className={`modal ${modalOpen ? "modal-open" : ""}}`> <form
          method="dialog" className="modal-box">

```

If 'modalOpen' is true, we add the 'modal-open' class (to reveal the modal form).

```

const AddBook = () => {
  ...
  return (
    <div>
      <button className="btn" onClick={()=>setModalOpen(true)}> Add Book
        </button>

```

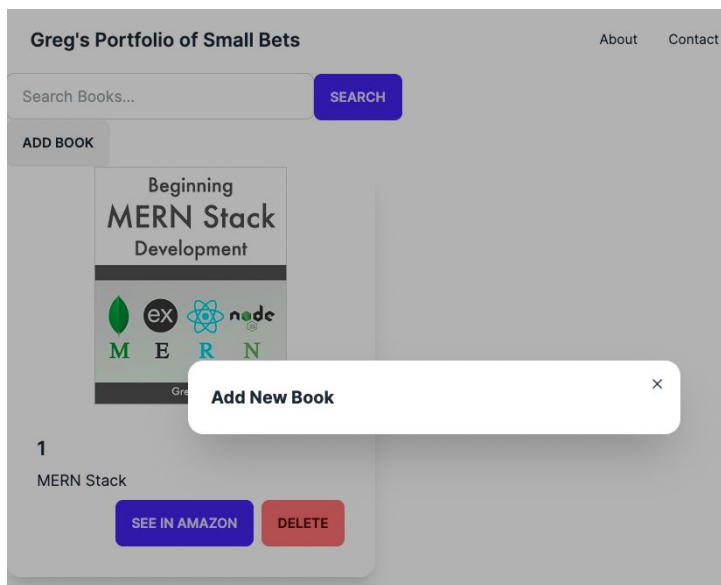
On the button's 'onClick' event, we'll set 'modalOpen' to true, causing the modal to open.


```
'use client';
import { useState } from 'react';
...
```

We can utilize 'useState' because we've specified this as a client component. If we didn't specify this, it would default to a server component, and 'useState' would be unusable.

```
<form method="dialog" className="modal-box"> <button onClick=
{()=>setModalOpen(false)}
htmlFor="my-modal-3"
className="btn btn-sm btn-circle btn-ghost absolute right-
2 top-2"> ✕
</button> <h3 className="font-bold text-lg">Add New Book</h3> </form>
```

We set 'modalOpen' to false when the 'X' button is clicked by adding an 'onClick' event to it and setting 'modalOpen' to false.



After implementing these changes, the 'Add book' button should open the modal, and the 'X' button should close it.

CHAPTER 23: CALLING THE ADDBOOK ENDPOINT

Next, we need some input fields in AddBook component. In /app/components/AddBook.jsx, add: ...

```
<form method="dialog" className="modal-box"> <button onClick={()=>setModalOpen(false)}
                                htmlFor="my-modal-3"
                                className="btn btn-sm btn-circle btn-ghost absolute right-
                                2 top-2"> ✕
    </button> <h3 className="font-bold text-lg">Add New Book</h3> <input
    type="text"
    placeholder="Enter New Book Title"
    className="input input-bordered w-full max-w-xs"
    /> <button type="submit" className='btn btn-primary'> Add Book
    </button> </form> ...
```

Below the input, we'll also add a ' Add Book ' button.

Form Submission Now, we handle the form submission by adding the following in **bold**: 'use client';

```
import { useState } from 'react';
const AddBook = () => {
    const [modalOpen, setModalOpen] = useState(false); const [newBookTitle, setNewBookTitle] =
useState('');
    const handleSubmitNewBook = (e) =>{
        e.preventDefault();
console.log(newBookTitle);
setNewBookTitle('');
    }
    ...
    ...
    ...
    <form method="dialog" className="modal-box"
        onSubmit={handleSubmitNewBook}> <button          onClick=
        {()=>setModalOpen(false)}
        htmlFor="my-modal-3"
        className="btn btn-sm btn-circle btn-ghost absolute right-
        2 top-2"> ✕
    </button> <h3 className="font-bold text-lg">Add New Book</h3> <input
    type="text"
```

```

        value={newBookTitle}
        onChange={e => setNewBookTitle(e.target.value)}
        placeholder="Enter New Book Title"
        className="input input-bordered w-full max-w-xs"
      />
      <button type="submit" className='btn btn-primary'> Add Book </button>
    </form>

```

Code Explanation <form method="dialog" className="modal-box"

onSubmit={handleSubmitNewBook}> We apply the form's 'onSubmit' event and link it to a function named 'handleSubmitNewBook'

```

const AddBook = () => {
  const [modalOpen, setModalOpen] = useState(false);
  const [newBookTitle, setNewBookTitle] = useState('');

```

To keep track of the new book title, we create a *newBookTitle* state with 'useState', setting it initially to an empty string.

```

const handleSubmitNewBook = (e) =>{
  e.preventDefault();
  console.log(newBookTitle);
  setNewBookTitle('');
}

```

'handleSubmitNewBook' will prevent the default form submission event and log the new book title to the console. After submitting the form, we set the 'newBookTitle' state back to an empty string. This way, the input field will be cleared after each submission.

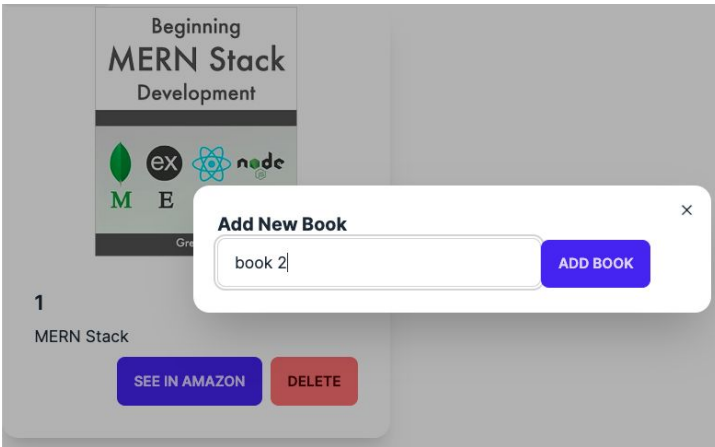
```

    <input type="text"
      value={newBookTitle}
      onChange={e => setNewBookTitle(e.target.value)}
      placeholder="Enter New Book Title"
      className="input input-bordered w-full max-w-xs"
    />

```

We then set the value of our input field to the 'newBookTitle' state. Also, we include an 'onChange' event that updates the 'newBookTitle' state every time the input field changes.

Testing your App Run your app now and try to add a book. The book title you entered in the Add Book form will be logged in the browser console.



Extending your Form For the book, there will also be the book's Amazon site and image links. We have illustrated adding the book title. As an exercise, add the image and link input fields on your own. They will follow the same pattern.

Revising 'handleSubmitNewBook'

Next, let's revise our 'handleSubmitNewBook' function. Instead of console logging, we want to call the API endpoint to add a book. In `/app/components/AddBook.jsx`, add the codes in **bold**: ...

```
const AddBook = () => {
  ...
  ...
  const handleSubmitNewBook = async (e) =>{
    e.preventDefault();
    const res = await fetch(`/api/books/`,{
      method: 'POST', headers:{
        'Content-type': 'application/json'
      }, body: JSON.stringify({
        title: newBookTitle, link: " https://www.amazon.com/dp/B0979MGJ5J", img: " https://via.placeholder.com/600/92c952 "
      }) })
    if(res.ok){
      setNewBookTitle(""); setModalOpen(false); }
    }
  }
}
```

Code Explanation `const res = await fetch('/api/books',{`

```
  method: 'POST', headers:{
    'Content-type': 'application/json'
  }, body: JSON.stringify({
    title: newBookTitle, ...
  }) })
```

We use 'fetch' to send a POST request to the '/api/books' endpoint, setting the headers to 'application/json' and the body to our new book details.

```
const handleSubmitNewBook = async (e) =>{
```

Ensure to label 'handleSubmitNewBook' as 'async' because we're using 'await'.

```
  body: JSON.stringify({
    title: newBookTitle, link: "  https://www.amazon.com/dp/B0979MGJ5J", img: "
https://via.placeholder.com/600/92c952 "
  })
```

We specify the title from *newBookTitle*. We also add placeholders for the *link* and *image* properties, which you can replace later with actual user input. For now, I'll hardcode these values for testing purposes.

uuid

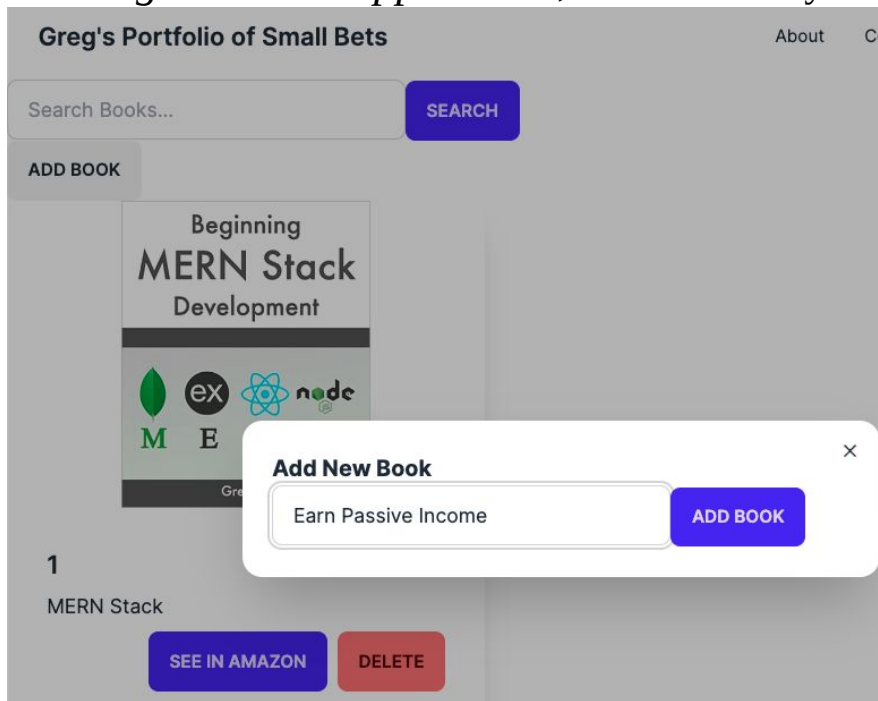
One more thing before we test our app. In the add books API route(/app/api/books/route.js), we will import and use 'uuidv4' to generate our book ids instead (run 'npm install uuidv4' in Terminal): import books from './data.json'; import { NextResponse } from 'next/server'; **import { v4 as uuidv4 } from 'uuid';**

```
export async function GET(req) {
  ...
}
```

```
export async function POST(req) {
  const { title, link, img } = await req.json();
  const newBook = {
    id: uuidv4(), title, link,
    img
  };
}
```

```
books.push(newBook); return NextResponse.json('Book added successfully'); }
```

Running our App Now, let ' s try adding a book:



The book will be added, the input field cleared, and the modal closed. But we meet a problem. The new book isn't immediately shown on the page. Let ' s address that in the next chapter.

CHAPTER 24: REFRESHING BOOKS AFTER ADDING

Currently, when we try to add a new book, the new book isn't immediately shown on the page. To fix this, we want to refresh our main page each time a new book is added.

Essentially, we need a 'refreshBooks' method in our AddBook component, which we call when a book is successfully added. Add in

```

/app/components/AddBook.jsx,
'use client';
import { useState } from 'react';

const AddBook = ({refreshBooks}) => {
  const [modalOpen, setModalOpen] = useState(false);
  const [newBookTitle, setNewBookTitle] = useState("");

  const handleSubmitNewBook = async (e) =>{
    ...

    if(res.ok){
      setNewBookTitle("");
      setModalOpen(false);
      refreshBooks();
    }
  }

  return (
    ...
  )
}
export default AddBook;

```

But where do we get this 'refreshBooks' method from? It will come from our 'Books' component. In /app/components/Books.jsx, add in **bold**:

```

const Books = () => {
  ...
  useEffect(() => {
    getBooks().then((books) =>{
      setBooks(books) ;
      setLoading(false) ;
    });
  }, []);
  ...
  ...

```

```

return (
  <div>
    <form onSubmit={handleSubmit}>
      ...
    </form>
    <AddBook refreshBooks={}> {books.map((book) => (
      ...
    ))}
  </div>
)
}
export default Books;

```

We pass in 'refreshBooks' as a prop to AddBook component. But where do we implement *refreshBooks*?

Looking at our 'useEffect' hook in Books component,

```

useEffect(() => {
  getBooks().then((books) =>{
    setBooks(books) ;
    setLoading(false) ;
  });
}, []);

```

We fetch the list of books directly within *useEffect*. We refactor this into a separate 'fetchBooks' method, which we can later pass to 'refreshBooks'.

Make the code changes in **bold**:

```

const Books = () => {
  const [books, setBooks] = useState([]);
  const [loading, setLoading] = useState(true);
  const [query, setQuery] = useState("");

  const fetchBooks = async () => {
const res = await fetch("/api/books");
const books = await res.json();
setBooks(books);

```



```
    setLoading(false);  
  }
```

```
  useEffect(() => {  
    fetchBooks();  
  },[]);  
  ...
```

Code Explanation

We create a new asynchronous function called 'fetchBooks' and move the book fetching logic into this function. In it, we also set 'books' and 'loading' states.

```
useEffect(() => {  
  fetchBooks();  
},[]);
```

In *useEffect*, instead of using 'getBooks', use 'fetchBooks'.

Reuse 'fetchBooks' in 'refreshBooks'

This allows us to reuse 'fetchBooks' in 'refreshBooks'.

```
    return (  
      <div>  
        <form onSubmit={handleSubmit}>  
          ...  
        </form>  
        <AddBook refreshBooks={fetchBooks}/> {books.map((book) => (  
          ...  
        ))}  
      </div>  
    )
```

Running your App

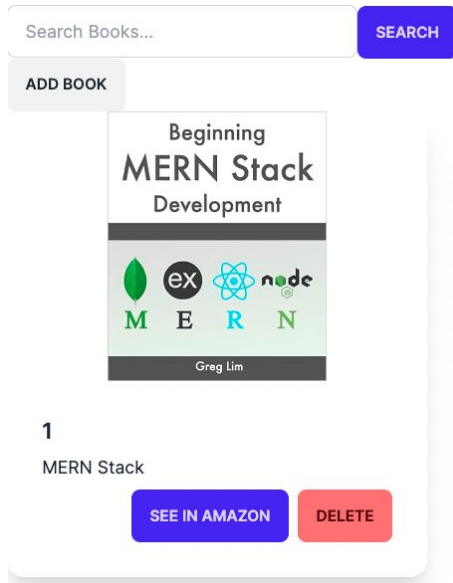
Upon saving and running the app, we should see the added book immediately.

The book image and link are still hardcoded, but you can change this on your own. Simply add more input fields and corresponding states, like 'newBookLink' and 'newBookImage' eg:

```
const AddBook = ({refreshBooks}) => {  
  const [modalOpen, setModalOpen] = useState(false);  
  const [newBookTitle, setNewBookTitle] = useState("");  
  const [newBookLink, setNewBookLink] = useState("");  
  const [newBookImage, setNewBookImage] = useState("");
```

CHAPTER 25: DELETING A BOOK

Next, we're going to implement the deletion of a book. We currently have a delete button beside the 'See in Amazon' link.



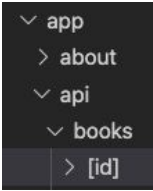
/app/components/Books.jsx

```
...  
    <div className="card-actions justify-end"> <Link href={book.link} className="btn btn-  
primary"> See in Amazon  
    </Link> <button className="btn btn-error"> Delete </button> </div>  
...
```

Delete Route Handler

Firstly, in the 'Books' route, we need a 'delete' route handler. However, unlike our previous endpoint, the 'delete' endpoint receives an id, eg. 'localhost:3000/api/books/1'. And we send a DELETE request to that endpoint.

To support this, we need to create a subfolder named 'id' in '/app/api/books'. This will hold the id of the book to be deleted.



In 'id', we need a 'route.js' file. This file will handle all the requests to this route. Fill in route.js with the following:

```
import books from '../data.json';
import {NextResponse} from 'next/server';

export const DELETE = async(request, {params}) =>{
  const id = params.id;

  console.log("id", id)

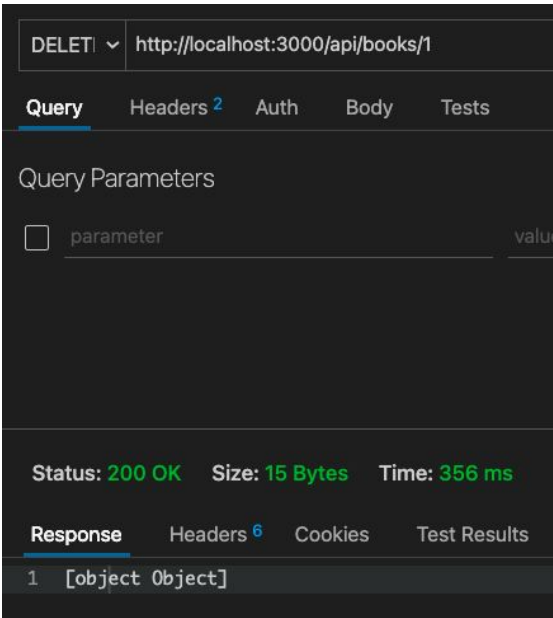
  return new NextResponse({"Book deleted": id}); }
```

Our 'delete' route handler will be asynchronous and receive a request and some params, which will contain the book id to be deleted.

We use 'console.log' first to make sure we can access the ID. After this, we return a new 'next response' saying 'book deleted' and the ID.

Testing our App

Now, using Thunder Client, if we send a new delete request to our local host, we should see the book deleted response.



Deleting from the Array

Let's now actually delete the book from the array. We'll see how to delete from an actual backend Prisma database later. Add in the codes in **bold**:

```
export const DELETE = async(request, {params}) =>{  
  const id = params.id;  
  
  const index = books.findIndex ( (book) => book.id === id)  
  if (index !== -1){  
    books.splice(index, 1); }  
  
  return new NextResponse({"Book deleted": id}); };
```

We get the index of the book we want to delete using 'books.findIndex' and loop through each book, checking if the book ID is equal to the ID we have specified.

If the index is not equal to -1 (meaning we have found the book to be deleted), we use 'book.splice' to remove it from the array.

Deleting from the Frontend

With the backend route in place, let's go to the Books component in the frontend and under the delete button, change in **bold**:

/app/components/Books.jsx

```
...
<div className="card-actions justify-end"> <Link href={book.link} className="btn btn-primary">
See in Amazon
  </Link>
  <button onClick={() => deleteBook(book.id)} className="btn btn-error"> Delete
</button>
</div>
...
```

We add a 'onClick' which calls a 'deleteBook' function with the book ID. Implement *deleteBook* by adding in **bold**:

```
...
const Books = () => {
  ...

  const fetchBooks = async () => {
    ...
  }

  useEffect(() => {
    fetchBooks();
  },[]);

  if(loading){ return <LoadingPage />}

  const handleSubmit = async (e) => {
    ...
  }

  const deleteBook = async(id) => {
const res = await fetch(`api/books/${id}`,{
method:'DELETE'
});
fetchBooks(); }
```

...
...

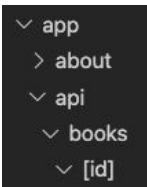
This function will send a 'delete' request to the API endpoint for that specific book ID. After deletion, we fetch the books again.

Running our App

On clicking 'delete' for a book, a 'delete' request is sent. However, all the books return upon refreshing since we have hardcoded it in the JSON file. To fix this, we need to replace the JSON file with an actual Prisma database, which we will cover in the next chapter.

Individual Dynamic Routes

I want to highlight that for individual or dynamic routes, like the one we have done here in 'books', we need to create a folder and name it as 'id' enclosed in square brackets.



This is a dynamic route where 'id' will be replaced by the specific book ID. To delete, view, or edit a specific book, we need to create dynamic routes like this.

CHAPTER 26: SETTING UP PRISMA DATABASE

So far, we've been accessing our data from the `data.json` file. Now we will be changing to a Prisma database to keep our data persistent.

Go to our terminal and run:

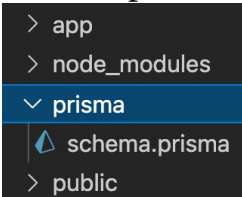
```
npm install prisma
```

We're using Prisma because it's relatively simple. Let's initialize Prisma with:

```
npx prisma init --datasource-provider sqlite
```

We're going to use SQLite (as an example) as it's the easiest to set up.

This initialization will generate the necessary files for us. We can see that we have a *prisma* folder in our code, and inside it, we have our *schema.prisma* file.



schema.prisma

```
// This is your Prisma schema file,  
// learn more about it in the docs: https://pris.ly/d/prisma-schema
```

```
generator client {  
  provider = "prisma-client-js"  
}
```



```
datasource db {  
  provider = "sqlite"  
  url = env("DATABASE_URL")  
}
```

It specifies that the provider is SQLite and the database URL is inside the .env file:

.env

```
DATABASE_URL="file:./dev.db"
```

We need to make sure that our .env files doesn't get committed to any git repository. So, let's go to .gitignore and make sure it already ignores local env files:

```
...  
# local env files  
.env*.local  
...
```

Next, we can work on our actual schema. In schema.prisma, add in bold:

```
// This is your Prisma schema file,  
// learn more about it in the docs: https://pris.ly/d/prisma-schema
```

```
generator client {  
  ...  
}
```

```
datasource db {  
  ...  
}
```

```
model Book{  
  id String @id @default(uuid())  
  title String  
  link String
```

```
img String
}
```

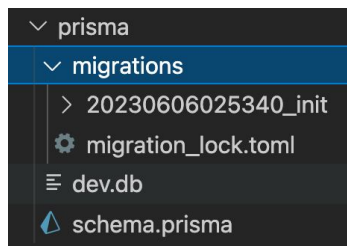
We have a model named 'Book' with all the properties as illustrated in our app.

At this point, we have a Prisma schema, but no database yet. In our Terminal, we need to run the command:

```
npx prisma migrate dev --name init
```

This command creates the 'Book' table based on our Book model, creates a new SQL migration file for this migration, and runs this SQL migration file against the database.

After running the command, our database is now in sync with our schema.

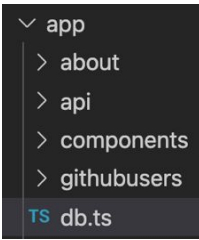


We can see a new 'migrations' folder under 'prisma', along with some .db files created for us. Since these .db files are for development, we should add to our *.gitignore*:

```
...
dev.db*
```

Using the Database in our Project

Next, to use the database inside our project, we create a new file under 'app' called db.ts.



Fill it with the following code:

```
import { PrismaClient } from '@prisma/client'
```

```
const globalForPrisma = global as unknown as {  
  prisma: PrismaClient | undefined  
}
```

```
export const prisma =  
  globalForPrisma.prisma ??  
  new PrismaClient({  
    log: ['query'],  
  })
```

```
if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma
```

We put this in a separate file because Next.js, in development mode, does hot reloading, which can cause issues with Prisma by constantly creating new connections with the Prisma client. This issue is known to Prisma (at time of creating this course). For more details, google for ‘prisma hot reload’, and click on the first link, you will see the following:

Best practice for instantiating PrismaClient with Next.js

Problem

Lots of users have come across this warning while working with [Next.js](#) in development:

```
warn(prisma-client) There are already 10 instances of Prisma Client active
```

There's a related [discussion](#) and [issue](#) for the same.

In development, the command `next dev` clears Node.js cache on run. This in turn initializes a new `PrismaClient` instance each time due to hot reloading that creates a connection to the database. This can quickly exhaust the database connections as each `PrismaClient` instance holds its own connection pool.

Prisma provides a solution (which is the same code we have pasted above):

Solution

The solution in this case is to instantiate a single instance `PrismaClient` and save it on the `globalThis` object. Then we keep a check to only instantiate `PrismaClient` if it's not on the `globalThis` object otherwise use the same instance again if already present to prevent instantiating extra `PrismaClient` instances.

`./db`

```
1 import { PrismaClient } from '@prisma/client'
2
3 const globalForPrisma = globalThis as unknown as {
4   prisma: PrismaClient | undefined
5 }
6
7 export const prisma = globalForPrisma.prisma ?? new PrismaClient()
8
9 if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma
```

Code Explanation

```
import { PrismaClient } from '@prisma/client'
```

In our `db.ts`, we import our client at the top

```
const globalForPrisma = global as unknown as {  
  prisma: PrismaClient | undefined  
}
```

Add our PrismaClient to the global object.

```
export const prisma =  
  globalForPrisma.prisma ??  
  new PrismaClient({  
    log: ['query'],  
  })
```

We create a *prisma* variable where we set it to either the global variable we've just created or create a brand new PrismaClient.

```
if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma
```

If we are not in production, we get Prisma from the global variable.

The global Prisma variable acts as a singleton, preventing us from having multiple Prisma client instances. This is particularly useful when we have the hot module loading environment.

With this, we have set up our Prisma database.

CHAPTER 27: GETTING BOOKS FROM PRISMA DATABASE

In our API routes, we will change getting our books from Prisma rather than from 'data.json'. Given that routes are hosted on the server, it's quite easy for us to access the Prisma database from the server end.

For instance, we can just call our database with *await prisma*. In `/app/api/books/route.js`, add:

```
//import books from './data.json'; import { NextResponse } from 'next/server';  
//import { v4 as uuidv4 } from 'uuid';  
import { prisma } from '../db';  
  
export async function GET(req) {  
  
    const books = await prisma.book.findMany(); console.log('GET books called');  
    return NextResponse.json(books);  
}
```

We import prisma from our db file. Then we can write *prisma.book.findMany()*. This command will retrieve all our books, and we assign this to a constant 'books'. Comment out the previous import books from *data.json* since we are no longer using it.

With React server components, we can access the database from there. This keeps our code secure on the server.

Adding some Dummy Books

However, our current database is empty, so we're going to add some dummy books into the database first. Add in **bold**:

```
export async function GET(req) {  
    // you can use your own dummy data  
    await prisma.book.create({data:{  
    title: 'Prisma Book',
```

```

    link:"https://www.amazon.com/dp/B0BXM RB5VF/", img:"
https://via.placeholder.com/600/92c952"
  })
}

```

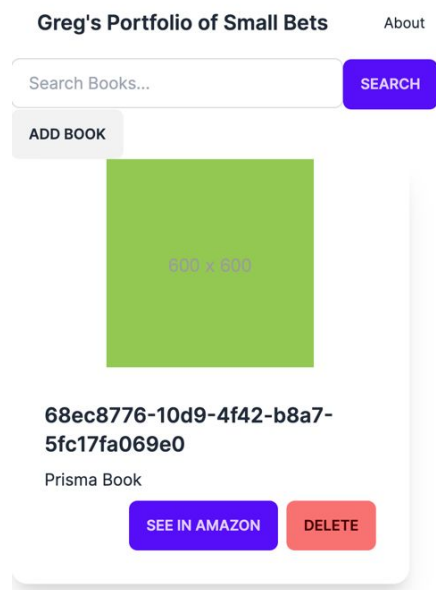
```

const books = await prisma.book.findMany();
console.log('GET books called');
return NextResponse.json(books);
}

```

We add books by writing *prisma.book.create()* and passing the data for our book object. For instance, we've added a title 'Prisma book', and specified a dummy link and image.

When we run our app, we successfully retrieve and display the book.



We can now get rid of the hardcoded.

```

export async function GET(req) {
  // you can use your own dummy data
  await prisma.book.create({data:{
    title: 'Prisma Book', link:"https://www.amazon.com/dp/B0BXM RB5VF/", img:"
https://via.placeholder.com/600/92c952"
  }}
}

```

```

    const books = await prisma.book.findMany();
    console.log('GET books called');
    return NextResponse.json(books);
}

```

We'll later implement the functionality to add new books.

CHAPTER 28: ADDING A BOOK INTO PRISMA

We have successfully retrieved our books from the Prisma database. Now, let's see how we can add a new book to our Prisma database as well. As we've done this earlier, it should be quite straightforward.

```

export async function GET(req) {
  // you can use your own dummy data
  await prisma.book.create({data:{
    title: 'Prisma Book', link: "https://www.amazon.com/dp/B0BXM RB5VF/", img: "
https://via.placeholder.com/600/92c952"
}})
  const books = await prisma.book.findMany();
  console.log('GET books called');
  return NextResponse.json(books);
}

```

We can just use the previous code from above and make the below changes in **bold** to our POST route:

```

import { NextResponse } from 'next/server';
//import { v4 as uuidv4 } from 'uuid'; import { prisma } from '../db';
...
...

export async function POST(req) {
  const { title, link, img } = await req.json();

  /*

```



```

const newBook = {
  id: uuidv4(), title, link, img };
*/
//books.push(newBook);
await prisma.book.create({data:{
  title: title,
  link:link,
  img:img
} })

return NextResponse.json('Book added successfully'); }

```

We won't be pushing it to our arrays any longer, so we won't need that line of code. Also, we won't need to create a new object here anymore because we can directly insert it into our Prisma database.

We also don't need the UUID anymore, as the Prisma database will create the UUID for us. In *schema.prisma*, we've specified that the ID will default to UUID:

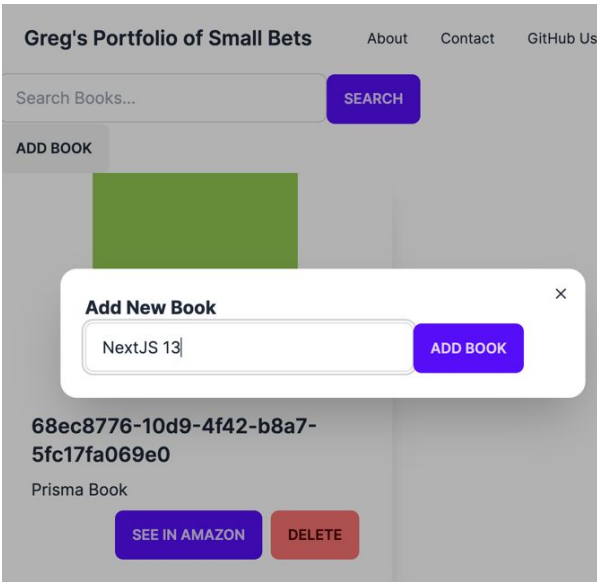
```

/app/prisma/schema.prisma
...
model Book{
  id String @id @default(uuid())
  title String
  link String
  img String
}

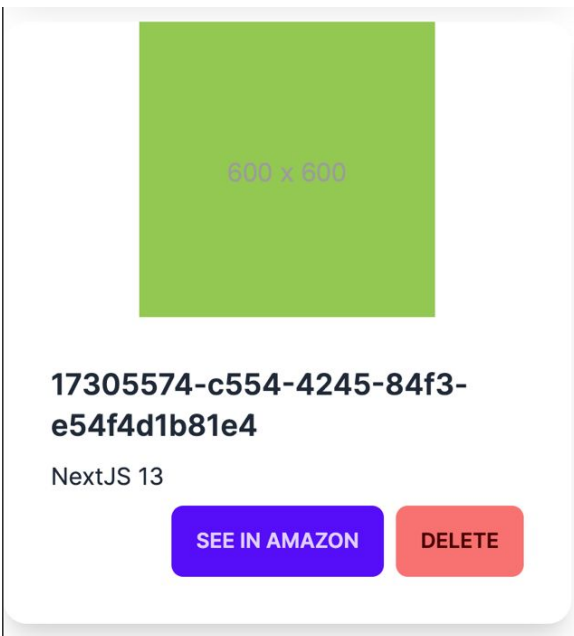
```

Running our App

Now, we can save this and go back to our page. Let's try adding something, say, a 'Next JS 13 book'.



As you can see below, our Next JS 13 book is added.



Although the book cover is hardcoded, you could specify our own book cover link.

If we examine the terminal, you can see that Prisma generates the SQL 'INSERT' statement for us.

```

prisma:query SELECT 1
prisma:query SELECT `main`.`Book`.`id`, `main`.`Book`.`title`, `main`.`Book`.`link`, `main`.`Book`.`img` FROM `main`.`Book` WHERE 1=1 LIMIT ? OFFSET ?
GET books called
prisma:query BEGIN
prisma:query INSERT INTO `main`.`Book` (`id`, `title`, `link`, `img`) VALUES (?, ?, ?, ?) RETURNING `id` AS `id`
prisma:query SELECT `main`.`Book`.`id`, `main`.`Book`.`title`, `main`.`Book`.`link`, `main`.`Book`.`img` FROM `main`.`Book` WHERE `main`.`Book`.`id` = ? LIMIT ? OFFSET ?
prisma:query COMMIT
prisma:query SELECT `main`.`Book`.`id`, `main`.`Book`.`title`, `main`.`Book`.`link`, `main`.`Book`.`img` FROM `main`.`Book` WHERE 1=1 LIMIT ? OFFSET ?
GET books called

```

Prisma converts transactions into SQL statements and runs them on the server to perform 'INSERT' or 'SELECT' operations.

Everything seems to be working fine. The book is added as expected.

CHAPTER 29: DELETING FROM PRISMA

Let's see how we can implement our 'delete' functionality.

If you remember, our dynamic id 'delete' route handler is located in '/app/api/books/[id]/route.js' which has the following code:

```
...
export const DELETE = async(request, {params}) =>{
  const id = params.id;

  const index = books.findIndex ( (book) => book.id === id)
  if (index !== -1){
    books.splice(index, 1);
  }

  return new NextResponse({"Book deleted": id}); }
```

Make the following changes in **bold**:

```
export const DELETE = async(request, {params}) =>{
  const id = params.id;

  const index = books.findIndex ( (book) => book.id === id)
if (index !== -1){
  books.splice(index, 1); }

  await prisma.book.delete({where: {id:id}})
  return new NextResponse({"Book deleted": id}); }
```

We remove the lines of code (where we delete from the array) as we'll delete from Prisma instead. We have to specify the selection criteria in 'where', then 'ID' is equal to our ID.

Save this and go back to our application. Let's try removing a book. If I click 'remove', it gets deleted successfully.

If you look at the terminal, you'll notice that it generates a 'delete' query:

```
prisma:query DELETE FROM `main`.`Book` WHERE (`main`.`Book`.`id` IN (?) AND (`main`.`Book`.`id` = ? AND 1=1))
```

We have successfully implemented our 'delete' functionality from a Prisma database.

CHAPTER 30: SEARCHING IN PRISMA

Our search function is currently still querying from the JSON array (we are using 'books.filter'):

/app/api/books/search/route.js ...

```
export async function GET(req) {  
  const { searchParams } = new URL(req.url); const query = searchParams.get('query')  
  const filteredBooks = books.filter((book) => {  
    return book.title.toLowerCase().includes(query.toLowerCase());  
  });  
  return NextResponse.json(filteredBooks);  
}
```

Let's change this to retrieve data from Prisma with the below changes in **bold**:

```
import { NextResponse } from "next/server";  
//import books from "../data.json"; import { prisma } from '../..db';
```

```
export async function GET(req) {  
  const { searchParams } = new URL(req.url);  
  const query = searchParams.get("query");  
  
  /*  
  const filteredBooks = books.filter((book) => {  
  return book.title.toLowerCase().includes(query.toLowerCase());  
  }  
  */  
  
  const filteredBooks = await prisma.book.findMany({  
    where:{  
      title:{  
        contains: query  
      }  
    }  
  })  
  
  return NextResponse.json(filteredBooks);  
}
```

Code Explanation

```
import { NextResponse } from "next/server";  
//import books from "../data.json"; import { prisma } from '../../db';
```

We import Prisma. Because we are accessing three levels down now, we need to adjust our import statement. We can also remove the import for 'books' from 'data.json'.

```
const filteredBooks = await prisma.book.findMany({
```

We use 'await Prisma.book.findMany' and specify the selection criteria within an object.

```
const filteredBooks = await prisma.book.findMany({  
  where:{  
    title:{  
      contains: query  
    }  
  }  
})
```

Within 'where', we indicate where 'title' contains the 'query'.

If you run the app and do a search, the search function will now retrieve data directly from our Prisma database.

Final Words

We have gone through quite a lot of content to equip you with the skills to create a NextJS full stack app.

Hopefully, you have enjoyed this book and would like to learn more from me. I would love to get your feedback, learning what you liked and didn't for us to improve.

Please feel free to email me at support@i-ducate.com to get updated versions of this book.

If you didn't like the book, or if you feel that I should have covered certain additional topics, please email us to let us know. This book can only get better thanks to readers like you.

If you like the book, I would appreciate if you could leave us a review too. Thank you and all the best for your learning journey in NextJS development!

ABOUT THE AUTHOR

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing.

Contact Greg at support@i-ducate.com