Adding a React Frontend to Complete MERN

A web application is incomplete without a frontend. It is the part that users interact with and it is crucial to any web experience. In this chapter, we will use React to add an interactive user interface to the basic user and auth features that have been implemented for the backend of the MERN skeleton application, which we started building in the previous chapter. This functional frontend will add React components that connect to the backend API and allow users to navigate seamlessly within the application based on authorization. By the end of this chapter, you will have learned how to easily integrate a React client-side with a Node-Express-MongoDB server-side to make a full-stack web application.

In this chapter, we will cover the following topics:

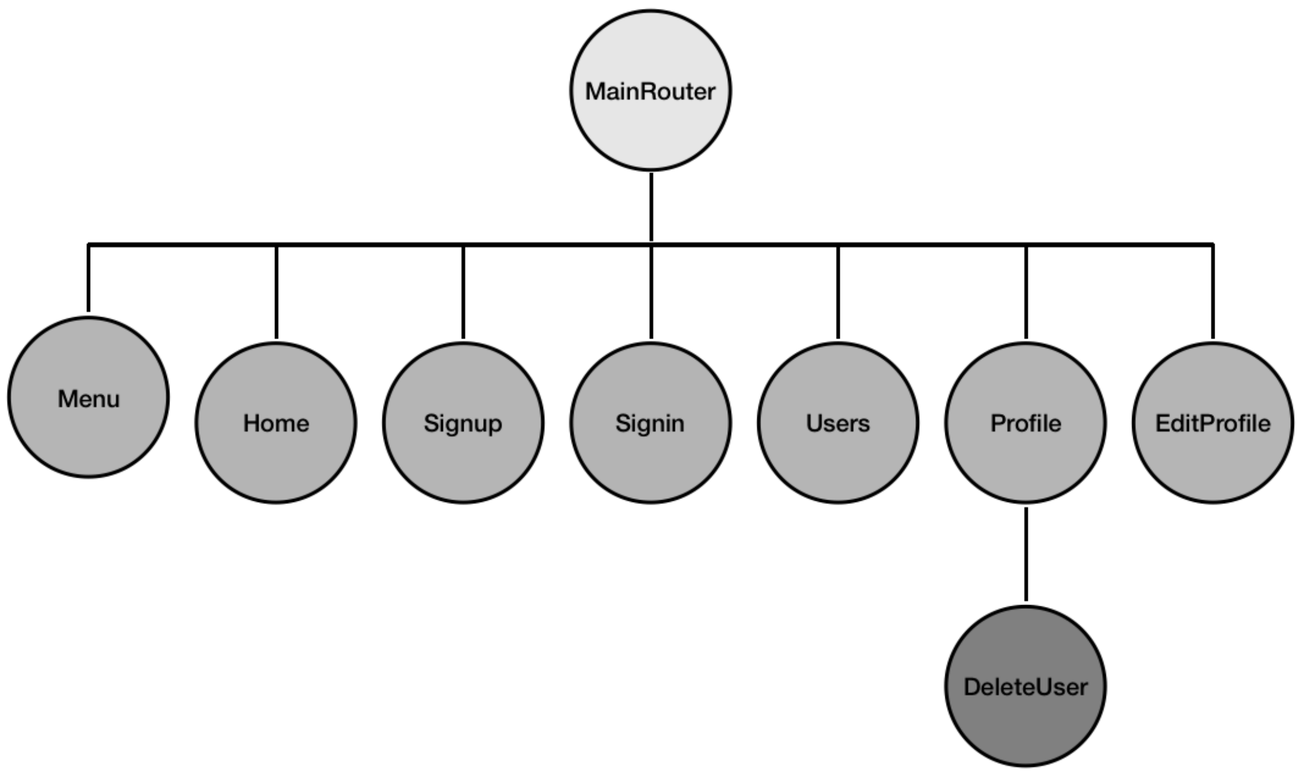
- Frontend features of the skeleton
- Setting up development with React, React Router, and Material-UI
- Rendering a home page built with React
- Backend user API integration
- Auth integration for restricted access
- User list, profile, edit, delete, sign up, and sign in UI to complete the user frontend
- Basic server-side rendering

Defining the skeleton application frontend

In order to fully implement the skeleton application features we discussed in the *Feature breakdown* section of Chapter 3, *Building a Backend with MongoDB, Express, and Node*, we will add the following user interface components to our base application:

- **Home page**: A view that renders at the root URL to welcome users to the web application.
- **Sign-up page**: A view with a form for user sign-up, allowing new users to create a user account and redirecting them to a sign-in page when successfully created.
- **Sign-in page**: A view with a sign-in form that allows existing users to sign in so they have access to protected views and actions.
- **User list page**: A view that fetches and shows a list of all the users in the database, and also links to individual user profiles.
- **Profile page**: A component that fetches and displays an individual user's information. This is only accessible by signed-in users and also contains edit and delete options, which are only visible if the signed-in user is looking at their own profile.
- **Edit profile page**: A form that fetches the user's information to prefill the form fields. This allows the user to edit the information and this form is accessible only if the logged-in user is trying to edit their own profile.
- **Delete user component**: An option that allows the signed-in user to delete their own profile after confirming their intent.
- **Menu navigation bar**: A component that lists all the available and relevant views to the user, and also helps to indicate the user's current location in the application.

The following React component tree diagram shows all the React components we will develop to build out the views for this base application:



MainRouter will be the main React component. This contains all the other custom React views in the application. **Home**, **Signup**, **Signin**, **Users**, **Profile**, and **EditProfile** will render at individual routes declared with React Router, whereas the **Menu** component will render across all these views. **DeleteUser** will be a part of the **Profile** view.

> *The code discussed in this chapter, as well as the complete skeleton, is available on GitHub at https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter03%20and%2004/mern-skeleton. You can clone this code and run the application as you go through the code explanations in the rest of this chapter.*

In order to implement these frontend React views, we will have to extend the existing project code, which contains the standalone server application for the MERN skeleton. Next, we'll take a brief look at the files that will make up this frontend and that are needed to complete the full-stack skeleton application code.

Folder and file structure

The following folder structure shows the new folders and files to be added to the skeleton project we started implementing in the previous chapter, in order to complete it with a React frontend:

```
| mern_skeleton/
   | -- client/
      | --- assets/
         | ---- images/
      | --- auth/
         | ---- api-auth.js
         | ---- auth-helper.js
         | ---- PrivateRoute.js
         | ---- Signin.js
      | --- core/
         | ---- Home.js
         | ---- Menu.js
      | --- user/
         | ---- api-user.js
         | ---- DeleteUser.js
         | ---- EditProfile.js
         | ---- Profile.js
         | ---- Signup.js
         | ---- Users.js
      | --- App.js
      | --- main.js
      | --- MainRouter.js
      | --- theme.js
   | -- server/
      | --- devBundle.js
   | -- webpack.config.client.js
   | -- webpack.config.client.production.js
```

The `client` folder will contain the React components, helpers, and frontend assets, such as images and CSS. Besides this folder and the Webpack configuration files for compiling and bundling the client code, we will also modify some of the other existing files to finish up the integration of the complete skeleton application in this chapter.

Before we start implementing the specific frontend features, we need to get set up for React development by installing the necessary modules and adding configuration to compile, bundle, and load the React views. We will go through these setup steps in the next section.

Setting up for React development

Before we can start developing with React in our existing skeleton codebase, we need to add configuration to compile and bundle the frontend code, add the React-related dependencies that are necessary to build the interactive interface, and tie this all together in the MERN development flow.

To achieve this, we will add frontend configuration for Babel, Webpack, and React Hot Loader to compile, bundle, and hot reload the code. Next, we will modify the server code to initiate code bundling for both the frontend and backend in one command to make the development flow simple. Then, we will update the code further so that it serves the bundled code from the server when the application runs in the browser. Finally, we will finish setting up by installing the React dependencies that are necessary to start implementing the frontend.

Configuring Babel and Webpack

It is necessary to compile and bundle the React code that we will write to implement the frontend before the code can run in browsers. To compile and bundle the client code so that we can run it during development and also bundle it for production, we will update the configuration for Babel and Webpack. Then, we will configure the Express app to initiate frontend and backend code bundling in one command, so that just starting the server during development gets the complete stack ready for running and testing.

Babel

To compile React, first, install the Babel React preset module as a development dependency by running the following command from the command line:

```
yarn add --dev @babel/preset-react
```

Then, update `.babelrc` with the following code. This will include the module and also configure the `react-hot-loader` Babel plugin as required for the `react-hot-loader` module.

`mern-skeleton/.babelrc`:

```
{
    "presets": [
      ["@babel/preset-env",
        {
          "targets": {
            "node": "current"
          }
        }
      ],
      "@babel/preset-react"
    ],
    "plugins": [
      "react-hot-loader/babel"
    ]
}
```

To put this updated Babel configuration to use, we need to update the Webpack configuration, which we will look at in the next section.

Webpack

To bundle client-side code after compiling it with Babel, and also to enable `react-hot-loader` for faster development, install the following modules by running these commands from the command line:

```
yarn add -dev webpack-dev-middleware webpack-hot-middleware fil
yarn add react-hot-loader @hot-loader/react-dom
```

Then, to configure Webpack for frontend development and to build the production bundle, we will add a `webpack.config.client.js` file and a `webpack.config.client.production.js` file with the same configuration code we described in Chapter 2, *Preparing the Development Environment*.

With Webpack configured and ready for bundling the frontend React code, next, we will add some code that we can use in our development flow. This will make the full-stack development process seamless.

Loading Webpack middleware for development

During development, when we run the server, the Express app should also load the Webpack middleware that's relevant to the frontend with respect to the configuration that's been set for the client-side code, so that the frontend and backend development workflow is integrated. To enable this, we will use the `devBundle.js` file we discussed in Chapter 2, *Preparing the Development Environment*, in order to set up a `compile` method that takes the Express app and configures it to use the Webpack middleware. The `devBundle.js` file in the `server` folder will look as follows.

`mern-skeleton/server/devBundle.js`:

```
import config from './../config/config'
import webpack from 'webpack'
import webpackMiddleware from 'webpack-dev-middleware'
import webpackHotMiddleware from 'webpack-hot-middleware'
import webpackConfig from './../webpack.config.client.js'

const compile = (app) => {
  if(config.env === "development"){
    const compiler = webpack(webpackConfig)
    const middleware = webpackMiddleware(compiler, {
      publicPath: webpackConfig.output.publicPath
    })
    app.use(middleware)
    app.use(webpackHotMiddleware(compiler))
  }
}

export default {
  compile
}
```

In this method, the Webpack middleware uses the values set in `webpack.config.client.js`, and we enable hot reloading from the server-side using Webpack Hot Middleware.

Finally, we need to import and call this `compile` method in `express.js` by adding the following highlighted lines, but only during development.

`mern-skeleton/server/express.js`:

```
import devBundle from './devBundle'
const app = express()
devBundle.compile(app)
```

These two highlighted lines are only meant for development mode and should be commented out when building the code for production. When the Express app runs in development mode, adding this code will import the middleware, along with the client-side Webpack configuration. Then, it will initiate Webpack to compile and bundle the client-side code and also enable hot reloading.

The bundled code will be placed in the `dist` folder. This code will be needed to render the views. Next, we will configure the Express server app so that it serves the static files from this `dist` folder. This will ensure that the bundled React code can be loaded in the browser.

Loading bundled frontend code

The frontend views that we will see rendered in the browser will load from the bundled files in the `dist` folder. For it to be possible to add these bundled files to the HTML view containing our frontend, we need to configure the Express app so that it serves static files, which are files that aren't generated dynamically by server-side code.

Serving static files with Express

To ensure that the Express server properly handles the requests to static files such as CSS files, images, or the bundled client-side JS, we will configure it so that it serves static files from the `dist` folder by adding the following configuration in `express.js`.

`mern-skeleton/server/express.js`:

```
import path from 'path'
const CURRENT_WORKING_DIR = process.cwd()
app.use('/dist', express.static(path.join(CURRENT_WORKING_DIR,
```

With this configuration in place, when the Express app receives a request at a route starting with `/dist`, it will know to look for the requested static resource in the `dist` folder before returning the resource in the response. Now, we can load the bundled files from the `dist` folder in the frontend.

Updating the template to load a bundled script

To add the bundled frontend code in the HTML to render our React frontend, we will update the `template.js` file so that it adds the script file from the `dist` folder to the end of the `<body>` tag.

`mern-skeleton/template.js`:

```
...
<body>
    <div id="root"></div>
    <script type="text/javascript" src="/dist/bundle.js"></scri
</body>
```

This script tag will load our React frontend code in the browser when we visit the root URL `'/'` with the server running. We are ready to see this in action and can start installing the dependencies that will add the React views.

Adding React dependencies

The frontend views in our skeleton application will primarily be implemented using React. In addition, to enable client-side routing, we will use React Router, and to enhance the user experience with a sleek look and feel, we will use Material-UI. To add these libraries, we will install the following modules in this section:

- **Core React modules**: `react` and `react-dom`
- **React Router modules**: `react-router` and `react-router-dom`
- **Material-UI modules**: `@material-ui/core` and `@material-ui/icons`

React

Throughout this book, we will use React to code up the frontend. To start writing the React component code, we will need to install the following modules as regular dependencies:

```
yarn add react react-dom
```

These are the core React library modules that are necessary for implementing the React-based web frontend. With other additional modules, we will add more functionality on top of React.

React Router

React Router provides a collection of navigational components that enable routing on the frontend for React applications. We will add the following React Router modules:

```
yarn add react-router react-router-dom
```

These modules will let us utilize declarative routing and have bookmarkable URL routes in the frontend.

Material-UI

In order to keep the UI in our MERN applications sleek without delving too much into UI design and implementation, we will utilize the Material-UI library. It provides ready to use and customizable React components that implement Google's material design. To start using Material-UI components to make the frontend, we need to install the following modules:

```
yarn add @material-ui/core @material-ui/icons
```

At the time of writing, the latest version of Material-UI is `4.9.8`*. It is recommended that you install this exact version in order to ensure the code for the example projects does not break.*

To add the `Roboto` fonts that are recommended by Material-UI and to use the `Material-UI` icons, we will add the relevant style links into the `template.js` file, in the HTML document's `<head>` section:

```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?f
<link href="https://fonts.googleapis.com/icon?family=Material+I
```

With the development configuration all set up and the necessary React modules added to the code base, we can now implement the custom React components, starting with a home page. This should load up as the first view of the complete application.

Rendering a home page view

To demonstrate how to implement a functional frontend for this MERN skeleton, we will start by detailing how to render a simple home page at the root route of the application, before covering backend API integration, user auth integration, and implementing the other view components in the rest of this chapter.

The process of implementing and rendering a working `Home` component at the root route will also expose the basic structure of the frontend code in the skeleton. We will start with the top-level entry component that houses the whole React app and renders the main router component, which links all the React components in the application.

In the following sections, we will begin implementing the React frontend. First, we will add the root React component, which is integrated with React Router and Material-UI and configured for hot reloading. We will also learn how to customize the Material-UI theme and make the theme available to all our components. Finally, we will implement and load the React component representing the home page, in turn demonstrating how to add and render React views in this application.

Entry point at main.js

The `client/main.js` file in the client folder will be the entry point to render the complete React app, as already indicated in the client-side Webpack configuration object. In `client/main.js`, we import the root or top-level React component that will contain the whole frontend and render it to the `div` element with the `'root'` ID specified in the HTML document in `template.js`.

`mern-skeleton/client/main.js`:

```
import React from 'react'
import { render } from 'react-dom'
import App from './App'

render(<App/>, document.getElementById('root'))
```

Here, the top-level root React component is the App component and it is being rendered in the HTML. The App component is defined in `client/App.js`, as discussed in the next subsection.

Root React component

The top-level React component that will contain all the components for the application's frontend is defined in the `client/App.js` file. In this file, we configure the React app so that it renders the view components with a customized Material-UI theme, enables frontend routing, and ensures that the React Hot Loader can instantly load changes as we develop the components.

In the following sections, we will add code to customize the theme, make this theme and React Router capabilities available to our React components, and configure the root component for hot reloading.

Customizing the Material-UI theme

The Material-UI theme can be easily customized using the `ThemeProvider` component. It can also be used to configure the custom values of theme variables in `createMuiTheme()`. We will define a custom theme for the skeleton application in `client/theme.js` using `createMuiTheme`, and then export it so that it can be used in the App component.

`mern-skeleton/client/theme.js`:

```
import { createMuiTheme } from '@material-ui/core/styles'
import { pink } from '@material-ui/core/colors'

const theme = createMuiTheme({
    typography: {
      useNextVariants: true,
    },
    palette: {
      primary: {
      light: '#5c67a3',
      main: '#3f4771',
      dark: '#2e355b',
      contrastText: '#fff',
    },
    secondary: {
      light: '#ff79b0',
      main: '#ff4081',
      dark: '#c60055',
      contrastText: '#000',
    },
      openTitle: '#3f4771',
      protectedTitle: pink['400'],
      type: 'light'
    }
})

export default theme
```

For the skeleton, we only apply minimal customization by setting some color values to be used in the UI. The theme variables that are generated here will be passed to, and available in, all the components we build.

Wrapping the root component with ThemeProvider and BrowserRouter

The custom React components that we will create to make up the user interface will be accessed with the frontend routes specified in the `MainRouter` component. Essentially, this component houses all the custom views that have been developed for the application and needs to be given the theme values and routing features. This component will be our core component in the root `App` component, which is defined in the following code.

`mern-skeleton/client/App.js`:

```
import React from 'react'
import MainRouter from './MainRouter'
import {BrowserRouter} from 'react-router-dom'
import { ThemeProvider } from '@material-ui/styles'
import theme from './theme'

const App = () => {
  return (
    <BrowserRouter>
      <ThemeProvider theme={theme}>
        <MainRouter/>
      </ThemeProvider>
    </BrowserRouter>
)}
```

When defining this root component in `App.js`, we wrap the `MainRouter` component with `ThemeProvider`, which gives it access to the Material-UI theme, and `BrowserRouter`, which enables frontend routing with React Router. The custom theme variables we defined previously are passed as a prop to `ThemeProvider`, making the theme available in all our custom React components. Finally, in the `App.js` file, we need to export this `App` component so that it can be imported and used in `main.js`.

Marking the root component as hot-exported

The last line of code in `App.js`, which exports the `App` component, uses the **higher-order component** (**HOC**) `hot` module from `react-hot-loader` to mark the root component as `hot`.

`mern-skeleton/client/App.js`:

```
import { hot } from 'react-hot-loader'
const App = () => { ... }
export default hot(module)(App)
```

Marking the `App` component as `hot` in this way essentially enables live reloading of our React components during development.

For our MERN applications, we won't have to change the `main.js` and `App.js` code all that much after this point, and we can continue building out the rest of the React app by injecting new components into the `MainRouter` component, which is what we'll do in the next section.

Marking the root component as hot-exported

The last line of code in `App.js`, which exports the `App` component, uses the **higher-order component** (**HOC**) `hot` module from `react-hot-loader` to mark the root component as `hot`.

`mern-skeleton/client/App.js`:

```
import { hot } from 'react-hot-loader'
const App = () => { ... }
export default hot(module)(App)
```

Marking the `App` component as `hot` in this way essentially enables live reloading of our React components during development.

For our MERN applications, we won't have to change the `main.js` and `App.js` code all that much after this point, and we can continue building out the rest of the React app by injecting new components into the `MainRouter` component, which is what we'll do in the next section.

Adding a home route to MainRouter

The `MainRouter.js` code will help render our custom React components with respect to the routes or locations in the application. In this first version, we will only add the root route for rendering the `Home` component.

`mern-skeleton/client/MainRouter.js`:

```
import React from 'react'
import {Route, Switch} from 'react-router-dom'
import Home from './core/Home'
const MainRouter = () => {
    return ( <div>
      <Switch>
        <Route exact path="/" component={Home}/>
      </Switch>
     </div>
    )
}
export default MainRouter
```

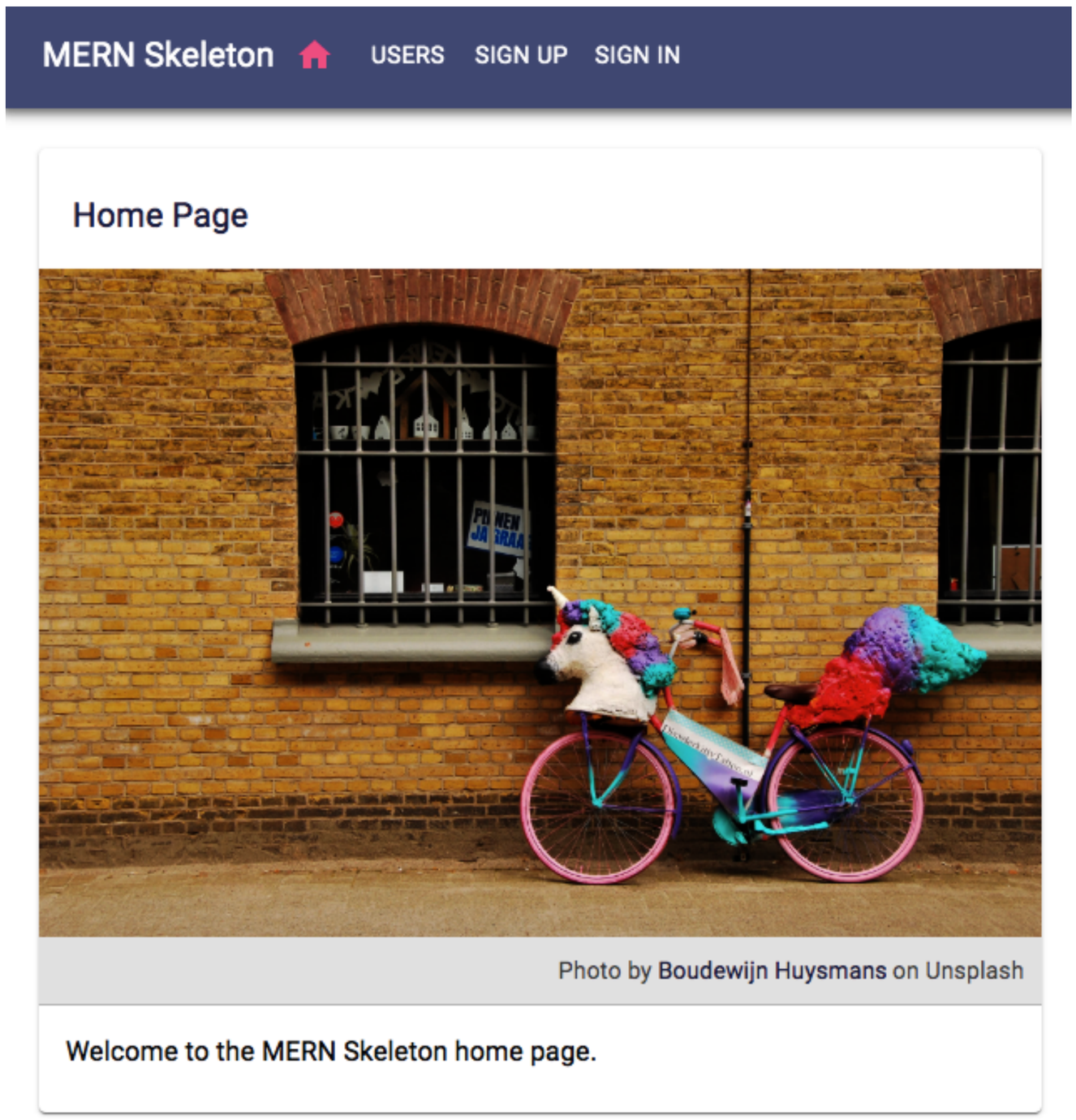
As we develop more view components, we will update the `MainRouter` and add routes for the new components inside the `Switch` component.

The `Switch` component in React Router renders a route exclusively. In other words, it only renders the first child that matches the requested route path. On the other hand, without being nested in a `Switch`, every `Route` component renders inclusively when there is a path match; for example, a request at `'/'` also matches a route at `'/contact'`.
The `Home` component, which we added this route for in `MainRouter`, needs to be defined and exported, which we'll do in the next section.

The Home component

The `Home` component will be the React component containing the home page view of the skeleton application. It will be rendered in the browser when the user visits the root route, and we will compose it with Material-UI components.

The following screenshot shows the `Home` component, as well as the `Menu` component, which will be implemented later in this chapter as an individual component that provides navigation across the application:



The `Home` component and other view components that will be rendered in the browser for the user to interact with will follow a common code structure that contains the following parts in the given order:

- Imports of libraries, modules, and files needed to construct the component
- Style declarations to define the specific CSS styles for the component elements
- A function that defines the React component

Throughout this book, as we develop new React components representing the frontend views, we will focus mainly on the React component definition part. But for our first implementation, we will elaborate on all these parts to introduce the necessary structure.

Imports

For each React component implementation, we need to import the libraries, modules, and files being used in the implementation code. The component file will start with imports from React, Material-UI, React Router modules, images, CSS, API fetch, and the auth helpers from our code, as required by the specific component. For example, for the `Home` component code in `Home.js`, we use the following imports.

`mern-skeleton/client/core/Home.js`:

```
import React from 'react'
import { makeStyles } from '@material-ui/core/styles'
import Card from '@material-ui/core/Card'
import CardContent from '@material-ui/core/CardContent'
import CardMedia from '@material-ui/core/CardMedia'
import Typography from '@material-ui/core/Typography'
import unicornbikeImg from './../assets/images/unicornbike.jpg'
```

The image file is kept in the `client/assets/images/` folder and is imported so that it can be added to the `Home` component. These imports will help us build the component and also define the styles to be used in the component.

Style declarations

After the imports, we will define the CSS styles that are required to style the elements in the component by utilizing the `Material-UI` theme variables and `makeStyles`, which is a custom React hook API provided by `Material-UI`.

Hooks are new to React. Hooks are functions that make it possible to use React state and life cycle features in function components, without having to write a class to define the component. React provides some built-in hooks, but we can also build custom hooks as needed to reuse stateful behavior across different components. To learn more about React Hooks, visit reactjs.org/docs/hooks-intro.html.

For the `Home` component in `Home.js`, we have the following styles.

`mern-skeleton/client/core/Home.js`:

```
const useStyles = makeStyles(theme => ({
  card: {
    maxWidth: 600,
    margin: 'auto',
    marginTop: theme.spacing(5)
  },
  title: {
    padding:`${theme.spacing(3)}px ${theme.spacing(2.5)}px ${th
    color: theme.palette.openTitle
  },
  media: {
    minHeight: 400
  }
}))
```

The JSS style objects defined here will be injected into the component using the hook returned by `makeStyles`. The `makeStyles` hook API takes a function as an argument and gives access to our custom theme variables, which we can use when defining the styles.

> *Material-UI uses JSS, which is a CSS-in-JS styling solution for adding styles to components. JSS uses JavaScript as a language to describe styles. This book will not cover CSS and styling implementations in detail. It will mostly rely on the default look and feel of Material-UI components. To learn more about JSS, visit* http://cssinjs.org/?v=v9.8.1. *For examples of how to customize the* `Material-UI` *component styles, check out the Material-UI documentation at* https://material-ui.com/.

We can use these generated styles to style the elements in the component, as shown in the following `Home` component definition.

Component definition

While writing the function to define the component, we will compose the content and behavior of the component. The `Home` component will contain a Material-UI `Card` with a headline, an image, and a caption, all styled with the styles we defined previously and returned by calling the `useStyles()` hook.

`mern-skeleton/client/core/Home.js`:

```
export default function Home(){
  const classes = useStyles()
    return (
     <Card className={classes.card}>
        <Typography variant="h6" className={classes.title}>
          Home Page
        </Typography>
        <CardMedia className={classes.media}
                   image={unicornbikeImg} title="Unicorn Bicycl
        <CardContent>
          <Typography variant="body2" component="p">
            Welcome to the MERN Skeleton home page.
          </Typography>
        </CardContent>
      </Card>
    )
}
```

In the preceding code, we defined and exported a function component named `Home`. The exported component can now be used for composition within other components. We already imported this `Home` component in a route in the `MainRouter` component, as we discussed earlier.

Throughout this book, we will define all our React components as functional components. We will utilize React Hooks, which is a new addition to React, to add state and life cycle features, instead of using class definitions to achieve the same.

The other view components to be implemented in our MERN applications will adhere to the same structure. In the rest of this book, we will focus mainly on the component definition, highlighting the unique aspects of the implemented component.

We are almost ready to run this code to render the home page component in the frontend. But before that, we need to update the Webpack configurations so that we can bundle and display images.

Bundling image assets

The static image file that we imported into the `Home` component view must also be included in the bundle with the rest of the compiled JS code so that the code can access and load it. To enable this, we need to update the Webpack configuration files and add a module rule to load, bundle, and emit image files to the `dist` output directory, which contains the compiled frontend and backend code.

Update the `webpack.config.client.js`, `webpack.config.server.js`, and `webpack.config.client.production.js` files so that you can add the following module rule after the use of `babel-loader`:

```
[ …
    {
       test: /\.(ttf|eot|svg|gif|jpg|png)(\?[\s\S]+)?$/,
       use: 'file-loader'
    }
]
```

This module rule uses the `file-loader` node module for Webpack, which needs to be installed as a development dependency, as follows:

```
yarn add --dev file-loader
```

With this image bundling configuration added, the home page component should successfully render the image when we run the application.

Running and opening in the browser

The client code up to this point can be run so that we can view the `Home` component in the browser at the root URL. To run the application, use the following command:

```
yarn development
```

Then, open the root URL (`http://localhost:3000`) in the browser to see the `Home` component.

The `Home` component we've developed in this section is a basic view component without interactive features and does not require the use of the backend APIs for user CRUD or auth. However, the remaining view components for our skeleton frontend will need the backend APIs and auth, so we will look at how to integrate these in the next section.

Integrating backend APIs

Users should be able to use the frontend views to fetch and modify user data in the database based on authentication and authorization. To implement these functionalities, the React components will access the API endpoints that are exposed by the backend using the Fetch API.

The Fetch API is a newer standard that makes network requests similar to **XMLHttpRequest** *(***XHR***) but using promises instead, enabling a simpler and cleaner API. To learn more about the Fetch API, visit* https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

Fetch for user CRUD

In the `client/user/api-user.js` file, we will add methods for accessing each of the user CRUD API endpoints, which the React components can use to exchange user data with the server and database as required. In the following sections, we will look at the implementation of these methods and how they correspond to each CRUD endpoint.

Creating a user

The `create` method will take user data from the view component, which is where we will invoke this method. Then, it will use `fetch` to make a `POST` call at the create API route, `'/api/users'`, to create a new user in the backend with the provided data.

`mern-skeleton/client/user/api-user.js`:

```
const create = async (user) => {
  try {
      let response = await fetch('/api/users/', {
        method: 'POST',
        headers: {
          'Accept': 'application/json',
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(user)
      })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

Finally, in this method, we return the response from the server as a promise. So, the component calling this method can use this promise to handle the response appropriately, depending on what is returned from the server. Similarly, we will implement the `list` method next.

Listing users

The `list` method will use fetch to make a `GET` call to retrieve all the users in the database, and then return the response from the server as a promise to the component.

`mern-skeleton/client/user/api-user.js`:

```
const list = async (signal) => {
  try {
    let response = await fetch('/api/users/', {
      method: 'GET',
      signal: signal,
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

The returned promise, if it resolves successfully, will give the component an array containing the user objects that were retrieved from the database. In the case of a single user read, we will deal with a single user object instead, as demonstrated next.

Reading a user profile

The `read` method will use fetch to make a `GET` call to retrieve a specific user by ID. Since this is a protected route, besides passing the user ID as a parameter, the requesting component must also provide valid credentials, which, in this case, will be a valid JWT received after a successful sign-in.

`mern-skeleton/client/user/api-user.js`:

```
const read = async (params, credentials, signal) => {
  try {
    let response = await fetch('/api/users/' + params.userId, {
      method: 'GET',
      signal: signal,
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      }
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

The JWT is attached to the `GET` fetch call in the `Authorization` header using the `Bearer` scheme, and then the response from the server is returned to the component in a promise. This promise, when it resolves, will either give the component the user details for the specific user or notify that access is restricted to authenticated users. Similarly, the updated user API method also needs to be passed valid JWT credentials for the fetch call, as shown in the next section.

Updating a user's data

The `update` method will take changed user data from the view component for a specific user, then use `fetch` to make a PUT call to update the existing user in the backend. This is also a protected route that will require a valid JWT as the credential.

`mern-skeleton/client/user/api-user.js`:

```
const update = async (params, credentials, user) => {
  try {
    let response = await fetch('/api/users/' + params.userId, {
      method: 'PUT',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      },
      body: JSON.stringify(user)
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

As we have seen with the other fetch calls, this method will also return a promise containing the server's response to the user update request. In the final method, we will learn how to call the user delete API.

Deleting a user

The `remove` method will allow the view component to delete a specific user from the database and use fetch to make a `DELETE` call. This, again, is a protected route that will require a valid JWT as a credential, similar to the `read` and `update` methods.

`mern-skeleton/client/user/api-user.js`:

```
const remove = async (params, credentials) => {
  try {
    let response = await fetch('/api/users/' + params.userId, {
      method: 'DELETE',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      }
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

The response from the server to the delete request will be returned to the component as a promise, as in the other methods.

In these five helper methods, we have covered calls to all the user CRUD-related API endpoints that we implemented on the backend. Finally, we can export these methods from the `api-user.js` file as follows.

`mern-skeleton/client/user/api-user.js`:

```
export { create, list, read, update, remove }
```

These user CRUD methods can now be imported and used by the React components as required. Next, we will implement similar helper methods to integrate the auth-related API endpoints.

Fetch for the auth API

In order to integrate the auth API endpoints from the server with the frontend React components, we will add methods for fetching sign-in and sign-out API endpoints in the `client/auth/api-auth.js` file. Let's take a look at them.

Sign-in

The `signin` method will take user sign-in data from the view component, then use `fetch` to make a `POST` call to verify the user with the backend.

`mern-skeleton/client/auth/api-auth.js`:

```
const signin = async (user) => {
  try {
    let response = await fetch('/auth/signin/', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      credentials: 'include',
      body: JSON.stringify(user)
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

The response from the server will be returned to the component in a promise, which may provide the JWT if sign-in was successful. The component invoking this method needs to handle the response appropriately, such as storing the received JWT locally so it can be used when making calls to other protected API routes from the frontend. We will look at the implementation for this when we implement the **Sign In** view later in this chapter.

After the user is successfully signed in, we also want the option to call the signout API when the user is signing out. The call to the signout API is discussed next.

Sign-out

We will add a `signout` method to `api-auth.js`, which will use fetch to make a GET call to the signout API endpoint on the server.

`mern-skeleton/client/auth/api-auth.js`:

```
const signout = async () => {
  try {
    let response = await fetch('/auth/signout/', { method: 'GET
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

This method will also return a promise to inform the component about whether the API request was successful.

At the end of the `api-auth.js` file, we will export the `signin` and `signout` methods.

`mern-skeleton/client/auth/api-auth.js`:

```
export { signin, signout }
```

Now, these methods can be imported into the relevant React components so that we can implement the user sign-in and signout features.

With these API fetch methods added, the React frontend has complete access to the endpoints we made available in the backend. Before we start putting these methods to use in our React components, we will look into how user auth state can be maintained across the frontend.

Adding auth in the frontend

As we discussed in the previous chapter, implementing authentication with JWT relinquishes responsibility to the client-side to manage and store user auth state. To this end, we need to write code that will allow the client-side to store the JWT that's received from the server on successful sign-in, make it available when accessing protected routes, delete or invalidate the token when the user signs out, and also restrict access to views and components on the frontend based on the user auth state.

Using examples of the auth workflow from the React Router documentation, in the following sections, we will write helper methods to manage the auth state across the components, and also use a custom `PrivateRoute` component to add protected routes to the frontend of the MERN skeleton application.

Managing auth state

To manage auth state in the frontend of the application, the frontend needs to be able to store, retrieve, and delete the auth credentials that are received from the server on successful user sign in. In our MERN applications, we will use the browser's `sessionsStorage` as the storage option to store the JWT auth credentials.

> *Alternatively, you can use `localStorage` instead of `sessionStorage` to store the JWT credentials. With `sessionStorage`, the user auth state will only be remembered in the current window tab. With `localStorage`, the user auth state will be remembered across tabs in a browser.*

In `client/auth/auth-helper.js`, we will define the helper methods discussed in the following sections to store and retrieve JWT credentials from client-side `sessionStorage`, and also clear out the `sessionStorage` on user sign-out.

Saving credentials

In order to save the JWT credentials that are received from the server on successful sign-in, we use the `authenticate` method, which is defined as follows.

`mern-skeleton/client/auth/auth-helper.js`:

```
authenticate(jwt, cb) {
    if(typeof window !== "undefined")
        sessionStorage.setItem('jwt', JSON.stringify(jwt))
    cb()
}
```

The `authenticate` method takes the JWT credentials, `jwt`, and a callback function, `cb`, as arguments. It stores the credentials in `sessionStorage` after ensuring `window` is defined, in other words ensuring this code is running in a browser and hence has access to `sessionStorage`. Then, it executes the callback function that is passed in. This callback will allow the component – in our case, the component where sign-in is called – to define actions that should take place after successfully signing in and storing credentials. Next, we will discuss the method that lets us access these stored credentials.

Retrieving credentials

In our frontend components, we will need to retrieve the stored credentials to check if the current user is signed in. In the `isAuthenticated()` method, we can retrieve these credentials from `sessionStorage`.

`mern-skeleton/client/auth/auth-helper.js`:

```
isAuthenticated() {
    if (typeof window == "undefined")
      return false

    if (sessionStorage.getItem('jwt'))
      return JSON.parse(sessionStorage.getItem('jwt'))
    else
      return false
}
```

A call to `isAuthenticated()` will return either the stored credentials or `false`, depending on whether credentials were found in `sessionStorage`. Finding credentials in storage will mean a user is signed in, whereas not finding credentials will mean the user is not signed in. We will also add a method that allows us to delete the credentials from storage when a signed-in user signs out from the application.

Deleting credentials

When a user successfully signs out from the application, we want to clear the stored JWT credentials from `sessionStorage`. This can be accomplished by calling the `clearJWT` method, which is defined in the following code.

`mern-skeleton/client/auth/auth-helper.js`:

```
clearJWT(cb) {
      if(typeof window !== "undefined")
        sessionStorage.removeItem('jwt')
      cb()
      signout().then((data) => {
          document.cookie = "t=; expires=Thu, 01 Jan 1970 00:00
          UTC; path=/;"
      })
  }
```

This `clearJWT` method takes a callback function as an argument, and it removes the JWT credential from `sessionStorage`. The passed in `cb()` function allows the component initiating the `signout` functionality to dictate what should happen after a successful sign-out.

The `clearJWT` method also uses the `signout` method we defined earlier in `api-auth.js` to call the signout API in the backend. If we had used `cookies` to store the credentials instead of `sessionStorage`, the response to this API call would be where we clear the cookie, as shown in the preceding code. Using the signout API call is optional since this is dependent on whether cookies are used as the credential storage mechanism.

With these three methods, we now have ways of storing, retrieving, and deleting JWT credentials on the client-side. Using these methods, the React components we build for the frontend will be able to check and manage user auth state to restrict access in the frontend, as demonstrated in the following section with the custom `PrivateRoute` component.

Deleting credentials

When a user successfully signs out from the application, we want to clear the stored JWT credentials from `sessionStorage`. This can be accomplished by calling the `clearJWT` method, which is defined in the following code.

`mern-skeleton/client/auth/auth-helper.js`:

```
clearJWT(cb) {
      if(typeof window !== "undefined")
        sessionStorage.removeItem('jwt')
      cb()
      signout().then((data) => {
          document.cookie = "t=; expires=Thu, 01 Jan 1970 00:00
          UTC; path=/;"
      })
  }
```

This `clearJWT` method takes a callback function as an argument, and it removes the JWT credential from `sessionStorage`. The passed in `cb()` function allows the component initiating the `signout` functionality to dictate what should happen after a successful sign-out.

The `clearJWT` method also uses the `signout` method we defined earlier in `api-auth.js` to call the signout API in the backend. If we had used `cookies` to store the credentials instead of `sessionStorage`, the response to this API call would be where we clear the cookie, as shown in the preceding code. Using the signout API call is optional since this is dependent on whether cookies are used as the credential storage mechanism.

With these three methods, we now have ways of storing, retrieving, and deleting JWT credentials on the client-side. Using these methods, the React components we build for the frontend will be able to check and manage user auth state to restrict access in the frontend, as demonstrated in the following section with the custom `PrivateRoute` component.

Deleting credentials

When a user successfully signs out from the application, we want to clear the stored JWT credentials from `sessionStorage`. This can be accomplished by calling the `clearJWT` method, which is defined in the following code.

`mern-skeleton/client/auth/auth-helper.js`:

```
clearJWT(cb) {
      if(typeof window !== "undefined")
        sessionStorage.removeItem('jwt')
      cb()
      signout().then((data) => {
          document.cookie = "t=; expires=Thu, 01 Jan 1970 00:00
          UTC; path=/;"
      })
  }
```

This `clearJWT` method takes a callback function as an argument, and it removes the JWT credential from `sessionStorage`. The passed in `cb()` function allows the component initiating the `signout` functionality to dictate what should happen after a successful sign-out.

The `clearJWT` method also uses the `signout` method we defined earlier in `api-auth.js` to call the signout API in the backend. If we had used `cookies` to store the credentials instead of `sessionStorage`, the response to this API call would be where we clear the cookie, as shown in the preceding code. Using the signout API call is optional since this is dependent on whether cookies are used as the credential storage mechanism.

With these three methods, we now have ways of storing, retrieving, and deleting JWT credentials on the client-side. Using these methods, the React components we build for the frontend will be able to check and manage user auth state to restrict access in the frontend, as demonstrated in the following section with the custom `PrivateRoute` component.

The PrivateRoute component

The code in the file defines the `PrivateRoute` component, as shown in the auth flow example at https://reacttraining.com/react-router/web/example/auth-work-flow, which can be found in the React Router documentation. It will allow us to declare protected routes for the frontend to restrict view access based on user auth.

`mern-skeleton/client/auth/PrivateRoute.js`:

```
import React, { Component } from 'react'
import { Route, Redirect } from 'react-router-dom'
import auth from './auth-helper'

const PrivateRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={props => (
    auth.isAuthenticated() ? (
      <Component {...props}/>
    ) : (
      <Redirect to={{
        pathname: '/signin',
        state: { from: props.location }
      }}/>
    )
  )}/>
)

export default PrivateRoute
```

Components to be rendered in this `PrivateRoute` will only load when the user is authenticated, which is determined by a call to the `isAuthenticated` method; otherwise, the user will be redirected to the `Signin` component. We load the components that should have restricted access, such as the user profile component, in a `PrivateRoute`. This will ensure that only authenticated users are able to view the user profile page.

With the backend APIs integrated and the auth management helper methods ready for use in the components, we can now start building the remaining view components that utilize these methods and complete the frontend.

Completing the User frontend

The React components that will be described in this section complete the interactive features we defined for the skeleton by allowing users to view, create, and modify user data stored in the database with respect to auth restrictions. The components we will implement are as follows:

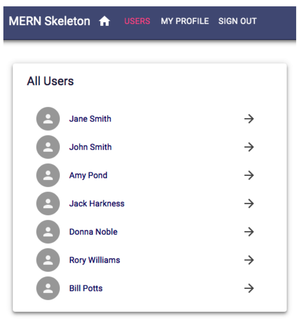
- `Users`: To fetch and list all users from the database to the view
- `Signup`: To display a form that allows new users to sign up
- `Signin`: To display a form that allows existing users to sign in
- `Profile`: To display details for a specific user after retrieving from the database
- `EditProfile`: To display details for a specific user and allow authorized user to update these details

- `DeleteUser`: To allow an authorized user to delete their account from the application
- `Menu`: To add a common navigation bar to each view in the application

For each of these components, we will go over their unique aspects, as well as how to add them to the application in the `MainRouter`.

The Users component

The `Users` component in `client/user/Users.js` shows the names of all the users that have been fetched from the database and links each name to the user profile. The following component can be viewed by any visitor to the application and will render at the `'/users'` route:



In the component definition, similar to how we implemented the `Home` component, we define and export a function component. In this component, we start by initializing the state with an empty array of users.

`mern-skeleton/client/user/Users.js`:

```
export default function Users() {
 ...
 const [users, setUsers] = useState([])
 ...
}
```

We are using the built-in React hook, `useState`, to add state to this function component. By calling this hook, we are essentially declaring a state variable named `users`, which can be updated by invoking `setUsers`, and also set the initial value of `users` to `[]`.

Using the built-in `useState` hook allows us to add state behavior to a function component in React. Calling it will declare a state variable, similar to using `this.state` in class component definitions. The argument that's passed to `useState` is the initial value of this variable – in other words, the initial state. Invoking `useState` returns the current state and a function that updates the state value, which is similar to `this.setState` in a class definition.

With the `users` state initialized, next, we will use another built-in React hook named `useEffect` to fetch a list of users from the backend and update the `users` value in the state.

The Effect Hook, `useEffect`, serves the purpose of the `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` React life cycle methods that we would otherwise use in React classes. Using this hook in a function component allows us to perform side effects such as fetching data from a backend. By default, React runs the effects defined with `useEffect` after every render, including the first render. But we can also instruct the effect to only rerun if something changes in state. Optionally, we can also define how to clean up after an effect, for example, to perform an action such as aborting a fetch signal when the component unmounts to avoid memory leaks.

In our `Users` component, we use `useEffect` to call the `list` method from the `api-user.js` helper methods. This will fetch the user list from the backend and load the user data into the component by updating the state.

`mern-skeleton/client/user/Users.js`:

```
  useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal

    list(signal).then((data) => {
      if (data && data.error) {
        console.log(data.error)
      } else {
        setUsers(data)
      }
    })

    return function cleanup(){
      abortController.abort()
    }
  }, [])
```

In this effect, we also add a cleanup function to abort the fetch call when the component unmounts. To associate a signal with the fetch call, we use the AbortController web API, which allows us to abort DOM requests as needed.

In the second argument of this `useEffect` hook, we pass an empty array so that this effect cleanup runs only once upon mounting and unmounting, and not after every render.

Finally, in the return of the `Users` function component, we add the actual view content. The view is composed of Material-UI components such as `Paper`, `List`, and `ListItem`. These elements are styled with the CSS that is defined and made available with the `makeStyles` hook, the same way as in the `Home` component.

`mern-skeleton/client/user/Users.js`:

```
return (
      <Paper className={classes.root} elevation={4}>
        <Typography variant="h6" className={classes.title}>
          All Users
        </Typography>
        <List dense>
          {users.map((item, i) => {
            return <Link to={"/user/" + item._id} key={i}>
                     <ListItem button>
                       <ListItemAvatar>
                         <Avatar>
                           <Person/>
                         </Avatar>
                       </ListItemAvatar>
                       <ListItemText primary={item.name}/>
                       <ListItemSecondaryAction>
                       <IconButton>
                           <ArrowForward/>
                       </IconButton>
                       </ListItemSecondaryAction>
                     </ListItem>
                   </Link>
                })
              }
        </List>
      </Paper>
)
```

In this view, to generate each list item, we iterate through the array of users in the state using the `map` function. A list item is rendered with an individual user's name from each item that's accessed per iteration on the users array.

To add this `Users` component to the React application, we need to update the `MainRouter` component with a `Route` that renders this component at the `'/users'` path. Add the `Route` inside the `Switch` component after the `Home` route.

`mern-skeleton/client/MainRouter.js`:

```
<Route path="/users" component={Users}/>
```

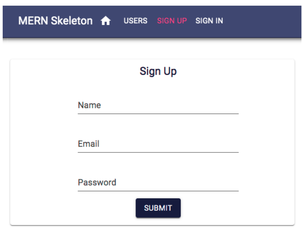
To see this view rendered in the browser, you can temporarily add a `Link` component to the `Home` component to be able to route to the `Users` component:

```
<Link to="/users">Users</Link>
```

Clicking on this link after rendering the Home view at the root route in the browser will display the `Users` component we implemented in this section. We will implement the other React components similarly, starting with the `Signup` component in the next section.

The Signup component

The `Signup` component in `client/user/Signup.js` presents a form with name, email, and password fields to the user for sign-up at the `'/signup'` path, as displayed in the following screenshot:



In the component definition, we initialize the state using the `useState` hook with empty input field values, an empty error message, and set the dialog open variable to false.

`mern-skeleton/client/user/Signup.js`:

```
export default function Signup() {
  ...
  const [values, setValues] = useState({
    name: '',
    password: '',
    email: '',
    open: false,
    error: ''
  })
  ...
}
```

We also define two handler functions to be called when the input values change or the submit button is clicked. The `handleChange` function takes the new value that's entered in the input field and sets it as the state.

`mern-skeleton/client/user/Signup.js`:

```
const handleChange = name => event => {
    setValues({ ...values, [name]: event.target.value })
}
```

The `clickSubmit` function is called when the form is submitted. It takes the input values from the state and calls the `create` fetch method to sign up the user with the backend. Then, depending on the response from the server, either an error message is shown or a success dialog is shown.

`mern-skeleton/client/user/Signup.js`:

```
  const clickSubmit = () => {
    const user = {
      name: values.name || undefined,
      email: values.email || undefined,
      password: values.password || undefined
    }
    create(user).then((data) => {
      if (data.error) {
        setValues({ ...values, error: data.error})
      } else {
        setValues({ ...values, error: '', open: true})
      }
    })
  }
```

In the `return` function, we compose and style the form components in the signup view using components such as `TextField` from Material-UI.

`mern-skeleton/client/user/Signup.js`:

```
return (
    <div>
      <Card className={classes.card}>
        <CardContent>
          <Typography variant="h6" className={classes.title}>
            Sign Up
          </Typography>
          <TextField id="name" label="Name"
            className={classes.textField}
            value={values.name} onChange={handleChange('name')
            margin="normal"/>
          <br/>
          <TextField id="email" type="email" label="Email"
            className={classes.textField}
            value={values.email} onChange={handleChange('email
            margin="normal"/>
          <br/>
          <TextField id="password" type="password" label="Passw
            className={classes.textField} value={values.passwo
            onChange={handleChange('password')} margin="normal
          <br/>
          {
            values.error && (<Typography component="p" color="e
              <Icon color="error" className={classes.error}>err
              {values.error}</Typography>)
          }
        </CardContent>
        <CardActions>
          <Button color="primary" variant="contained" onClick={
            className={classes.submit}>Submit</Button>
        </CardActions>
      </Card>
    </div>
)
```

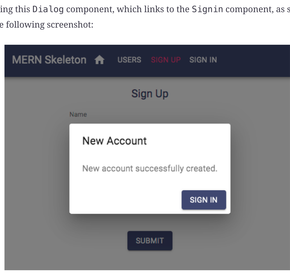
This return also contains an error message block, along with a `Dialog` component that is conditionally rendered depending on the signup response from the server. If the server returns an error, the error block that was added below the form, which we implemented in the preceding code, will render in the view with the corresponding error message. If the server returns a successful response, a `Dialog` component will be rendered instead.

The `Dialog` component in `Signup.js` is composed as follows.

`mern-skeleton/client/user/Signup.js`:

```
<Dialog open={values.open} disableBackdropClick={true}>
    <DialogTitle>New Account</DialogTitle>
    <DialogContent>
      <DialogContentText>
        New account successfully created.
      </DialogContentText>
    </DialogContent>
    <DialogActions>
      <Link to="/signin">
        <Button color="primary" autoFocus="autoFocus"
          variant="contained">
          Sign In
        </Button>
      </Link>
    </DialogActions>
</Dialog>
```

On successful account creation, the user is given confirmation and asked to sign in using this `Dialog` component, which links to the `Signin` component, as shown in the following screenshot:



To add the `Signup` component to the app, add the following `Route` to `MainRouter` in the `Switch` component.

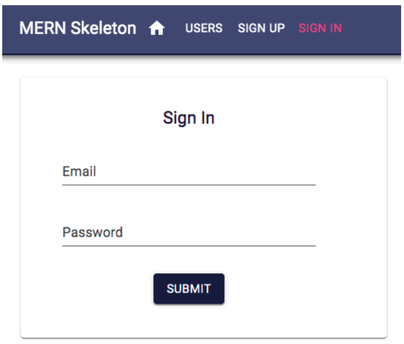
`mern-skeleton/client/MainRouter.js`:

```
<Route path="/signup" component={Signup}/>
```

This will render the `Signup` view at `'/signup'`. Similarly, we will implement the `Signin` component next.

The Signin component

The `Signin` component in `client/auth/Signin.js` is also a form with only email and password fields for signing in. This component is quite similar to the `Signup` component and will render at the `'/signin'` path. The key difference is in the implementation of redirection after a successful sign-in and storing the received JWT credentials. The rendered `Signin` component can be seen in the following screenshot:



For redirection, we will use the `Redirect` component from React Router. First, initialize a `redirectToReferrer` value to `false` in the state with the other fields:

`mern-skeleton/client/auth/Signin.js`:

```
export default function Signin(props) {
  const [values, setValues] = useState({
      email: '',
      password: '',
      error: '',
      redirectToReferrer: false
  })
}
```

The `Signin` function will take props in the argument that contain React Router variables. We will use these for the redirect. `redirectToReferrer` should be set to `true` when the user successfully signs in after submitting the form and the received JWT is stored in `sessionStorage`. To store the JWT and redirect afterward, we will call the `authenticate()` method defined in `auth-helper.js`. This implementation will go in the `clickSubmit()` function so that it can be called on form submit.

`mern-skeleton/client/auth/Signin.js`:

```
const clickSubmit = () => {
    const user = {
      email: values.email || undefined,
      password: values.password || undefined
    }

    signin(user).then((data) => {
      if (data.error) {
        setValues({ ...values, error: data.error})
      } else {
        auth.authenticate(data, () => {
          setValues({ ...values, error: '',redirectToReferrer:
        })
      }
    })
}
```

The redirection will happen conditionally based on the `redirectToReferrer` value using the `Redirect` component from React Router. We add the redirect code inside the function before the return block, as follows.

`mern-skeleton/client/auth/Signin.js`:

```
const {from} = props.location.state || {
      from: {
        pathname: '/'
      }
  }
const {redirectToReferrer} = values
if (redirectToReferrer) {
      return (<Redirect to={from}/>)
}
```

The `Redirect` component, if rendered, will take the app to the last location that was received in the props or to the `Home` component at the root.

The function return code is not displayed here as it is very similar to the code in `Signup`. It will contain the same form elements with just `email` and `password` fields, a conditional error message, and the `submit` button.

To add the `Signin` component to the app, add the following Route to `MainRouter` in the `Switch` component.

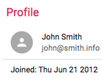
`mern-skeleton/client/MainRouter.js`:

```
<Route path="/signin" component={Signin}/>
```

This will render the `Signin` component at `"/signin"` and can be linked in the Home component, similar to the `Signup` component, so that it can be viewed in the browser. Next, we will implement the profile view to display the details of a single user.

The Profile component

The `Profile` component in `client/user/Profile.js` shows a single user's information in the view at the `'/user/:userId'` path, where the `userId` parameter represents the ID of the specific user. The completed `Profile` will display user details, and also conditionally show edit/delete options. The following screenshot shows how the `Profile` renders when the user currently browsing is viewing someone else's profile and not their own profile:



This profile information can be fetched from the server if the user is signed in. To verify this, the component has to provide the JWT credential to the `read` fetch call; otherwise, the user should be redirected to the Sign In view.

In the `Profile` component definition, we need to initialize the state with an empty user and set `redirectToSignin` to `false`.

`mern-skeleton/client/user/Profile.js`:

```
export default function Profile({ match }) {
    ...
    const [user, setUser] = useState({})
    const [redirectToSignin, setRedirectToSignin] = useState(fa
    ...
}
```

We also need to get access to the `match` props passed by the `Route` component, which will contain a `:userId` parameter value. This can be accessed as `match.params.userId`.

The `Profile` component should fetch user information and render the view with these details. To implement this, we will use the `useEffect` hook, as we did in the `Users` component.

`mern-skeleton/client/user/Profile.js`:

```
  useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal
    const jwt = auth.isAuthenticated()
    read({
      userId: match.params.userId
    }, {t: jwt.token}, signal).then((data) => {
      if (data && data.error) {
        setRedirectToSignin(true)
      } else {
        setUser(data)
      }
    })

    return function cleanup(){
      abortController.abort()
    }

  }, [match.params.userId])
```

This effect uses the `match.params.userId` value and calls the `read` user fetch method. Since this method also requires credentials to authorize the signed-in user, the JWT is retrieved from `sessionStorage` using the `isAuthenticated` method from `auth-helper.js`, and passed in the call to `read`.

Once the server responds, either the state is updated with the user information or the view is redirected to the Sign In view if the current user is not authenticated. We also add a cleanup function in this effect hook to abort the fetch signal when the component unmounts.

This effect only needs to rerun when the `userId` parameter changes in the route, for example, when the app goes from one profile view to the other. To ensure this effect reruns when the `userId` value updates, we will add `[match.params.userId]` in the second argument to `useEffect`.

If the current user is not authenticated, we set up the conditional redirect to the Sign In view.

`mern-skeleton/client/user/Profile.js`

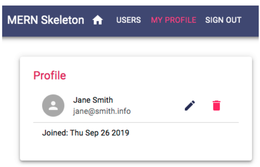
```
if (redirectToSignin) {
  return <Redirect to='/signin'/>
}
```

The function will return the `Profile` view with the following elements if the user who's currently signed in is viewing another user's profile.

`mern-skeleton/client/user/Profile.js`:

```
return (
      <Paper className={classes.root} elevation={4}>
        <Typography variant="h6" className={classes.title}>
          Profile
        </Typography>
        <List dense>
          <ListItem>
            <ListItemAvatar>
              <Avatar>
                <Person/>
              </Avatar>
            </ListItemAvatar>
            <ListItemText primary={user.name} secondary={user.e
          </ListItem>
          <Divider/>
          <ListItem>
            <ListItemText primary={"Joined: " + (
              new Date(user.created)).toDateString()}/>
          </ListItem>
        </List>
      </Paper>
)
```

However, if the user that's currently signed in is viewing their own profile, they will be able to see edit and delete options in the `Profile` component, as shown in the following screenshot:



To implement this feature, in the first `ListItem` component in the `Profile`, add a `ListItemSecondaryAction` component containing the `Edit` button and a `DeleteUser` component, which will render conditionally based on whether the current user is viewing their own profile.

`mern-skeleton/client/user/Profile.js`:

```
{ auth.isAuthenticated().user && auth.isAuthenticated().user._i
    (<ListItemSecondaryAction>
        <Link to={"/user/edit/" + user._id}>
          <IconButton aria-label="Edit" color="primary">
            <Edit/>
          </IconButton>
        </Link>
        <DeleteUser userId={user._id}/>
    </ListItemSecondaryAction>)
}
```

The `Edit` button will route to the `EditProfile` component, while the custom `DeleteUser` component will handle the delete operation with the `userId` passed to it as a prop.

To add the `Profile` component to the app, add the `Route` to `MainRouter` in the `Switch` component.

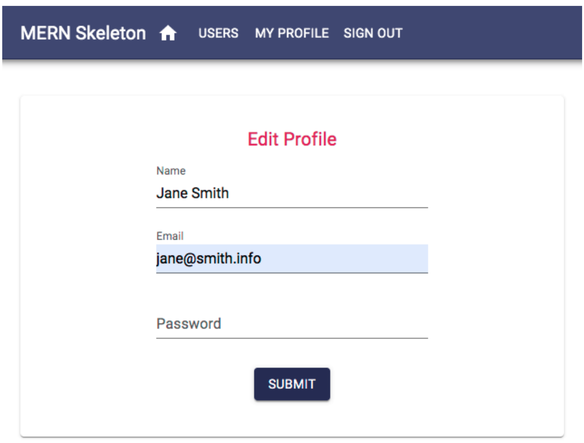
`mern-skeleton/client/MainRouter.js`:

```
<Route path="/user/:userId" component={Profile}/>
```

To visit this route in the browser and render a `Profile` with user details, the link should be composed with a valid user ID in it. In the next section, we will use this same approach of retrieving single user details and rendering it in the component to implement the Edit Profile view.

The EditProfile component

The `EditProfile` component in `client/user/EditProfile.js` has similarities in its implementation to both the `Signup` and `Profile` components. It allows the authorized user to edit their own profile information in a form similar to the signup form, as shown in the following screenshot:



Upon loading at `'/user/edit/:userId'`, the component will fetch the user's information with their ID after verifying JWT for auth, and then load the form with the received user information. The form will allow the user to edit and submit only the changed information to the `update` fetch call, and, on successful update, redirect the user to the `Profile` view with updated information.

`EditProfile` will load the user information the same way as in the `Profile` component, that is, by fetching with `read` in `useEffect` using the `userId` parameter from `match.params`. It will gather credentials from `auth.isAuthenticated`. The form view will contain the same elements as the `Signup` component, with the input values being updated in the state when they change.

On form submit, the component will call the `update` fetch method with the `userId`, JWT and updated user data.

`mern-skeleton/client/user/EditProfile.js`:

```
const clickSubmit = () => {
const jwt = auth.isAuthenticated()
    const user = {
      name: values.name || undefined,
      email: values.email || undefined,
      password: values.password || undefined
    }
    update({
      userId: match.params.userId
    }, {
      t: jwt.token
    }, user).then((data) => {
      if (data && data.error) {
        setValues({...values, error: data.error})
      } else {
        setValues({...values, userId: data._id, redirectToProfi
      }
    })
}
```

Depending on the response from the server, the user will either see an error message or be redirected to the updated Profile page using the `Redirect` component, as follows.

`mern-skeleton/client/user/EditProfile.js`:

```
if (values.redirectToProfile) {
      return (<Redirect to={'/user/' + values.userId}/>)
}
```

To add the `EditProfile` component to the app, we will use a `PrivateRoute`, which will restrict the component from loading at all if the user is not signed in. The order of placement in `MainRouter` will also be important.

`mern-skeleton/client/MainRouter.js`:

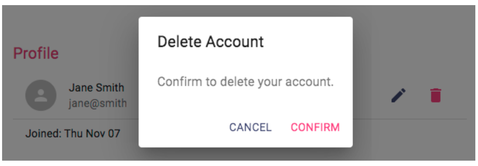
```
<Switch>
  ...
  <PrivateRoute path="/user/edit/:userId" component={EditProfil
  <Route path="/user/:userId" component={Profile}/>
</Switch>
```

The route with the `'/user/edit/:userId'` path needs to be placed before the route with the `'/user/:userId'` path, so that the edit path is matched first exclusively in the Switch component when this route is requested, and not confused with the `Profile` route.

With this profile edit view added, we only have the user delete UI implementation left to complete the user-related frontend.

The DeleteUser component

The `DeleteUser` component in `client/user/DeleteUser.js` is basically a button that we will add to the Profile view that, when clicked, opens a `Dialog` component asking the user to confirm the `delete` action, as shown in the following screenshot:



This component initializes the state with `open` set to `false` for the `Dialog` component, as well as `redirect` set to `false` so that it isn't rendered first.

`mern-skeleton/client/user/DeleteUser.js`:

```
export default function DeleteUser(props) {
    ...
    const [open, setOpen] = useState(false)
    const [redirect, setRedirect] = useState(false)
    ...
}
```

The `DeleteUser` component will also receive props from the parent component. In this case, the props will contain the `userId` that was sent from the `Profile` component.

Next, we need some handler methods to open and close the `dialog` button. The dialog is opened when the user clicks the `delete` button.

`mern-skeleton/client/user/DeleteUser.js`:

```
const clickButton = () => {
    setOpen(true)
}
```

The dialog is closed when the user clicks `cancel` on the dialog.

`mern-skeleton/client/user/DeleteUser.js`:

```
const handleRequestClose = () => {
    setOpen(false)
}
```

The component will have access to the `userId` that's passed in as a prop from the `Profile` component, which is needed to call the `remove` fetch method, along with the JWT credentials, after the user confirms the `delete` action in the dialog.

`mern-skeleton/client/user/DeleteUser.js`:

```
const deleteAccount = () => {
const jwt = auth.isAuthenticated()
    remove({
      userId: props.userId
    }, {t: jwt.token}).then((data) => {
      if (data && data.error) {
        console.log(data.error)
      } else {
        auth.clearJWT(() => console.log('deleted'))
        setRedirect(true)
      }
    })
}
```

On confirmation, the `deleteAccount` function calls the `remove` fetch method with the `userId` from props and JWT from `isAuthenticated`. On successful deletion, the user will be signed out and redirected to the Home view. The `Redirect` component from React Router is used to redirect the current user to the Home view, as follows:

```
if (redirect) {
    return <Redirect to='/'/>
}
```

The component function returns the `DeleteUser` component elements, including a `DeleteIcon` button and the confirmation `Dialog`.

`mern-skeleton/client/user/DeleteUser.js`:

```
return (<span>
      <IconButton aria-label="Delete"
          onClick={clickButton} color="secondary">
        <DeleteIcon/>
      </IconButton>

      <Dialog open={open} onClose={handleRequestClose}>
        <DialogTitle>{"Delete Account"}</DialogTitle>
        <DialogContent>
          <DialogContentText>
            Confirm to delete your account.
          </DialogContentText>
        </DialogContent>
        <DialogActions>
          <Button onClick={handleRequestClose} color="primary">
 Cancel
          </Button>
          <Button onClick={deleteAccount}
              color="secondary" autoFocus="autoFocus">
            Confirm
          </Button>
        </DialogActions>
      </Dialog>
    </span>)
```

`DeleteUser` takes the `userId` as a prop to be used in the `delete` fetch call, so we need to add a required prop validation check for this React component. We'll do this next.

Validating props with PropTypes

To validate the required injection of `userId` as a prop to the component, we'll add the `PropTypes` requirement validator to the defined component.

`mern-skeleton/client/user/DeleteUser.js`:

```
DeleteUser.propTypes = {
  userId: PropTypes.string.isRequired
}
```

Since we are using the `DeleteUser` component in the `Profile` component, it gets added to the application view when `Profile` is added in `MainRouter`.

With the delete user UI added, we now have a frontend that contains all the React component views in order to complete the skeleton application features. But, we still need a common navigation UI to link all these views together and make each view easy to access for the frontend user. In the next section, we will implement this navigation menu component.

The Menu component

The `Menu` component will function as a navigation bar across the frontend application by providing links to all the available views, and also by indicating the user's current location in the application.

To implement these navigation bar functionalities, we will use the HOC `withRouter` from React Router to get access to the `history` object's properties. The following code in the `Menu` component adds just the title, the `Home` icon linked to the root route, and the `Users` button, which is linked to the `'/users'` route.

`mern-skeleton/client/core/Menu.js`:

```
const Menu = withRouter(({history}) => (
  <AppBar position="static">
    <Toolbar>
      <Typography variant="h6" color="inherit">
        MERN Skeleton
      </Typography>
      <Link to="/">
        <IconButton aria-label="Home" style={isActive(history,
          <HomeIcon/>
        </IconButton>
      </Link>
      <Link to="/users">
        <Button style={isActive(history, "/users")}>Users</Butt
      </Link>
    </Toolbar>
  </AppBar>))
```

To indicate the current location of the application on the `Menu`, we will highlight the link that matches the current location path by changing the color conditionally.

`mern-skeleton/client/core/Menu.js`:

```
const isActive = (history, path) => {
  if (history.location.pathname == path)
    return {color: '#ff4081'}
  else
    return {color: '#ffffff'}
}
```

The `isActive` function is used to apply color to the buttons in the `Menu`, as follows:

```
style={isActive(history, "/users")}
```

The remaining links such as SIGN IN, SIGN UP, MY PROFILE, and SIGN OUT will show up on the `Menu` based on whether the user is signed in or not. The following screenshot shows how the Menu renders when the user is not signed in:

For example, the links to SIGN UP and SIGN IN should only appear on the menu when the user is not signed in. Therefore, we need to add it to the `Menu` component after the `Users` button with a condition.

`mern-skeleton/client/core/Menu.js`:

```
{
  !auth.isAuthenticated() && (<span>
    <Link to="/signup">
       <Button style={isActive(history, "/signup")}> Sign Up </
    </Link>
    <Link to="/signin">
       <Button style={isActive(history, "/signin")}> Sign In </
    </Link>
</span>)
}
```

Similarly, the link to `MY PROFILE` and the `SIGN OUT` button should only appear on the menu when the user is signed in, and should be added to the `Menu` component with the following condition check.

`mern-skeleton/client/core/Menu.js`:

```
{
 auth.isAuthenticated() && (<span>
   <Link to={"/user/" + auth.isAuthenticated().user._id}>
      <Button style={isActive(history, "/user/"
            + auth.isAuthenticated().user._id)}>
          My Profile
      </Button>
   </Link>
   <Button color="inherit"
          onClick={() => { auth.clearJWT(() => history.push('/
       Sign out
   </Button>
 </span>)
}
```

The MY PROFILE button uses the signed-in user's information to link to the user's own profile, while the SIGN OUT button calls the `auth.clearJWT()` method when it's clicked. When the user is signed in, the `Menu` will look as follows:

To have the `Menu` navigation bar present in all the views, we need to add it to the `MainRouter` before all the other routes, and outside the `Switch` component.

`mern-skeleton/client/MainRouter.js`:

```
    <Menu/>
    <Switch>
    …
    </Switch>
```

This will make the `Menu` component render on top of all the other components when these components are accessed at their respective routes.

The skeleton frontend is now complete and has all necessary components to allow a user to sign up, view, and modify user data on the backend while considering authentication and authorization restrictions. However, it is still not possible to visit the frontend routes directly in the browser address bar; these can only be accessed when they're linked from within the frontend view. To enable this functionality in the skeleton application, we need to implement basic server-side rendering.

Implementing basic server-side rendering

Currently, when the React Router routes or pathnames are directly entered in the browser address bar or when a view that is not at the root path is refreshed, the URL does not work. This happens because the server does not recognize the React Router routes we defined in the frontend. We have to implement basic server-side rendering on the backend so that the server is able to respond when it receives a request to a frontend route.

To render the relevant React components properly when the server receives requests to the frontend routes, we need to initially generate the React components on the server-side with regard to the React Router and Material-UI components, before the client-side JS is ready to take over the rendering.

The basic idea behind server-side rendering React apps is to use the `renderToString` method from `react-dom` to convert the root React component into a markup string. Then, we can attach it to the template that the server renders when it receives a request.

In `express.js`, we will replace the code that returns `template.js` in response to the `GET` request for `'/'` with code that, upon receiving any incoming GET request, generates some server-side rendered markup and the CSS of the relevant React component tree, before adding this markup and CSS to the template. This updated code will achieve the following:

```
app.get('*', (req, res) => {
    // 1. Generate CSS styles using Material-UI's ServerStyleS
    // 2. Use renderToString to generate markup which renders
          components specific to the route requested
    // 3. Return template with markup and CSS styles in the re
})
```

In the following sections, we will look at the implementation of the steps outlined in the preceding code block, and also discuss how to prepare the frontend so that it accepts and handles this server-rendered code.

Modules for server-side rendering

To implement basic server-side rendering, we will need to import the following React, React Router, and Material-UI-specific modules into the server code. In our code structure, the following modules will be imported into `server/express.js`:

- **React modules**: The following modules are required to render the React components and use `renderToString`:

```
import React from 'react'
import ReactDOMServer from 'react-dom/server'
```

- **Router modules**: `StaticRouter` is a stateless router that takes the requested URL to match with the frontend route which was declared in the `MainRouter` component. The `MainRouter` is the root component in our frontend.

```
import StaticRouter from 'react-router-dom/StaticRouter'
import MainRouter from './../client/MainRouter'
```

- **Material-UI modules and the custom theme**: The following modules will help generate the CSS styles for the frontend components based on the stylings and Material-UI theme that are used on the frontend:

```
import { ServerStyleSheets, ThemeProvider } from '@material-ui/
import theme from './../client/theme'
```

With these modules, we can prepare, generate, and return server-side rendered frontend code, as we will discuss next.

Generating CSS and markup

To generate the CSS and markup representing the React frontend views on the server-side, we will use Material-UI's `ServerStyleSheets` and React's `renderToString`.

On every request received by the Express app, we will create a new `ServerStyleSheets` instance. Then, we will render the relevant React tree with the server-side collector in a call to `renderToString`, which ultimately returns the associated markup or HTML string version of the React view that is to be shown to the user in response to the requested URL.

The following code will be executed on every GET request that's received by the Express app.

`mern-skeleton/server/express.js`:

```
const sheets = new ServerStyleSheets()
const context = {}
const markup = ReactDOMServer.renderToString(
    sheets.collect(
          <StaticRouter location={req.url} context={context}>
            <ThemeProvider theme={theme}>
              <MainRouter />
            </ThemeProvider>
          </StaticRouter>
        )
  )
```

While rendering the React tree, the client app's root component, `MainRouter`, is wrapped with the Material-UI `ThemeProvider` to provide the styling props that are needed by the `MainRouter` child components. The stateless `StaticRouter` is used here instead of the `BrowserRouter` that's used on the client-side in order to wrap `MainRouter` and provide the routing props that are used for implementing the client-side components.

Based on these values, such as the requested `location` route and `theme` that are passed in as props to the wrapping components, `renderToString` will return the markup containing the relevant view.

Sending a template with markup and CSS

Once the markup has been generated, we need to check if there was a `redirect` rendered in the component to be sent in the markup. If there was no redirect, then we get the CSS string from `sheets` using `sheets.toString`, and, in the response, we send the `Template` back with the markup and CSS injected, as shown in the following code.

`mern-skeleton/server/express.js`:

```
if (context.url) {
   return res.redirect(303, context.url)
}
const css = sheets.toString()
res.status(200).send(Template({
   markup: markup,
   css: css
}))
```

An example of a case where redirect is rendered in the component is when we're trying to access a `PrivateRoute` via a server-side render. As the server-side cannot access the auth token from the browser's `sessionStorage`, the redirect in `PrivateRoute` will render. The `context.url` value , in this case, will have the `'/signin'` route, and hence, instead of trying to render the `PrivateRoute` component, it will redirect to the `'/signin'` route.

This completes the code we need to add to the server-side to enable the basic server-side rendering of the React views. Next, we need to update the frontend so it is able to integrate and render this server-generated code.

Updating template.js

The markup and CSS that we generated on the server must be added to the `template.js` HTML code for it to be loaded when the server renders the template.

`mern-skeleton/template.js`:

```
export default ({markup, css}) => {
    return `...
           <div id="root">${markup}</div>
           <style id="jss-server-side">${css}</style>
           ...`
}
```

This will load the server-generated code in the browser before the frontend script is ready to take over. In the next section, we will learn how the frontend script needs to account for this takeover from server-rendered code.

Updating App.js

Once the code that's been rendered on the server-side reaches the browser and the frontend script takes over, we need to remove the server-side injected CSS when the root React component mounts, using the `useEffect` hook.

`mern-skeleton/client/App.js`:

```
React.useEffect(() => {
    const jssStyles = document.querySelector('#jss-server-side'
    if (jssStyles) {
      jssStyles.parentNode.removeChild(jssStyles)
    }
}, [])
```

This will give back full control over rendering the React app to the client-side. To ensure this transfer happens efficiently, we need to update how the ReactDOM renders the views.

Hydrate instead of render

Now that the React components will be rendered on the server-side, we can update the `main.js` code so that it uses `ReactDOM.hydrate()` instead of `ReactDOM.render()`:

```
import React from 'react'
import { hydrate } from 'react-dom'
import App from './App'

hydrate(<App/>, document.getElementById('root'))
```

The `hydrate` function hydrates a container that already has HTML content rendered by `ReactDOMServer`. This means the server-rendered markup is preserved and only event handlers are attached when React takes over in the browser, allowing the initial load performance to be better.

With basic server-side rendering implemented, direct requests to the frontend routes from the browser address bar can now be handled properly by the server, making it possible to bookmark the React frontend views.

The skeleton MERN application that we've developed in this chapter is now a completely functioning MERN web application with basic user features. We can extend the code in this skeleton to add a variety of features for different applications.

Summary

In this chapter, we completed the MERN skeleton application by adding a working React frontend, including frontend routing and basic server-side rendering of the React views.

We started off by updating the development flow so that it included client-side code bundling for the React views. We updated the configuration for Webpack and Babel to compile the React code and discussed how to load the configured Webpack middleware from the Express app to initiate server-side and client-side code compilation from one place during development.

With the development flow updated, and before building out the frontend, we added the relevant React dependencies, along with React Router for frontend routing and Material-UI, to use their existing components in the skeleton app's user interface.

Then, we implemented the top-level root React components and integrated React Router, which allowed us to add client-side routes for navigation. Using these routes, we loaded the custom React components that we developed using Material-UI components to make up the skeleton application's user interface.

To make these React views dynamic and interactive with data fetched from the backend, we used the Fetch API to connect to the backend user APIs. Then, we incorporated authentication and authorization on the frontend views. We did this using `sessionStorage`, which stores user-specific details, and JWT fetched from the server on successful sign-in, as well as by limiting access to certain views using a `PrivateRoute` component.

Finally, we modified the server code so that we could implement basic server-side rendering, which allows us to load the frontend routes directly in the browser with server-side rendered markup after the server recognizes that the incoming request is actually for a React route.

Now, you should be able to implement and integrate a React-based frontend that incorporates client-side routing and auth management with a standalone server application.

In the next chapter, we will use the concepts we've learned in this chapter to extend the skeleton application code so that we can build a fully-featured social media application.