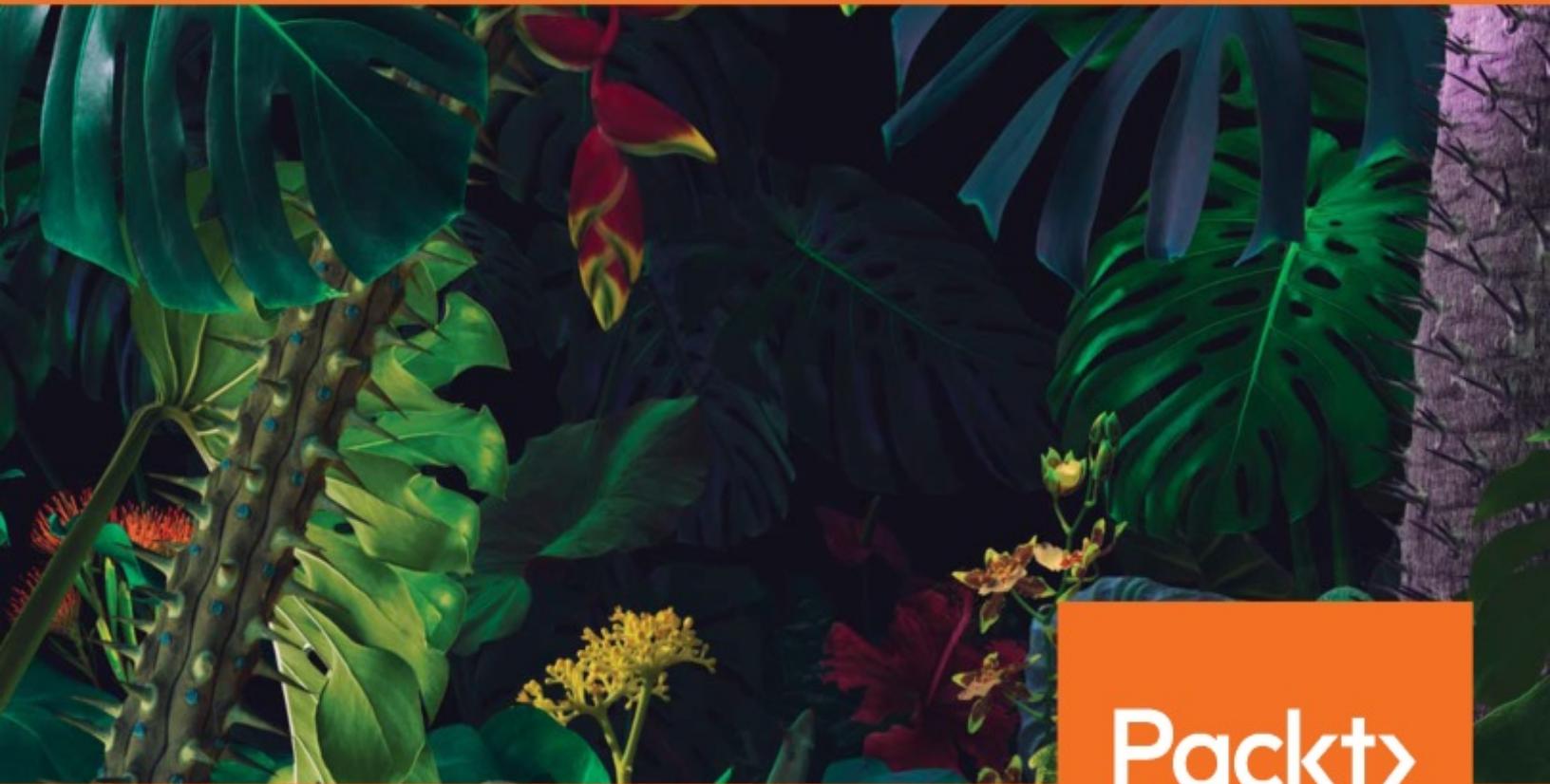


Full-Stack React Projects

Second Edition

Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node.js



Packt

www.packt.com

Shama Hoque

Full-Stack React Projects

Second Edition

Learn MERN stack development by building modern web apps using MongoDB, Express, React, and Node.js

Shama Hoque

Packt

BIRMINGHAM - MUMBAI

Full-Stack React Projects

Second Edition

Copyright © 2020 Packt Publishing

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

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

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisition Editor: Clint Rodricks

Content Development Editor: Aamir Ahmed

Senior Editor: Hayden Edwards

Technical Editor: Deepesh Patel

Copy Editor: Safis Editing

Project Coordinator: Kinjal Bari

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Joshua Misquitta

First published: May 2018

Second edition: April 2020

Production reference: 1160420

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83921-541-4

www.packt.com

To my parents, for setting examples of perseverance and relentless dedication.

-Shama Hoque



[Packt.com](https://www.packt.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Shama Hoque has more than 9 years of experience as a software developer and mentor, with a master's in software engineering from Carnegie Mellon University.

From Java programming to full-stack development with JavaScript, the applications she has worked on include national Olympiad registration websites, universally accessible widgets, video conferencing apps, and 3D medical reconstruction software.

Currently, she makes web-based prototypes for R&D start-ups in California, while training aspiring software engineers and teaching web development to CS undergrads in Bangladesh.

About the reviewers

Kirill Ezhemenskii is an experienced software engineer, frontend and mobile developer, solution architect, and the CTO of a healthcare company. He is also a functional programming advocate; an expert in the React stack, GraphQL, and TypeScript; and a React Native mentor.

Carlos Santana Roldán is a senior web developer with more than 12 years of experience. Currently, he is working as a senior React developer at MindBody Inc. He is the founder of Dev Education, one of the most popular developer communities in Latin America, training people in web technologies such as React, Node.js, GraphQL, and JavaScript in general.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [author](https://s.packtpub.com) s.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Title Page	
Copyright and Credits	
Full-Stack React Projects Second Edition	
Dedication	
About Packt	
Why subscribe?	
Contributors	
About the author	
About the reviewers	
Packt is searching for authors like you	
Preface	
Who this book is for	
What this book covers	
To get the most out of this book	
Download the example code files	
Conventions used	
Get in touch	
Reviews	
1. Getting Started with MERN	<hr/>
1. Unleashing React Applications with MERN	
What is new in the second edition?	
Book structure	
Getting started with MERN	
Building MERN applications from the ground up 	
Developing web applications with MERN	
Advancing to complex MERN applications	
Going forward with MERN	
Getting the most out of this book	
The MERN stack	
Node	
Express	
MongoDB	
React	

Relevance of MERN

- Consistency across the technology stack
- Takes less time to learn, develop, deploy, and extend
- Widely adopted in the industry
- Community support and growth

Range of MERN applications

- MERN applications developed in this book
 - Social media platform
 - Web-based classroom application
 - Online marketplace
 - Expense tracking application
 - Media streaming application
 - VR game for the web

Summary

2. Preparing the Development Environment

Selecting development tools

- Workspace options
 - Local and cloud development
- IDE or text editors
- Chrome Developer Tools
- Git
 - Installation
 - Remote Git hosting services

Setting up MERN stack technologies

- MongoDB
 - Installation
 - Running the mongo shell
- Node.js
 - Installation
 - Node version management with nvm
 - Yarn package manager
- Modules for MERN
 - Key modules
 - devDependency modules

Checking your development setup

- Initializing package.json and installing Node.js modules
- Configuring Babel, Webpack, and Nodemon

- Babel
- Webpack
 - Client-side Webpack configuration for development
 - Server-side Webpack configuration
 - Client-side Webpack configuration for production
- Nodemon
- Frontend views with React
- Server with Express and Node
 - Express app
 - Bundling React app during development
 - Serving static files from the dist folder
 - Rendering templates at the root
- Connecting the server to MongoDB
- Run scripts
- Developing and debugging in real time

Summary

2. Building MERN from the Ground Up

3. Building a Backend with MongoDB, Express, and Node

- Overview of the skeleton application
 - Feature breakdown
 - Defining the backend components
 - User model
 - API endpoints for user CRUD
 - Auth with JSON Web Tokens
 - How JWT works
- Setting up the skeleton backend
 - Folder and file structure
 - Initializing the project
 - Adding package.json
 - Development dependencies
 - Babel
 - Webpack
 - Nodemon
 - Config variables
 - Running scripts
- Preparing the server
 - Configuring Express

```
Starting the server
Setting up Mongoose and connecting to MongoDB
Serving an HTML template at a root URL

Implementing the user model
  User schema definition
    Name
    Email
    Created and updated timestamps
    Hashed password and salt
    Password for auth
      Handling the password string as a virtual field
      Encryption and authentication
      Password field validation
    Mongoose error handling

Adding user CRUD APIs;
  User routes
  User controller
  Creating a new user
  Listing all users
  Loading a user by ID to read, update, or delete
    Loading;
    Reading
    Updating
    Deleting

Integrating user auth and protected routes
  Auth routes
  Auth controller
  Sign-in
  Signout
  Protecting routes with express-jwt
    Protecting user routes
    Requiring sign-in
    Authorizing signed in users
    Auth error handling for express-jwt

Checking the standalone backend
  Creating a new user
  Fetching the user list
```

```
    Trying to fetch a single user
    Signing in
    Fetching a single user successfully
```

Summary

4. Adding a React Frontend to Complete MERN

Defining the skeleton application frontend

 Folder and file structure

Setting up for React development

 Configuring Babel and Webpack

 Babel

 Webpack

 Loading Webpack middleware for development

 Loading bundled frontend code

 Serving static files with Express

 Updating the template to load a bundled script

 Adding React dependencies

 React

 React Router

 Material-UI

Rendering a home page view ;

 Entry point at main.js

 Root React component

 Customizing the Material-UI theme

 Wrapping the root component with ThemeProvider and BrowserRouter

 er

 Marking the root component as hot-exported

 Adding a home route to MainRouter

 The Home component

 Imports

 Style declarations

 Component definition

 Bundling image assets

 Running and opening in the browser

Integrating backend APIs

 Fetch for user CRUD

 Creating a user

 Listing users

```
    Reading a user profile
    Updating a user's data
    Deleting a user
    Fetch for the auth API
        Sign-in
        Sign-out
    Adding auth in the frontend
        Managing auth state
            Saving credentials
            Retrieving credentials
            Deleting credentials
        The PrivateRoute component
    Completing the User frontend&#xA0;;
        The Users component
        The Signup component&#xA0;
        The Signin component
        The Profile component
        The EditProfile component
        The DeleteUser component
            Validating props with PropTypes&#xA0;
        The&#xA0; Menu component
    Implementing basic server-side rendering
        Modules for server-side rendering
        Generating CSS and markup
        Sending a template with markup and CSS
        Updating template.js
        Updating App.js
        Hydrate instead of render
    Summary
```

5. Growing the Skeleton into a Social Media Application

```
    Introducing MERN Social
    Updating the user profile
        Adding an about description
        Uploading a profile photo
            Updating the user model to store a photo in MongoDB
            Uploading a photo from the edit form
        File input with Material-UI
```

```
    Form submission with the file attached
    Processing a request containing a file upload
    Retrieving a profile photo
        Profile photo URL
        Showing a photo in a view
    Following users in MERN Social
        Following and unfollowing
            Updating the user model
            Updating the userByID controller method
            Adding APIs to follow and unfollow
            Accessing the follow and unfollow APIs in views
            Follow and unfollow buttons
                The FollowProfileButton component
                Updating the Profile component
            Listing followings and followers
                Making a FollowGrid component
            Finding people to follow
                Fetching users not followed
                The FindPeople component
    Posting on MERN Social
        Mongoose schema model for Post
        The Newsfeed component
        Listing posts
            Listing posts in Newsfeed
                Newsfeed API for posts
                Fetching Newsfeed posts in the view
            Listing posts by user in Profile
                API for posts by a user
                Fetching user posts in the view
        Creating a new post
            Creating the&#xA0;post API
            Retrieving a post's photo
            Fetching the create post API in the view
            Making the NewPost component
    The Post component
        Layout
            Header
```

- Content
- Actions
- Comments
- Deleting a post

Interacting with Posts

- Likes
 - The Like API
 - The Unlike API
 - Checking if a post has been liked and counting likes
 - Handling like clicks
- Comments
 - Adding a comment
 - The Comment API
 - Writing something in the view
 - Listing comments
 - Deleting a comment
 - The Uncomment API
 - Removing a comment from the view
 - Comment count update

Summary

-
- 3. Developing Web Applications with MERN**
 - 6. Building a Web-Based Classroom Application**

- Introducing MERN Classroom
- Updating the user with an educator role
 - Adding a role to the user model
 - Updating the EditProfile view
 - Rendering an option to teach
- Adding courses to the classroom
 - Defining a Course model
 - Creating a new course
 - The create course API
 - Fetching the create API in the view
 - The NewCourse component
 - Listing courses by educator
 - The list course API
 - Fetching the list API in the view
 - The MyCourses component

Display a course
A read course API
The Course component

Updating courses with lessons
Storing lessons
Adding new lessons
Adding a lesson API
The NewLesson component
Displaying lessons

Editing a course;
Updating the course API
The EditCourse component;
Updating lessons
Editing lesson details
Moving the lessons to rearrange the order
Deleting a lesson

Deleting a course
The delete course API
The DeleteCourse component

Publishing courses
Implementing the publish option
Publish button states
Confirm to publish

Listing published courses
The published courses API
The Courses component

Enrolling on courses;
Defining an Enrollment model
The create Enrollment API
The Enroll component
The Enrolled Course view
The read enrollment API
The Enrollment component

Tracking progress and enrollment stats
Completing lessons
Lessons completed API
Completed lessons from the view

- Listing all enrollments for a user
 - The list of enrollments API
 - The Enrollments component
- Enrollment stats
 - The enrollment stats API
 - Displaying enrollment stats for a published course
- Summary

7. Exercising MERN Skills with an Online Marketplace

- Introducing the MERN Marketplace app
- Allowing users to be sellers
 - Updating the user model
 - Updating the Edit Profile view
 - Updating the menu
- Adding shops to the marketplace
 - Defining a Shop model
 - Creating a new shop
 - The create shop API
 - Fetching the create API in the view
 - The NewShop component
 - Listing shops
 - Listing all shops
 - The shops list API
 - Fetch all shops for the view
 - The Shops component
 - Listing shops by owner
 - The shops by owner API
 - Fetch all shops owned by a user for the view
 - The MyShops component
 - Displaying a shop
 - The read a shop API
 - The Shop component;
 - Editing a shop
 - The edit shop API
 - The EditShop component
 - Deleting a shop
 - The delete shop API
 - The DeleteShop component

- Adding products to shops
 - Defining a Product model
 - Creating a new product
 - The create product API
 - The NewProduct component
 - Listing products
 - Listing by shop
 - The products by shop API
 - Products component for buyers
 - MyProducts component for shop owners
 - Listing product suggestions
 - Latest products
 - Related products
 - The Suggestions component
 - Displaying a product
 - Read a product API
 - Product component
 - Editing and deleting a product
 - Edit
 - Delete
- Searching for products with name and category
 - The categories API
 - The search products API
 - Fetch search results for the view
 - The Search component
 - The Categories component
- Summary

8. Extending the Marketplace for Orders and Payments

- Introducing cart, payments, and orders in the MERN Marketplace
- Implementing a shopping cart
 - Adding to the cart
 - Cart icon in the menu
 - The cart view
 - The CartItems component
 - Retrieving cart details
 - Modifying quantity
 - Removing items

```
    Showing the total price
    Option to check out

Using Stripe for payments
    Stripe-connected account for each seller
        Updating the user model
        Button to connect with Stripe
        The StripeConnect component
        The stripe auth update API
        Stripe Card Elements for checkout
        Stripe Customer for recording card details
            Updating the user model
            Updating the user controller
                Creating a new Stripe Customer
                Updating an existing Stripe Customer
            Creating a charge for each product that's processed

Integrating the checkout process
    Initializing checkout details
        Customer information
        Delivery address
    Placing an order
        Using Stripe Card Elements
        The CardElement component
        Adding a button to place an order
        Empty cart
        Redirecting to the order view

Creating a new order
    Defining an Order model
        The Order schema
    The CartItem schema
    Create order API
        Decrease product stock quantity
        Create controller method

Listing orders by shop
    The list by shop API
    The ShopOrders component
        List orders
    The ProductOrderEdit component
```

- Handling actions to cancel a product order
- Handling the action to process charge for a product
- Handling the action to update the status of a product
- APIs for products ordered
 - Get status values
 - Update order status
 - Cancel product order
 - Process charge for a product
- Viewing single-order details
- Summary

9. Adding Real-Time Bidding Capabilities to the Marketplace

- Introducing real-time bidding in the MERN Marketplace
- Adding auctions to the marketplace
 - Defining an Auction model
 - Creating a new auction
 - The create auction API
 - Fetching the create API in the view
 - The NewAuction component
 - Listing auctions
 - The open Auctions API
 - The Auctions by bidder API
 - The Auctions by seller API
 - The Auctions component
 - Editing and deleting auctions
 - Updating the list view
 - Edit and delete auction APIs
 - Displaying the auction view
 - The read auction API
 - The Auction component
 - Adding the Timer component
 - Implementing real-time bidding with Socket.IO
 - Integrating Socket.IO
 - Placing bids
 - Adding a form to enter a bid
 - Receiving a bid on the server
 - Displaying the changing bidding history
 - Updating the view state with a new bid

Rendering the bidding history

Summary

4. Advancing to Complex MERN Applications

10. Integrating Data Visualization with an Expense Tracking Application

Introducing the MERN Expense Tracker

Adding expense records

Defining an Expense model;

Creating a new expense record

The create expense API

The NewExpense component

Listing expenses

The expenses by user API

The Expenses component

Searching by date range

Rendering expenses

Modifying an expense record

Rendering the edit form and delete option

Editing and deleting an expense in the backend

Visualizing expense data over time

Summarizing recent expenses

Previewing expenses in the current month

The current month preview API

Rendering the preview of current expenses

Tracking current expenses by category

The current expenses by category API

Rendering an overview of expenses per category;

Displaying expense data charts

A month's expenses in a scatter plot

The scatter plot data API

The MonthlyScatter component

Total expenses per month in a year

The yearly expenses API

The YearlyBar component

Average expenses per category in a pie chart

The average expenses by category API

The CategoryPie component

Summary

11. Building a Media Streaming Application

- Introducing MERN Mediastream
- Uploading and storing media
 - Defining a Media model
 - Using MongoDB GridFS to store large files
 - Creating a new media post
 - The create media API
 - The NewMedia component
- Retrieving and streaming media
 - The video API
 - Using a React media player to render the video
- Listing media
 - The MediaList component
 - Listing popular media
 - Listing media by users
- Displaying, updating, and deleting media
 - Displaying media
 - The read media API
 - The Media component
 - Updating media details
 - The media update API
 - The media edit form
 - Deleting media
 - The delete media API
 - The DeleteMedia component

Summary

12. Customizing the Media Player and Improving SEO

- Adding a custom media player to MERN Mediastream
 - The play media page
 - The component structure
- Listing related media
 - The related media list API
 - The RelatedMedia component
- The PlayMedia component
- Customizing the media player
 - Updating the Media component
 - Initializing the media player

- Custom media controls
 - Play, pause, and replay
 - Play next
 - Loop when a video ends
 - Volume control
 - Progress control
 - Fullscreen
 - Played duration
- Autoplaying related media
 - Toggling autoplay
 - Handling autoplay across components
 - Updating the state when a video ends in MediaPlayer
- Server-side rendering with data
 - Adding a route configuration file
 - Updating SSR code for the Express server
 - Using route configuration to load data
 - Isomorphic-fetch
 - Absolute URLs
 - Injecting data into the React app
 - Applying server-injected data to client code
 - Passing data props to PlayMedia from MainRouter
 - Rendering received data in PlayMedia
 - Checking the implementation of SSR with data
 - Testing in Chrome
 - Loading a page with JavaScript enabled
 - Disabling JavaScript from settings
 - PlayMedia view with JavaScript blocked
 - Summary

13. Developing a Web-Based VR Game

- Introducing the MERN VR Game
- Game features
- Getting started with React 360
 - Setting up a React 360 project
- Key concepts for developing the VR game
 - Equirectangular panoramic images
 - 3D position – coordinates and transforms
 - 3D coordinate system

- Transforming 3D objects
- React 360 components
 - Core components
 - View
 - Text
 - Components for the 3D VR experience
 - Entity
 - VrButton
- The React 360 API
 - Environment
 - Native modules
 - AudioModule
 - Location
 - StyleSheet
 - VrHeadModel
 - Loading assets
- React 360 input events
- Defining game details
 - Game data structure
 - Details of VR objects
 - Static data versus dynamic data
- Building the game view in React 360
 - Updating client.js and mounting to Location
 - Defining styles with StyleSheet
 - World background
 - Adding 3D VR objects
 - Interacting with VR objects
 - Rotating a VR object
 - Animation with requestAnimationFrame
 - Clicking on the 3D objects
 - Collecting the correct object on click
- Game completed state
- Bundling for production and integration with MERN
 - Bundling React 360 files
 - Integrating with a MERN application
 - Adding the React 360 production files
 - Updating references in index.html

Trying out the integration

Summary

14. Making the VR Game Dynamic using MERN

Introducing the dynamic MERN VR Game application

Defining a Game model

Exploring the game schema

Specifying the VR object schema

Validating array length in the game schema

Implementing game CRUD APIs

Creating a new game

Listing all games

Listing games by the maker

Loading a game

Editing a game

Deleting a game;

Adding a form for creating and editing games

Making a new game

Updating the menu

The NewGame component

Editing the game

The EditGame component

Implementing the GameForm component

Inputting simple game details

Modifying arrays of VR objects

Iterating and rendering the object details form

Adding a new object to the array

Removing an object from the array

Handling the object detail change

The VRObjectForm component

Adding the game list views

Rendering lists of games

The GameDetail component

Playing the VR game

Implementing the API to render the VR game view

Updating the game code in React 360

Getting the game ID from a link

Fetching the game data with the load game API

Bundling and integrating the updated code

Summary

5. Going Forward with MERN

15. Following Best Practices and Developing MERN Further

Separation of concerns with modularity

Revisiting the application folder structure

Server-side code

Client-side code

Adding CSS styles

External style sheets

Inline styles

JavaScript Style Sheets (JSS)

Selective server-side rendering with data

When is server-side rendering with data relevant?

Using stateful versus pure functional components

Stateful React components with ES6 classes or Hooks

Stateless React components as pure functions

Designing the UI with stateful components and stateless functional components

Using Redux or Flux

Enhancing security

JSON web tokens – client-side or server-side storage

Securing password storage

Writing test code

Testing tools for full-stack JavaScript projects

Static analysis with ESLint

End-to-end testing with Cypress

Comprehensive testing with Jest

Adding a test to the MERN Social application

Installing the packages

Defining the script to run tests

Adding a tests folder

Adding the test

Generating a snapshot of the correct Post view

Running and checking the test

Optimizing the bundle size

Code splitting

[Dynamic import\(\)](#)

[Extending the applications](#)

[Extending the server code](#)

[Adding a model](#)

[Implementing the APIs](#)

[Adding controllers](#)

[Adding routes](#)

[Extending the client code](#)

[Adding the API fetch methods](#)

[Adding components](#)

[Loading new components](#)

[Summary](#)

[Other Books You May Enjoy](#)

[Leave a review - let other readers know what you think](#)

Preface

This book explores the development of full-stack JavaScript web applications by combining the power of React with industry-tested server-side technologies, such as Node.js, Express, and MongoDB. The JavaScript landscape has been growing rapidly for some time now. With an abundance of options and resources available in relation to full-stack JavaScript web applications, it is easy to get lost when you need to choose from these frequently changing entities, learn about them, and make them work together to build your own web applications. In an attempt to address this pain point, this book adopts a practical approach to help you set up and build a diverse range of working applications using the popular MERN stack.

Who this book is for

This book is for JavaScript developers who may have worked with React but have minimal experience with full-stack development involving Node.js, Express, and MongoDB.

What this book covers

[Chapter 1](#), *Unleashing React Applications with MERN*, introduces the MERN stack technologies and the applications that will be developed in this book. We will discuss developing web applications with React, Node.js, Express, and MongoDB.

[Chapter 2](#), *Preparing the Development Environment*, helps you to set up the MERN stack technologies for development. We will explore essential development tools; install Node.js, MongoDB, Express, React, and any other required libraries; and then run code to check the setup.

[Chapter 3](#), *Building a Backend with MongoDB, Express, and Node*, implements the backend of a skeleton MERN application. We will build a standalone server-side application with MongoDB, Express, and Node.js that stores user details and has APIs for user authentication and CRUD operations.

[Chapter 4](#), *Adding a React Frontend to Complete MERN*, completes the MERN skeleton application by integrating a React frontend. We will implement a working frontend with React views for interacting with the user CRUD operations and auth APIs on the server.

[Chapter 5](#), *Growing the Skeleton into a Social Media Application*, builds a social media application by extending the skeleton application. We will explore the capabilities of the MERN stack by implementing social media features, such as post sharing, liking, commenting, following friends, and an aggregated newsfeed.

[Chapter 6](#), *Building a Web-Based Classroom Application*, focuses on building a simple online classroom application by extending the MERN stack skeleton application. This classroom application will support multiple user roles, the addition of course content and

lessons, student enrollments, progress tracking, and course enrollment statistics.

[Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*, utilizes the MERN stack technologies to develop basic features in an online marketplace application. We will implement buying-and selling-related features with support for seller accounts, product listings, and product search by category.

[Chapter 8](#), *Extending the Marketplace for Orders and Payments*, focuses on extending the online marketplace we built in the previous chapter by implementing capabilities for buyers to add products to a shopping cart, checkout, and place orders, and for sellers to manage these orders and have payments processed from the marketplace application. We will also integrate Stripe to collect and process payments.

[Chapter 9](#), *Adding Real-Time Bidding Capabilities to the Marketplace*, focuses on teaching you how to use the MERN stack technologies, along with Socket.IO, to easily integrate real-time behavior in a full-stack application. We will do this by incorporating an auctioning feature with real-time bidding capabilities in the MERN marketplace application.

[Chapter 10](#), *Integrating Data Visualization with an Expense Tracking Application*, focuses on using MERN stack technologies along with Victory—a charting library for React—to easily integrate data visualization features in a full-stack application. We will extend the MERN skeleton application to build an expense tracking application that will incorporate data processing and visualization features for expense data recorded by a user over time.

[Chapter 11](#), *Building a Media Streaming Application*, focuses on extending the MERN skeleton application to build a media uploading and streaming application using MongoDB GridFS. We will start by building a basic media streaming application, allowing registered

users to upload video files that will be stored on MongoDB and streamed back so that viewers can play each video in a simple React media player.

[Chapter 12](#), *Customizing the Media Player and Improving SEO*, upgrades the media viewing capabilities of our media application with a custom media player and autoplay media list. We will implement customized controls on the default React media player, add a playlist that can be autoplayed, and improve SEO for the media details by adding selective server-side rendering with data for just the media detail view.

[Chapter 13](#), *Developing a Web-Based VR Game*, uses React 360 to develop a three-dimensional **virtual reality (VR)**-infused game for the web. We will explore the three-dimensional and VR capabilities of React 360 and build a simple web-based VR game.

[Chapter 14](#), *Making the VR Game Dynamic Using MERN*, is where you will build a dynamic VR game application by extending the MERN skeleton application and integrating React 360. We will implement a game data model that allows users to create their own VR games and incorporate the dynamic game data with the game developed using React 360.

[Chapter 15](#), *Following Best Practices and Developing MERN Further*, reflects on the lessons learned in previous chapters and suggests improvements for further MERN-based application development. We will expand on some of the best practices already applied, such as modularity in the app structure, other practices that should be applied, such as writing test code, and possible improvements, such as optimizing bundle size.

To get the most out of this book

This book assumes that you have familiarity with basic web-based technologies, working knowledge of programming constructs in JavaScript, and a general idea of how React applications work. As you go through the book, you will uncover how these concepts come together when building fully-fledged web applications with React 16.13.1, Node.js 13.12.0, Express 4.17.1, and MongoDB 4.2.5.

In order to maximize your learning experience while reading through the chapters, it is recommended that you run the associated application code in parallel, maintaining the specified package versions and using the relevant instructions provided in each chapter.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the Support tab.
3. Click on Code Downloads.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
addItem(item, cb) {
  let cart = []
  if (typeof window !== "undefined") {
    if (localStorage.getItem('cart')) {
      cart = JSON.parse(localStorage.getItem('cart'))
    }
    cart.push({
      product: item,
      quantity: 1,
      shop: item.shop._id
    })
    localStorage.setItem('cart', JSON.stringify(cart))
    cb()
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<Grid container spacing={24}>
  <Grid item xs={6} sm={6}>
    <CartItems checkout={checkout}
               setCheckout={showCheckout}/>
  </Grid>
  {checkout &&
    <Grid item xs={6} sm={6}>
      <Checkout/>
    </Grid>}
</Grid>
```

Any command-line input or output is written as follows:

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

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

TIP

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

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

For more information about Packt, please visit packt.com.

Getting Started with MERN

In this part, we get an introduction to MERN and an overview of its different components. Additionally, you will develop an understanding of how to configure your development environment correctly, before you start developing full-fledged web applications with these technologies.

This section comprises the following chapters:

- [Chapter 1](#), *Unleashing React Applications with MERN*
- [Chapter 2](#), *Preparing the Development Environment*

Unleashing React Applications with MERN

React may have opened up new frontiers for frontend web development and changed the way we program JavaScript user interfaces, but we still need a solid backend to build a complete web application. While there are myriad options when selecting backend technologies, the benefits and appeal of using a full JavaScript stack are undeniable, especially when there are robust and widely adopted backend technologies such as Node, Express, and MongoDB. Combining the potential of React with these industry-tested, server-side technologies creates a diverse array of possibilities when developing real-world web applications. This book guides you through Setting up **MERN** (short for **MongoDB, Express.js, React, and Node.js**) -based web development, to building real-world web applications of varying complexities.

Before diving into the development of these web applications, we are going to answer the following questions in this chapter, so you can use this book effectively to acquire full-stack development skills, and also understand the context behind choosing the MERN stack to build your applications:

- What is new in the second edition?
- How is this book organized to help master MERN?
- What is the MERN stack?
- Why is MERN relevant today?
- When is MERN a good fit for developing web apps?

What is new in the second edition?

MERN stack technologies along with the whole full-stack development ecosystem are continuously growing and improving with increased adoption and usage in the industry. In this edition, we take these new developments into account and update all the applications and corresponding code bases from the first edition.

We use the latest versions and conventions of each technology, library, and module needed for both development-related setup and feature implementations. Additionally, we highlight the use of new features from these technology upgrades such as React Hooks, and JavaScript features such as `async/await`.

In order to showcase even more possibilities with the MERN stack, we updated the existing marketplace application to add a more advanced feature such as real-time bidding. We also add two new projects, a web-based classroom application and an expense tracking application with data visualization features.

To better understand the content and concepts covered throughout the book, we expand on explanations and provide leads to the latest resources that may help you get a deeper grasp and improve your learning experience.

Besides covering the latest updates to MERN technologies and providing elaborate explanations, the concepts and projects in this book are organized to help you learn from easy to advanced topics with the flexibility to start at any project of your choosing. In the next section, we will discuss the structure of the book and how you can utilize it based on your preference and experience level.

Book structure

This book aims to help JavaScript developers who have zero-to-some experience with the MERN stack to set up and start developing web applications of varying complexity. It includes guidelines for building out and running the different applications supplemented with code snippets and explanations of key concepts.

The book is organized into five parts, progressing from basic to advanced topics, taking you on a journey of building MERN from the ground up, then using it to develop different applications with simple to complex features, while demonstrating how to extend the capabilities of the MERN stack based on an application's requirements.

Getting started with MERN

[Chapter 1](#), *Unleashing React Applications with MERN*, and [Chapter 2](#), *Preparing the Development Environment*, set the context for developing web applications in a MERN stack and guide you through setting up your development environment.

Building MERN applications from the ground up

[Chapter 3](#), *Building a Backend with MongoDB, Express, and Node*, and [Chapter 4](#), *Adding a React Frontend to Complete MERN*, show you how to bring the MERN stack technologies together to form a skeleton web application with minimal and basic features. [Chapter 5](#), *Growing the Skeleton into a Social Media Application*, demonstrates how this skeletal MERN application can act as a base and be easily extended to build a simple social media platform. This ability to extend and customize the base application will be employed with the other applications developed in the rest of this book.

Developing web applications with MERN

In this part, you will become more familiar with the core attributes of a MERN stack web application by building out two real-world applications—a web-based classroom application in [Chapter 6](#), *Building a Web-Based Classroom Application*, and a feature-rich online marketplace in [Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*, [Chapter 8](#), *Extending the Marketplace for Orders and Payments*, and [Chapter 9](#), *Adding Real-Time Bidding Capabilities to the Marketplace*.

Advancing to complex MERN applications

[Chapter 10](#), *Integrating Data Visualization with an Expense Tracking Application*, [Chapter 11](#), *Building a Media Streaming Application*, [Chapter 12](#), *Customizing the Media Player and Improving SEO*, [Chapter 13](#), *Developing a Web-Based VR Game*, and [Chapter 14](#), *Making the VR Game Dynamic Using MERN*, demonstrate how this stack can be used to develop applications with more complex and immersive features, such as data visualization, media streaming, and **virtual reality (VR)** using React 360.

Going forward with MERN

Finally, [Chapter 15](#), *Following Best Practices and Developing MERN Further*, wraps up the preceding chapters and applications developed by expanding on the best practices to follow to make successful MERN applications, suggesting improvements and further developments.

You may choose to use the book out of the prescribed order based on your experience level and preference. A developer who is new to MERN can follow the path set out in the book. For a more seasoned JavaScript developer, the chapters in the *Building MERN from the Ground up* section would be a good place to start setting up the base application, then pick any of the six applications to build and extend.

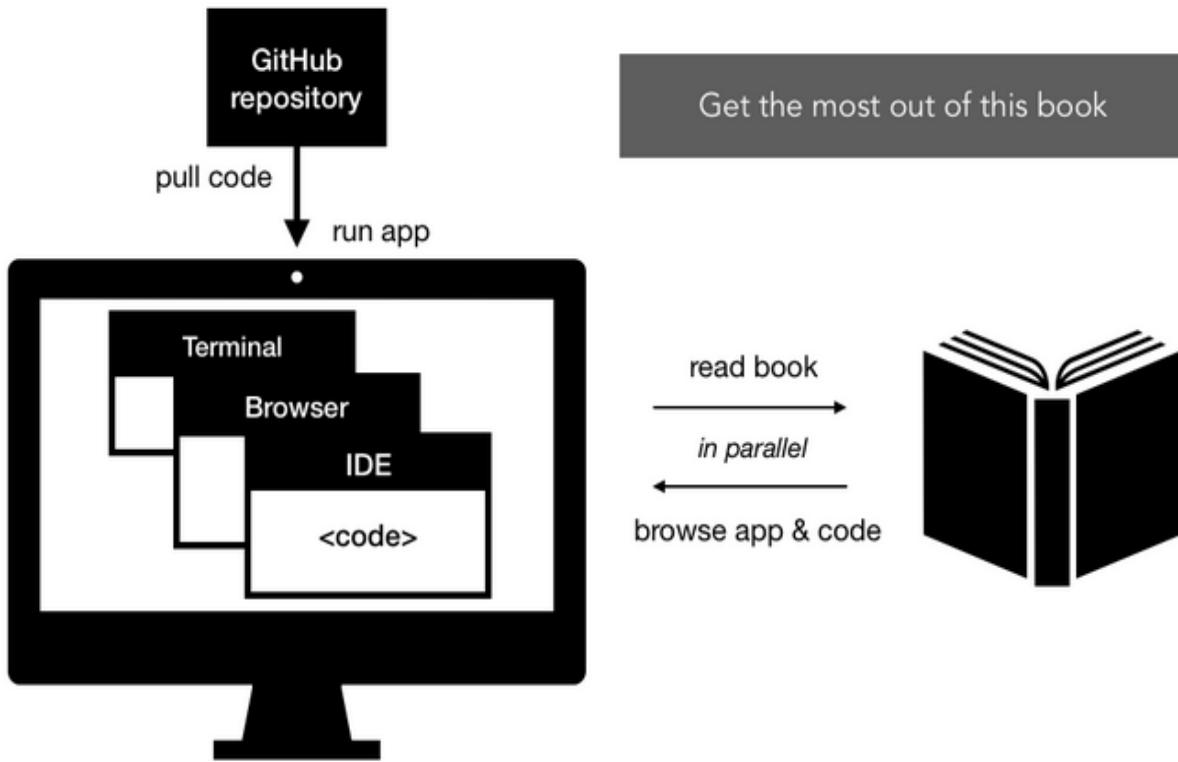
This structure is set out with the intention to enable hands-on learning for developers from varying backgrounds. In order to maximize this intent, we recommend a practical approach for following along with the material in the book, as described in more detail in the next section.

Getting the most out of this book

The content in this book is practical-oriented and covers the implementation steps, code, and concepts relevant to building each MERN application. However, most of the code explanations will refer to specific snippets from files that may contain more lines of code, making up the complete and working application code.

Put simply, it is highly recommended that, rather than attempting to just read through the chapters, you should run the relevant code in parallel, and browse through the application features while following the explanations in the book.

Chapters that discuss code implementations will point to the GitHub repositories containing the complete code with instructions on how to run the code. You can pull the code, install it, and then run it before reading through the chapter:



You should consider the recommended steps outlined here to follow the implementations in this book:

- Before diving into the implementation details discussed in the chapter, pull the code from the relevant GitHub repository.
- Follow the instructions with the code to install and run the application.
- Browse the features of the running application while reading the feature descriptions in the relevant chapter.
- With the code running in development mode and also open in the editor, refer to the steps and explanations in the book to get a deeper understanding of the implementations.

This book aims to provide a quick onboarding with the working code for each application. You can experiment with, improve, and extend this code as desired. For an active learning experience, you are encouraged to refactor and modify the code while following the book. In some examples, the book chooses verbose code over succinct and

cleaner code because it is easier to reason about for newcomers. In some other implementations, the book sticks with more widely used and traditional conventions over modern and upcoming JavaScript conventions. This is done to minimize disparity when you refer to online resources and documentation while researching the discussed technologies and concepts on your own. These instances, where the code in the book can be updated, serve as good opportunities to explore and grow skills beyond what is covered in the book.

You should now have an overall idea of what to expect in this book and how you can utilize its content and structure to the fullest as we move on to discussing the specifics of the MERN stack and start uncovering its potential.

The MERN stack

MongoDB, Express, React, and Node are all used in tandem to build web applications and make up the MERN stack. In this lineup, Node and Express bind the web backend together, MongoDB serves as the NoSQL database, and React makes the frontend that the user sees and interacts with.

All four of these technologies are free, open source, cross-platform, and JavaScript-based, with extensive community and industry support. Each technology has a unique set of attributes, which, when integrated together, make a simple but effective full JavaScript stack for web development.

Since these are independent technologies, it is also important to recognize these as moving parts in your project that need to be configured, combined, and extended with additional parts to meet the specific requirements of your project. Even if you are not an expert in all the technologies in this stack, you need familiarity with each and an understanding of how these can work together.

Node

Node was developed as a JavaScript runtime environment built on Chrome's V8 JavaScript engine. Node made it possible to start using JavaScript on the server side to build a variety of tools and applications beyond previous use cases that were limited to being within a browser.

Node has an event-driven architecture capable of asynchronous, non-blocking **I/O** (short for **Input/Output**). Its unique non-blocking I/O model eliminates the waiting approach to serving requests. This allows you to build scalable and lightweight real-time web applications that can efficiently handle many requests.

Node's default package management system, the Node Package Manager or `npm`, comes bundled with the Node installation. `npm` gives you access to hundreds of thousands of reusable Node packages built by developers all over the world and boasts that it is currently the largest ecosystem of open source libraries in the world.



Learn more about Node at <https://nodejs.org/en/>, and browse through the available `npm` registry at <https://www.npmjs.com/>.

However, `npm` isn't the only package management system at your disposal. Yarn is a newer package manager developed by Facebook and has been gaining popularity in recent years. It can be used as an alternative to `npm`, with access to all the same modules from the `npm` registry and more features that are not yet available with `npm`.



Learn more about Yarn and its features at <https://yarnpkg.com>.

Node will enable us to build and run complete full-stack JavaScript applications. However, to implement an extensible server-side application with web application-specific features such as API routing, we will use the Express module on top of Node.

Express

Express is a simple server-side web framework for building web applications with Node. It complements Node with a layer of rudimentary web application features that provide HTTP utility methods and middleware functionality.



*In general terms, **middleware** functionality in any application enables different components to be added on to work together. In the specific context of server-side web application frameworks, middleware functions have access to the HTTP request-response pipeline, which means access to request-response objects and also the next middleware functions in the web application's request-response cycle.*

In any web application developed with Node, Express can be used as an API routing and middleware web framework. It is possible to insert almost any compatible middleware of your choice into the request handling chain, in almost any order, making Express very flexible to work with.



Find out what is possible with Express.js at expressjs.com.

In the MERN-based applications that we will develop, Express can be used to handle API routing on the server side, serve static files to the client, restrict access to resources with authentication integration, implement error handling, and, essentially, add on any middleware package that will extend the web application functionality as required.

A crucial functionality in any complete web application is the data storage system. The Express module does not define requirements or put restrictions on integrating databases with a Node-Express web application. Therefore, this gives you the flexibility to choose any database option, be it a relational database such as PostgreSQL or a NoSQL database such as MongoDB.

MongoDB

MongoDB is a top choice when deciding on a NoSQL database for any application. It is a document-oriented database that stores data in flexible, JSON-like documents. This means that fields can vary from document to document and data models can evolve over time in response to changing application requirements.

Applications that place a high priority on availability and scalability benefit from MongoDB's distributed architecture features. It comes with built-in support for high availability, horizontal scaling using sharding, and multi-data center scalability across geographic distributions.

MongoDB has an expressive query language, enabling ad hoc queries, indexing for fast lookups, and real-time aggregation that provides powerful ways to access and analyze data while maintaining performance even when data size grows exponentially.



Explore MongoDB's features and services at <https://www.mongodb.com/>.

Choosing MongoDB as the database for a Node and Express web application will make a fully JavaScript-based and standalone server-side application. This will leave you with the option to integrate a client-side interface that may be built with a compatible frontend library such as React to complete the full-stack application.

React

React is a declarative and component-based JavaScript library for building user interfaces. Its declarative and modular nature makes it easy for developers to create and maintain reusable, interactive, and complex user interfaces.

Large applications that display a lot of changing data can be fast and responsive if built with React, as it takes care of efficiently updating and rendering just the right user interface components when specific data changes. React does this efficient rendering with its notable implementation of a virtual DOM, setting React apart from other web user interface libraries that handle page updates with expensive manipulations directly in the browser's DOM.

Developing user interfaces using React also forces frontend programmers to write well-reasoned, modular code that is reusable and easier to debug, test, and extend.



Take a look at the resources on React at <https://reactjs.org/>.

Since all four technologies are JavaScript-based, these are inherently optimized for integration. However, how these are actually put together in practice to form the MERN stack can vary based on application requirements and developer preferences, making MERN customizable and extensible to specific needs. Whether this stack is a relevant option for your next full-stack web project not only depends on how well it can meet your requirements, but also on how it is currently faring in the industry and where these technologies are headed.

Relevance of MERN

JavaScript has come a long way since its inception, and it is ever-growing. MERN stack technologies have challenged the status quo and broken new ground for what is possible with JavaScript. But when it comes to developing real-world applications that need to be sustainable, is it a worthy choice? Some of the reasons that make a strong case for choosing MERN for your next web application are briefly outlined in the following sections.

Consistency across the technology stack

As JavaScript is used throughout, developers don't need to learn and change gears frequently to work with very different technologies. This also enables better communication and understanding across teams working on different parts of the web application.

Takes less time to learn, develop, deploy, and extend

Consistency across the stack also makes it easy to learn and work with MERN, reducing the overhead of adopting a new stack and the time to develop a working product. Once the working base of a MERN application is set up and a workflow established, it takes less effort to replicate, further develop, and extend any application.

Widely adopted in the industry

Organizations of all sizes have been adopting the technologies in this stack based on their needs because they can build applications faster, handle highly diverse requirements, and manage applications more efficiently at scale.

Community support and growth

Developer communities surrounding the very popular MERN stack technologies are quite diverse and are growing on a regular basis. With lots of people continuously using, fixing, updating, and willing to help grow these technologies, the support system will remain strong for the foreseeable future. These technologies will continue to be maintained, and resources are very likely to be available in terms of documentation, add-on libraries, and technical support.

The ease and benefits of using these technologies are already widely recognized. Because of the high-profile companies that continue adoption and adaptation, and the growing number of people contributing to the code bases, providing support, and creating resources, the technologies in the MERN stack will continue to be relevant for a long time to come.

In order to determine whether this widely adopted stack will meet the specific requirements of your project, you can explore the extent of feature implementations possible with this group of technologies. In the next section, we will highlight a few aspects of this stack and also several features of the book's example applications that demonstrate the diverse array of options that are available with these technologies.

Range of MERN applications

Given the unique features attributed to each technology, along with the ease of extending functionalities of this stack by integrating other technologies, the range of applications that can be built with this stack is actually quite vast.

These days, web applications are, by default, expected to be rich client apps that are immersive, interactive, and don't fall short on performance or availability. The grouping of MERN strengths makes it perfect for developing web applications that meet these very aspects and demands.

Moreover, novel and upcoming attributes of some of these technologies, such as low-level operation manipulation with Node, large file streaming capabilities with MongoDB GridFS, and VR features on the web using React 360, make it possible to build even more complex and unique applications with MERN.

It may seem reasonable to pick specific features in the MERN technologies and argue why they don't work for certain applications. However, given the versatile nature of how a MERN stack can come together and be extended, these concerns can be addressed in MERN on a case-by-case basis. In this book, we will demonstrate how to make such considerations when faced with specific requirements and demands in the application being built.

MERN applications developed in this book

To demonstrate the breadth of possibilities with MERN and how you can easily start building a web application with varying features, this book will showcase everyday-use web applications alongside complex and rare web experiences.

Social media platform

For the first MERN application, we will build a basic social media application inspired by Twitter and Facebook, as follows:

Newsfeed



Cecil McQueen

Share your thoughts ...



POST



Violet Bernstein

Fri Dec 22 2017

"Injustice anywhere is a threat to justice everywhere." - Martin Luther King, Jr.



3



1



Write something ...



Cecil McQueen

Truth!

Fri Dec 22 2017 |

Who to follow



Dominic Kotsopolis



FOLLOW



Daniel Nakamura



FOLLOW



Delbert Catessen



FOLLOW



Antonia Thatcher



FOLLOW



Jack Michaels



FOLLOW



Thomas Brogan



FOLLOW



Katherine Hemstridge



FOLLOW



Jamie Woolcraft



FOLLOW

This social media platform will implement simple features such as post sharing, liking and commenting, following friends, and an aggregated news feed.

Web-based classroom application

Remote or online learning is a common practice these days, with both instructors and students utilizing internet connectivity to teach and learn over online platforms. We will implement a simple web-based classroom application using MERN, which will look like the following screenshot:

The screenshot shows a web-based classroom application with the following features:

- Header:** "MERN Classroom" with a home icon, "TEACH" button, "MY PROFILE" link, and "SIGN OUT" link.
- Courses you are enrolled in:** A row of four course cards:
 - Node.js Fundamentals
 - React for Beginners
 - Learn Git
 - JavaScript Pro
- All Courses:** A grid of four course cards:
 - Probability & Statistics (Mathematics) - ENROLL button
 - Introduction to Machine Learning (Computer Science) - ENROLL button
 - Search Engine Optimization (Digital Marketing) - ENROLL button
 - Basics of Microeconomics (Economics) - ENROLL button

This classroom will have features that allow instructors to add courses with lessons, while students can enroll in these courses and

track their progress.

Online marketplace

All sorts of e-commerce web applications are abundant on the internet, and they will not go out of style anytime soon. Using MERN, we will build a comprehensive online marketplace application with basic-to-advanced e-commerce features. The following screenshot shows the completed home page of the marketplace with product listings:

Select category

Search products

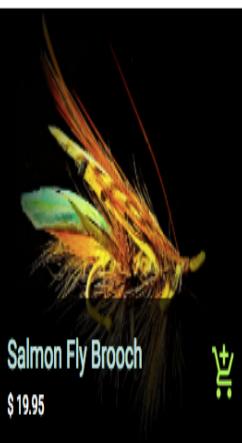
Jewelry

brooch



Beaded Rhino Brooch

\$27.99



Salmon Fly Brooch

\$19.95



Silver Owl Brooch

\$33

Latest Products

Converse All Star
Superheroes

Added on Mon Jan 08 2018

\$140



Joker Figurine



Added on Mon Jan 08 2018

\$51.45

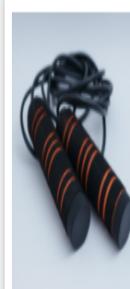


Darth Vader Figurine



Added on Mon Jan 08 2018

\$19.99



Speed Skipping Rope



Added on Mon Jan 08 2018

\$7.55



Explore by category

Collectibles

Tools

Shoes

Sports



Old Ford Car Model

\$20



Wonder Woman Figurine

\$11.45



Tardis Figurine

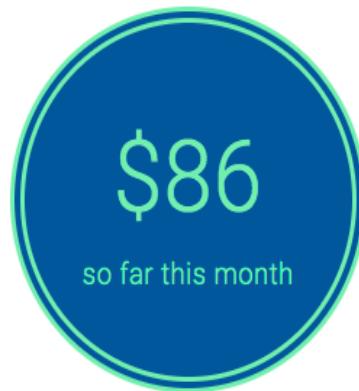
\$14

The features of this marketplace application will cover aspects such as support for seller accounts, product listings, a shopping cart for customers, payment processing, order management, and real-time bidding capabilities.

Expense tracking application

Adding data visualization to any data-intensive application can boost its value considerably. We will extend MERN with an expense-tracking application to demonstrate how you can incorporate data visualization features, including graphs and charts, in a full-stack MERN application. The following screenshot shows the completed home page of the expense tracker application with an overview of the user's current expenses:

You've spent



\$15 today

\$5 yesterday

[See more](#)

Groceries

past average	this month	spent extra
\$38.5	\$57	\$18.5

Commute

past average	this month	saved
\$19.5	\$14	\$5.5

Eating Out

past average	this month	saved
\$15	\$15	\$0

With this application, users will be able to keep track of their day-to-day expenses. The application will add the expenses incurred over time. Then, the application will extract data patterns to give the users

a visual representation of how their expense habits fare as time progresses.

Media streaming application

To test out some advanced MERN capabilities, a more immersive application, such as a media streaming application, is the next pick. The following screenshot shows the home page view containing a list of popular videos added to this platform, which is inspired by features from Netflix and YouTube:

Popular Videos



Endeavour Lift-off

425 views

Space Exploration



Goldfinch pair

365 views

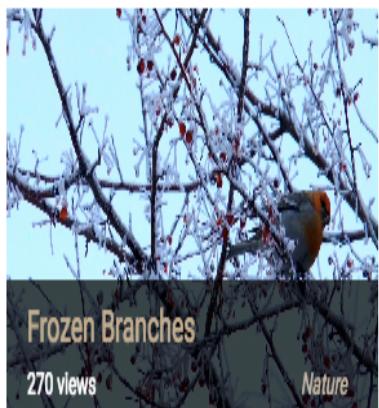
Nature



Ballerina

298 views

Music



Frozen Branches

270 views

Nature



Atlantis Lift-off

244 views

Space Exploration



Carousel Music Box

223 views

Music



Discovery Lift-off

201 views

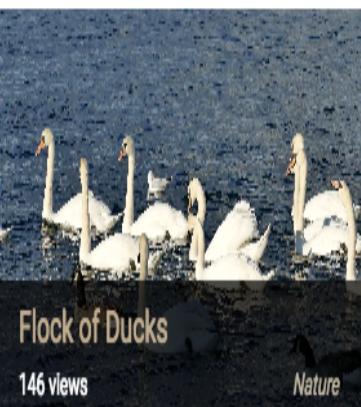
Space Exploration



White Music Carousel

182 views

Music



Flock of Ducks

146 views

Nature

In this media streaming application, we will implement content uploading and viewing capabilities with a media content upload feature for content providers, and real-time content streaming for viewers.

VR game for the web

With frameworks such as React 360, which is built on top of React, it is possible to apply web VR and 3D capabilities to React's user interfaces. We will explore how to create rare web experiences with React 360 in MERN by putting together a basic VR game application for the web, as shown in the following screenshot:

Space Exploration

by Jamie Woolcraft

If I could, then I surely would, cruise the Milky Way, and collect the celestial

PLAY GAME

Hidden Treasures

by Jack Michaels

Look at every nook and corner, hidden in plain sight is your treasure

PLAY GAME

Woodland Critters

by Antonia Thatcher

The woods awake, the trees rustle, and the little critters scurry and scamper all day

PLAY GAME

The Declutterer

by Thomas Brogan

Clutter, clutter everywhere! I wish to make it all disappear!

PLAY GAME

Users will be able to play the VR games and also make their own games with this web-based application. Each game will have animated VR objects placed across a 360 world, and players will have to find and collect these objects to complete the game.

Following along with the implementations for these diverse applications in the book will teach you how to combine, extend, and use MERN stack technologies to build full-stack web applications, and also reveal a diverse range of options for your own full-stack projects.

Summary

In this chapter, we discovered the context for developing web applications in the MERN stack and how this book will help you to develop with this stack. MERN stack projects integrate MongoDB, Express, React, and Node to build web applications. Each of the technologies in this stack has made relevant strides in the world of web development. These are widely adopted and continue to improve with the support of growing communities. It is possible to develop MERN applications with diverse requirements, ranging from everyday-use applications to more complex web experiences. The practical-oriented approach in this book can be used to grow MERN skills from basic to advanced, or for diving right into building more complex applications.

In the next chapter, we will start gearing up for MERN application development by learning how to set up the development environment with each MERN stack technology, and also write code for a MERN starter application to ensure the setup on your system is correct.

Preparing the Development Environment

Before building applications with the MERN stack, we first need to prepare the development environment with each technology, and also with tools to aid development and debugging. Working with this stack requires that you make different technologies and tools work well together, and given the many options and resources available on this topic, it can seem like a daunting task to figure out how it all comes together. This chapter guides you through the workspace options, the essential development tools, how to set up the MERN technologies in your workspace, and how to check this setup with actual code.

We are going to cover the following topics:

- Selecting development tools
- Setting up MERN stack technologies
- Checking your development setup

Selecting development tools

There are plenty of options available when it comes to selecting basic development tools such as text editors or IDEs, version control software, and even the development workspace itself. In this section, we will go over the options and recommendations that are relevant to web development with the MERN stack so you can make informed decisions when selecting these tools based on individual preferences.

Workspace options

Developing on a local machine is a common practice among programmers, but with the advent of good cloud and remote development services, such as AWS Cloud9 (<https://aws.amazon.com/cloud9/?origin=c9io>) and Visual Studio Code's Remote Development extension (<https://code.visualstudio.com/docs/remote>), you can set up your local workspace with MERN technologies (and this will be assumed to be the case for the rest of the book), but you can also choose to run and develop the code in cloud services that are equipped for Node development.

Local and cloud development

You can choose to use both types of workspaces to enjoy the benefits of working locally without worrying about bandwidth/internet issues and to work remotely when you don't physically have your favorite local machine. To do this, you can use Git to version control your code, store your latest code on remote Git hosting services such as GitHub or BitBucket, and then share the same code across all your workspaces. On your workspaces, you can compose the code in an IDE of your choice from the many available options, some of which are discussed next.

IDE or text editors

Most cloud development environments will come integrated with source code editors, but for your local workspace, you can pick any based on your preference as a programmer and then customize it for MERN development. For example, the following popular options can each be customized as required:

- **Visual Studio Code** (<https://code.visualstudio.com/>): A feature-rich source code editor by Microsoft with extensive support for modern web application development workflow, including support for MERN stack technologies
- **Atom** (<https://atom.io/>): A free, open source text editor for GitHub that has many packages relevant to the MERN stack from other developers
- **Sublime Text** (<https://www.sublimetext.com/>): A proprietary, cross-platform text editor that also has many packages relevant to the MERN stack, along with support for JavaScript development
- **WebStorm** (<https://www.jetbrains.com/webstorm/>): A full-fledged JavaScript IDE by JetBrains, with support for MERN stack development

There are other editors at your disposal, but besides focusing on what each has to offer, it is important that you choose one that is right for you, enables a productive workflow, and also integrates well with the other tools necessary for web application development.

Chrome Developer Tools

Loading, viewing, and debugging the frontend is a very crucial part of the web development process. Chrome DevTools (<https://developers.google.com/web/tools/chrome-devtools>), which is a part of the Chrome browser, has many great features that allow debugging, testing, and experimenting with the frontend code and the look, feel, responsiveness, and performance of the UI. Additionally, the React Developer Tools extension is available as a Chrome plugin, and it adds React debugging tools to Chrome DevTools.

Utilizing tools like this in your development workflow can help you to understand the code better and to build your applications effectively. Similarly, integrating code version control with a tool such as Git can increase your productivity and efficiency as a developer.

Git

Any development workflow is incomplete without a version control system that enables tracking code changes, code sharing, and collaboration. Over the years, Git has become the leading version control system for many developers and is the most widely used distributed source code management tool. For code development in this book, Git will help primarily to track progress as we go through the steps to build each application.

Installation

To start using Git, first install it on your local machine or cloud development environment based on your system specifications. Instructions to download and install the latest version of Git, along with documentation on using Git commands, can be found at [https://g
it-scm.com/downloads](https://git-scm.com/downloads).

Remote Git hosting services

Cloud-based Git repository hosting services such as GitHub and BitBucket help share your latest code across workspaces and deployment environments, and also to back up your code. These services pack in a lot of useful features to help with code management and the development workflow. To get started, you can create an account and set up remote repositories for your code bases.

All these essential tools will help enrich your web development workflow and increase productivity. Once you've completed the necessary setup in your workspace according to the discussion in the next section, we'll move on and start building MERN applications.

Setting up MERN stack technologies

MERN stack technologies are being developed and upgraded as this book is being written, so for the work demonstrated throughout this book, we use the latest stable versions at the time of writing. Installation guidelines for most of these technologies are dependent on the system environment of your workspaces, so this section points to all relevant installation resources and also acts as a guide for setting up a fully functioning MERN stack.

MongoDB

MongoDB must be set up, running, and accessible to your development environment before any database features are added to MERN applications. At the time of writing, the current stable version of MongoDB is 4.2.0, and this version of the MongoDB Community Edition is used for developing the applications in this book. The rest of this section provides resources on how to install and run MongoDB.

Installation

You need to install and start MongoDB on your workspace before you can use it for development. The installation and startup process for MongoDB depends on your workspace specifications:

- Cloud development services will have their own instructions for installing and setting up MongoDB. For example, the how-to steps for AWS Cloud9 can be found at <https://docs.c9.io/docs/setup-a-database#mongodb>.
- The guide for MongoDB installation on your local machine is at <https://docs.mongodb.com/manual/administration/install-community>

After you have successfully installed MongoDB on your workspace and have it running, you can interact with it using the **mongo shell**.

Running the mongo shell

The mongo shell is an interactive tool for MongoDB that comes as a part of the MongoDB installation. It is a good place to start getting familiar with MongoDB operations. Once MongoDB is installed and running, you can run the mongo shell on the command line. In the mongo shell, you can use commands to query and update data and perform administrative operations.



You could also skip the local installation of MongoDB and instead deploy a MongoDB database in the cloud using MongoDB Atlas (<https://www.mongodb.com/cloud/atlas>). It is a global cloud database service that can be used to add fully managed MongoDB databases to modern applications.

The next core component of MERN development is Node.js, which will be necessary to complete the remaining MERN setup.

Node.js

Backend server implementation for the MERN applications relies on Node.js. At the time of writing, 13.12.0 is the latest stable Node.js version available, and the code in the book has also been tested with version 14.0.0 from the latest nightly builds. The version of Node.js you choose to download will come bundled with npm as the package manager. Depending on whether you choose npm or Yarn as the package manager for your MERN projects, you can install Node.js with or without npm.

Installation

Node.js can be installed via direct download, installers, or the Node Version Manager:

- You can install Node.js by directly downloading the source code or a pre-built installer specific to your workspace platform. Downloads are available at nodejs.org/en/download.
- Cloud development services may come with Node.js pre-installed, as AWS Cloud9 does, or will have specific instructions for adding and updating it.

To test if the installation was successful, you can open the command line and run `node -v` to see if it correctly returns the version number.

Node version management with nvm

If you need to maintain multiple versions of Node.js and npm for different projects, nvm is a useful command-line tool to install and manage different versions on the same workspace. You have to install nvm separately. Instructions for setup can be found at github.com/creationix/nvm.

With Node.js set up on your system, you can use a package manager such as npm or Yarn to start integrating the remaining parts of the MERN stack.

Yarn package manager

Yarn is a relatively new package manager for JavaScript dependencies, and it can be used as an alternative to npm. It is a fast, reliable, and secure dependency manager that provides a different range of additional features, including an offline mode for re-installation of packages without an internet connection and support for multiple package registries, such as npmjs.com and Bower.

We will use Yarn (v1.22.4) to manage Node modules and packages for the projects in this book. Before using Yarn, you will need to install it on your workspace. There are a number of ways to install Yarn depending on your operating system and its version.



To learn more about your options for installing Yarn on your workspace, visit <https://yarnpkg.com/lang/en/docs/install>.

Once Yarn is installed, it can be used to add the other necessary modules including Express and React.

Modules for MERN

The remaining MERN stack technologies are all available as Node.js package modules and can be added to each project using Yarn. These include key modules, such as React and Express, that are required to run each MERN application, and also modules that will be necessary during development. In this section, we list and discuss these modules, then see how to use the modules in a working project in the following section.

Key modules

To integrate the MERN stack technologies and run your applications, we will need the following modules:

- **React**: To start using React, we will need two modules:
 - `react`
 - `react-dom`
- **Express**: To use Express in your code, you will need the `express` module.
- **MongoDB**: To use MongoDB directly with Node applications, you need to add the driver, which is available as a Node module named `mongodb`.

These key modules will produce full-stack web applications, but we will need some additional modules to aid in the development and generation of the application code.

devDependency modules

To maintain consistency throughout the development of our MERN applications, we will use new JavaScript syntax from ES6 and higher versions in both client- and server-side implementations. As a consequence, and also to aid the development process, we will use the following additional modules to compile and bundle the code, and also to automatically reload the server and browser app as the code is updated during development:

- Babel modules are needed to convert ES6 and JSX to suitable JavaScript for all browsers. The modules needed to get Babel working are as follows:
 - `@babel/core`
 - `babel-loader`
 - for transpiling JavaScript files with Webpack
 - `@babel/preset-env` and `@babel/preset-react` to provide support for React, and the latest JavaScript feature
- Webpack modules will help bundle the compiled JavaScript, both for the client-side and the server-side code. Modules needed to get Webpack working are as follows:
 - `webpack.`
 - `webpack-cli` to run Webpack commands.
 - `webpack-node-externals` to ignore external Node.js module files when bundling in Webpack.
 - `webpack-dev-middleware` to serve the files emitted from Webpack over a connected server during the development of the code.
 - `webpack-hot-middleware` to add hot module reloading into an existing server by connecting a browser client to a Webpack server and receiving updates as code changes during development.
 - `nodemon` to watch server-side changes during development, so the server can be reloaded to put changes into effect.

- `react-hot-loader` for faster development on the client side. Every time a file changes in the React frontend, `react-hot-loader` enables the browser app to update without re-bundling the whole frontend code.
- `@hot-loader/react-dom` to enable hot-reloading support for React hooks. It essentially replaces the `react-dom` package of the same version, but with additional patches to support hot reloading.



Although `react-hot-loader` is meant to help the development flow, it is safe to install this module as a regular dependency rather than a devDependency. It automatically ensures that hot reloading is disabled in production and the footprint is minimal.

With the necessary MERN stack technologies and associated tools installed and ready for use, in the next section, we will use this toolset and write code that can confirm if your workspace is set up correctly to begin developing MERN-based web applications.

Checking your development setup

In this section, we will go through the development workflow and write code step by step to ensure that the environment is correctly set up to start developing and running MERN applications.

We will generate these project files in the following folder structure to run a simple setup project:

```
| mern-simplesetup/
| -- client/
|   --- HelloWorld.js
|   --- main.js
| -- server/
|   --- devBundle.js
|   --- server.js
| -- .babelrc
| -- nodemon.json
| -- package.json
| -- template.js
| -- webpack.config.client.js
| -- webpack.config.client.production.js
| -- webpack.config.server.js
```



The code discussed in this section is available on GitHub in the repository at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter02/mern-simple-setup>. You can clone this code and run it as you go through the code explanations in the rest of this chapter.

We will leave the configuration files in the root folder and organize the application code into client-side and server-side folders. The `client` folder will contain the frontend code and the `server` folder will contain the backend code. In the rest of this section, we will generate these files and implement both frontend and backend code to build a working full-stack web application.

Initializing package.json and installing Node.js modules

We will begin by using Yarn to install all the required modules. It is a best practice to add a `package.json` file in every project folder to maintain, document, and share the Node.js modules being used in the MERN application. The `package.json` file will contain meta-information about the application, as well as list the module dependencies.

Perform the steps outlined in the following points to generate a `package.json` file, modify it, and use it to install the modules:

- `yarn init`: From the command line, enter your project folder and run `yarn init`. You will be asked a series of questions to gather meta-information about your project, such as name, license, and author. Then, a `package.json` file will be auto generated with your answers.
- `dependencies`: Open `package.json` in your editor and modify the `JSON` object to add the key modules and `react-hot-loader` as regular dependencies:



The file path mentioned before a code block indicates the location of the code in the project directory. This convention has been maintained throughout the book to provide better context and guidance as you follow along with the code.

`mern-simplesetup/package.json`:

```
"dependencies": {  
  "@hot-loader/react-dom": "16.13.0",  
  "express": "4.17.1",  
  "mongodb": "3.5.5",  
  "react": "16.13.1",  
  "react-dom": "16.13.1",  
  "react-hot-loader": "4.12.20"  
}
```

- `devDependencies`: **Modify** `package.json` **further** to add the following Node modules required during development as `devDependencies`:

`mern-simplesetup/package.json`:

```
"devDependencies": {  
  "@babel/core": "7.9.0",  
  "@babel/preset-env": "7.9.0",  
  "@babel/preset-react": "7.9.4",  
  "babel-loader": "8.1.0",  
  "nodemon": "2.0.2",  
  "webpack": "4.42.1",  
  "webpack-cli": "3.3.11",  
  "webpack-dev-middleware": "3.7.2",  
  "webpack-hot-middleware": "2.25.0",  
  "webpack-node-externals": "1.7.2"  
}
```

- `yarn`: **Save** `package.json` and, from the command line, run the `yarn` command to fetch and add all these modules to your project.

With all the necessary modules installed and added to the project, next we will add configuration to compile and run the application code.

Configuring Babel, Webpack, and Nodemon

Before we start coding up the web application, we need to configure Babel, Webpack, and Nodemon to compile, bundle, and auto-reload the changes in the code during development.

Babel

Create a `.babelrc` file in your project folder and add the following JSON with `presets` and `plugins` specified:

mern-simplesetup/.babelrc:

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

In this configuration, we specify that we need Babel to transpile the latest JavaScript syntax with support for code in a Node.js environment and also React/JSX code. The `react-hot-loader/babel` plugin is required by the `react-hot-loader` module to compile React components.

Webpack

We will have to configure Webpack to bundle both the client and server code and the client code separately for production. Create `webpack.config.client.js`, `webpack.config.server.js`, and `webpack.config.client.production.js` files in your project folder. All three files will have the following code structure, starting with imports, then the definition of the `config` object, and finally the export of the defined `config` object:

```
const path = require('path')
const webpack = require('webpack')
const CURRENT_WORKING_DIR = process.cwd()

const config = { ... }

module.exports = config
```

The `config` JSON object will differ with values specific to the client- or server-side code, and development versus production code. In the following sections, we will highlight the relevant properties in each configuration instance.



Alternatively, you can also generate Webpack configurations using the interactive portal [Generate Custom Webpack Configuration](https://generatwebpackconfig.netlify.com/) at <https://generatwebpackconfig.netlify.com/> or using the Webpack-cli's `init` command from the command line.

Client-side Webpack configuration for development

Update the `config` object with the following in your `webpack.config.client.js` file in order to configure Webpack to bundle and hot-load React code during development:

mern-simplesetup/webpack.config.client.js:

```
const config = {
  name: "browser",
  mode: "development",
  devtool: 'eval-source-map',
  entry: [
    'webpack-hot-middleware/client?reload=true',
    path.join(CURRENT_WORKING_DIR, 'client/main.js')
  ],
  output: {
    path: path.join(CURRENT_WORKING_DIR, '/dist'),
    filename: 'bundle.js',
    publicPath: '/dist/'
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        use: ['babel-loader']
      }
    ]
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoEmitOnErrorsPlugin()
  ],
  resolve: {
    alias: {
      'react-dom': '@hot-loader/react-dom'
    }
  }
}
```

The highlighted keys and values in the `config` object will determine how Webpack bundles the code and where the bundled code will be placed:

- `mode` sets `process.env.NODE_ENV` to the given value and tells Webpack to use its built-in optimizations accordingly. If not set explicitly, it defaults to a value of "production". It can also be set via the command line by passing the value as a CLI argument.
- `devtool` specifies how source maps are generated, if at all. Generally, a source map provides a way of mapping code within a compressed file back to its original position in a source file to aid debugging.
- `entry` specifies the entry file where Webpack starts bundling, in this case with the `main.js` file in the `client` folder.
- `output` specifies the output path for the bundled code, in this case set to `dist/bundle.js`.
- `publicPath` allows specifying the base path for all assets in the application.
- `module` sets the regex rule for the file extension to be used for transpilation, and the folders to be excluded. The transpilation tool to be used here is `babel-loader`.
- `HotModuleReplacementPlugin` enables hot module replacement for `react-hot-loader`.
- `NoEmitOnErrorsPlugin` allows skipping emitting when there are compile errors.
- We also add a Webpack alias to point `react-dom` references to the `@hot-loader/react-dom` version.

The client-side code of the application will be loaded in the browser from the bundled code in `bundle.js`.

Webpack provides other configuration options too, which can be used as required depending on your code and bundling specifications, as we will see next when we explore server-side-specific bundling.

Server-side Webpack configuration

Modify the code to require `nodeExternals`, and update the `config` object with the following code in your `webpack.config.server.js` file to configure Webpack for bundling server-side code:

mern-simplesetup/webpack.config.server.js:

```
const nodeExternals = require('webpack-node-externals')
const config = {
  name: "server",
  entry: [ path.join(CURRENT_WORKING_DIR , ' ./server/server.js') ],
  target: "node",
  output: {
    path: path.join(CURRENT_WORKING_DIR , '/dist/'),
    filename: "server.generated.js",
    publicPath: '/dist/',
    libraryTarget: "commonjs2"
  },
  externals: [nodeExternals()],
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: [ 'babel-loader' ]
      }
    ]
  }
}
```

The `mode` option is not set here explicitly but will be passed as required when running the Webpack commands with respect to running for development or building for production.

Webpack starts bundling from the `server` folder with `server.js`, then outputs the bundled code in `server.generated.js` in the `dist` folder. During bundling, a CommonJS environment will be assumed as we are specifying `commonjs2` in `libraryTarget`, so the output will be assigned to `module.exports`.

We will run the server-side code using the generated bundle in `server.generated.js`.

Client-side Webpack configuration for production

To prepare the client-side code for production, update the `config` object with the following code in your `webpack.config.client.production.js` file:

mern-simplesetup/webpack.config.client.production.js:

```
const config = {
  mode: "production",
  entry: [
    path.join(CURRENT_WORKING_DIR, 'client/main.js')
  ],
  output: {
    path: path.join(CURRENT_WORKING_DIR, '/dist'),
    filename: 'bundle.js',
    publicPath: "/dist/"
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        use: [
          'babel-loader'
        ]
      }
    ]
  }
}
```

This will configure Webpack to bundle the React code to be used in production mode. The configuration here is similar to the client-side configuration for development mode, but without the hot-reloading plugin and debug configuration as these will not be required in production.

With the bundling configurations in place, we can add configuration for running these generated bundles automatically on code updates during development using Nodemon.

Nodemon

Create a `nodemon.json` file in your project folder and add the following configuration:

mern-simplesetup/nodemon.json:

```
{  
  "verbose": false,  
  "watch": [ "./server" ],  
  "exec": "webpack --mode=development --config  
          webpack.config.server.js  
          && node ./dist/server.generated.js"  
}
```

This configuration will set up `nodemon` to watch for changes in the server files during development, then execute compile and build commands as necessary. We can begin writing the code for a simple full-stack web application to see these configurations in action.

Frontend views with React

In order to start developing a frontend, first create a root template file called `template.js` in the project folder, which will render the HTML with React components:

mern-simplesetup/template.js:

```
export default () => {
  return `<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>MERN Kickstart</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/javascript" src="/dist/bundle.js">
  </script>
  </body>
</html>`}
```

When the server receives a request to the root URL, this HTML template will be rendered in the browser, and the `div` element with ID "root" will contain our React component.

Next, create a `client` folder where we will add two React files, `main.js`, and `HelloWorld.js`.

The `main.js` file simply renders the top-level entry React component in the `div` element in the HTML document:

mern-simplesetup/client/main.js:

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

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

In this case, the entry React component is the `HelloWorld` component imported from `HelloWorld.js`:

mern-simplesetup/client/HelloWorld.js:

```
import React from 'react'
import { hot } from 'react-hot-loader'

const HelloWorld = () => {
  return (
    <div>
      <h1>Hello World!</h1>
    </div>
  )
}

export default hot(module)(HelloWorld)
```

`HelloWorld.js` contains a basic `HelloWorld` React component, which is hot-exported to enable hot reloading with `react-hot-loader` during development.

To see the React component rendered in the browser when the server receives a request to the root URL, we need to use the Webpack and Babel setup to compile and bundle this code, and also add server-side code that responds to the root route request with the bundled code. We will implement this server-side code next.

Server with Express and Node

In the project folder, create a folder called `server`, and add a file called `server.js` that will set up the server. Then, add another file called `devBundle.js`, which will help compile the React code using Webpack configurations while in development mode. In the following sections, we will implement the Node-Express app, which initiates client-side code bundling, starts the server, sets up the path to serve static assets to the client, and renders the React view in a template when a `GET` request is made to the root route.

Express app

In `server.js`, we will first add code to import the `express` module in order to initialize an Express app:

mern-simplesetup/server/server.js:

```
| import express from 'express'  
| const app = express()
```

Then we will use this Express app to build out the rest of the Node server application.

Bundling React app during development

In order to keep the development flow simple, we will initialize Webpack to compile the client-side code when the server is run. In `devBundle.js`, we will set up a `compile` method that takes the Express app and configures it to use the Webpack middleware to compile, bundle, and serve code, as well as enable hot reloading in development mode:

mern-simplesetup/server/devBundle.js:

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

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

export default {
  compile
}
```

We will call this `compile` method in `server.js` by adding the following lines while in development mode:

mern-simplesetup/server/server.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 application code for production. In development mode, when these lines are executed, Webpack will compile and bundle the React code to place it in dist/bundle.js.

Serving static files from the dist folder

Webpack will compile client-side code in both development and production mode, then place the bundled files in the `dist` folder. To make these static files available on requests from the client side, we will add the following code in `server.js` to serve static files from the `dist` folder:

mern-simplesetup/server/server.js:

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

This will configure the Express app to return static files from the `dist` folder when the requested route starts with `/dist`.

Rendering templates at the root

When the server receives a request at the root URL `/`, we will render `template.js` in the browser. In `server.js`, add the following route-handling code to the Express app to receive `GET` requests at `/`:

mern-simplesetup/server/server.js:

```
import template from './.../template'
app.get('/', (req, res) => {
  res.status(200).send(template())
})
```

Finally, configure the Express app to start a server that listens on the specified port for incoming requests:

mern-simplesetup/server/server.js:

```
let port = process.env.PORT || 3000
app.listen(port, function onStart(err) {
  if (err) {
    console.log(err)
  }
  console.info('Server started on port %s.', port)
})
```

With this code, when the server is running, it will be able to accept requests at the root route and render the React view with the "Hello World" text in the browser. The only part missing from this full-stack implementation is a connection to the database, which we will add in the next section.

Connecting the server to MongoDB

To connect your Node server to MongoDB, add the following code to `server.js`, and make sure you have MongoDB running in your workspace or you have the URL of a cloud MongoDB database instance:

`mern-simplesetup/server/server.js`:

```
import { MongoClient } from 'mongodb'
const url = process.env.MONGODB_URI ||
  'mongodb://localhost:27017/mernSimpleSetup'
MongoClient.connect(url, (err, db)=>{
  console.log("Connected successfully to mongodb server")
  db.close()
})
```

In this code example, `MongoClient` is the driver that connects to the running MongoDB instance using its URL. It allows us to implement the database-related code in the backend. This completes our full-stack integration for this simple web application using the MERN setup, and finally, we can run this code to see this application working live.

Run scripts

In order to run the application, we will update the `package.json` file to add the following run scripts for development and production:

mern-simplesetup/package.json:

```
"scripts": {  
  "development": "nodemon",  
  "build": "webpack --config webpack.config.client.production.js  
    && webpack --mode=production --config  
      webpack.config.server.js",  
  "start": "NODE_ENV=production node ./dist/server.generated.js"  
}
```

Let's look at the code:

- `yarn development`: This command will get Nodemon, Webpack, and the server started for development.
- `yarn build`: This will generate the client and server code bundles for production mode (before running this script, make sure to remove the `devBundle.compile` code from `server.js`).
- `yarn start`: This command will run the bundled code in production.

You can use these commands to run the application either for debugging while you are developing the application or when the application is ready to go live in production.

Developing and debugging in real time

To run the code developed so far, and to ensure that everything is working, you can go through the following steps:

1. **Run the application from the command line:** `yarn development`.
2. **Load in browser:** Open the root URL in the browser, which is `http://localhost:3000` if you are using your local machine. You should see a page with the title MERN Kickstart that just shows Hello World!.
3. **Develop code and debug live:** Change the `HelloWorld.js` component text from `"Hello World!"` to just `"hello"`. Save the changes to see the instantaneous update in the browser, and also check the command-line output to see that `bundle.js` is not re-created. Similarly, you can also see instant updates when you change the server-side code, increasing productivity during development.

If you have made it this far, congratulations! You are all set to start developing exciting MERN applications.

Summary

In this chapter, we discussed development tool options and how to install MERN technologies, and then we wrote code to check whether the development environment is set up correctly.

We began by looking at the recommended workspace, IDE, version control software, and browser options suitable for web development. You can select from these options based on your preferences as a developer.

Next, we set up the MERN stack technologies by first installing MongoDB, Node, and Yarn, and then adding the remaining required libraries using Yarn.

Before moving on to writing code to check this setup, we configured Webpack and Babel to compile and bundle code during development, and to build production-ready code. We learned that it is necessary to compile the ES6 and JSX code that is used for developing a MERN application before opening the application on browsers.

Additionally, we made the development flow efficient by including React Hot Loader for frontend development, configuring Nodemon for backend development, and compiling both the client and server code in one command when the server is run during development.

In the next chapter, we will use this setup to start building a skeleton MERN application that will function as a base for full-featured applications.

Building MERN from the Ground Up

In this part, we build out a full-stack MERN application base from scratch and demonstrate how it can be easily extended to develop the first example application.

This section comprises the following chapters:

- [Chapter 3](#), *Building a Backend with MongoDB, Express, and Node*
- [Chapter 4](#), *Adding a React Frontend to Complete MERN*
- [Chapter 5](#), *Growing the Skeleton into a Social Media Application*

Building a Backend with MongoDB, Express, and Node

While developing different web applications, you will find there are common tasks, basic features, and implementation code repeated across the process. The same is true for the MERN applications that will be developed in this book. Taking these similarities into consideration, we will first lay the foundations for a skeleton MERN application that can be easily modified and extended to implement a variety of MERN applications.

In this chapter, we will cover the following topics and start with the backend implementation of the MERN skeleton using Node, Express, and MongoDB:

- Overview of the skeleton application
- Backend code setup
- User model with Mongoose
- User CRUD API endpoints with Express
- User Auth with JSON Web Tokens
- Running backend code and checking APIs

Overview of the skeleton application

The skeleton application will encapsulate rudimentary features and a workflow that's repeated for most MERN applications. We will build the skeleton as a basic but fully functioning MERN web application with user **create, read, update, delete (CRUD)**, and **authentication-authorization (auth)** capabilities; this will also demonstrate how to develop, organize, and run code for general web applications built using this stack. The aim is to keep the skeleton as simple as possible so that it is easy to extend and can be used as a base application for developing different MERN applications.

Feature breakdown

In the skeleton application, we will add the following use cases with user CRUD and auth functionality implementations:

- **Sign up:** Users can register by creating a new account using an email address.
- **User list:** Any visitor can see a list of all registered users.
- **Authentication:** Registered users can sign-in and sign-out.
- **Protected user profile:** Only registered users can view individual user details after signing in.
- **Authorized user edit and delete:** Only a registered and authenticated user can edit or remove their own user account details.

With these features, we will have a simple working web application that supports user accounts. We will start building this basic web application with the backend implementation, then integrate a React frontend to complete the full stack.

Defining the backend components

In this chapter, we will focus on building a working backend for the skeleton application with Node, Express, and MongoDB. The completed backend will be a standalone server-side application that can handle HTTP requests to create a user, list all users, and view, update, or delete a user in the database while taking user authentication and authorization into consideration.

User model

The user model will define the user details to be stored in the MongoDB database, and also handle user-related business logic such as password encryption and user data validation. The user model for this skeletal version will be basic with support for the following attributes:

Field name	Type	Description
name	String	Required field to store the user's name.
email	String	Required unique field to store the user's email and identify each account (only one account allowed per unique email).
password	String	A required field for authentication. The database will store the encrypted password and not the actual string for security purposes.
created	Date	Automatically generated timestamp when a new user account is created.
updated	Date	Automatically generated timestamp when existing user details are updated.

When we build applications by extending this skeleton, we can add more fields as required. But starting with these fields will be enough

to identify unique user accounts, and also for implementing user CRUD operation-related features.

API endpoints for user CRUD

To enable and handle user CRUD operations on the user database, the backend will implement and expose API endpoints that the frontend can utilize in the views, as follows:

Operation	API route	HTTP method
Create a user	/api/users	POST
List all users	/api/users	GET
Fetch a user	/api/users/:userId	GET
Update a user	/api/users/:userId	PUT
Delete a user	/api/users/:userId	DELETE
User sign-in	/auth/signin	POST
User signout (optional)	/auth/signout	GET

Some of these user CRUD operations will have protected access, which will require the requesting client to be authenticated, authorized, or both, as defined by the feature specifications. The last two routes in the table are for authentication and will allow the user to sign-in and sign-out. For the applications developed in this book, we will use the JWT mechanism to implement these authentication features, as discussed in more detail in the next section.

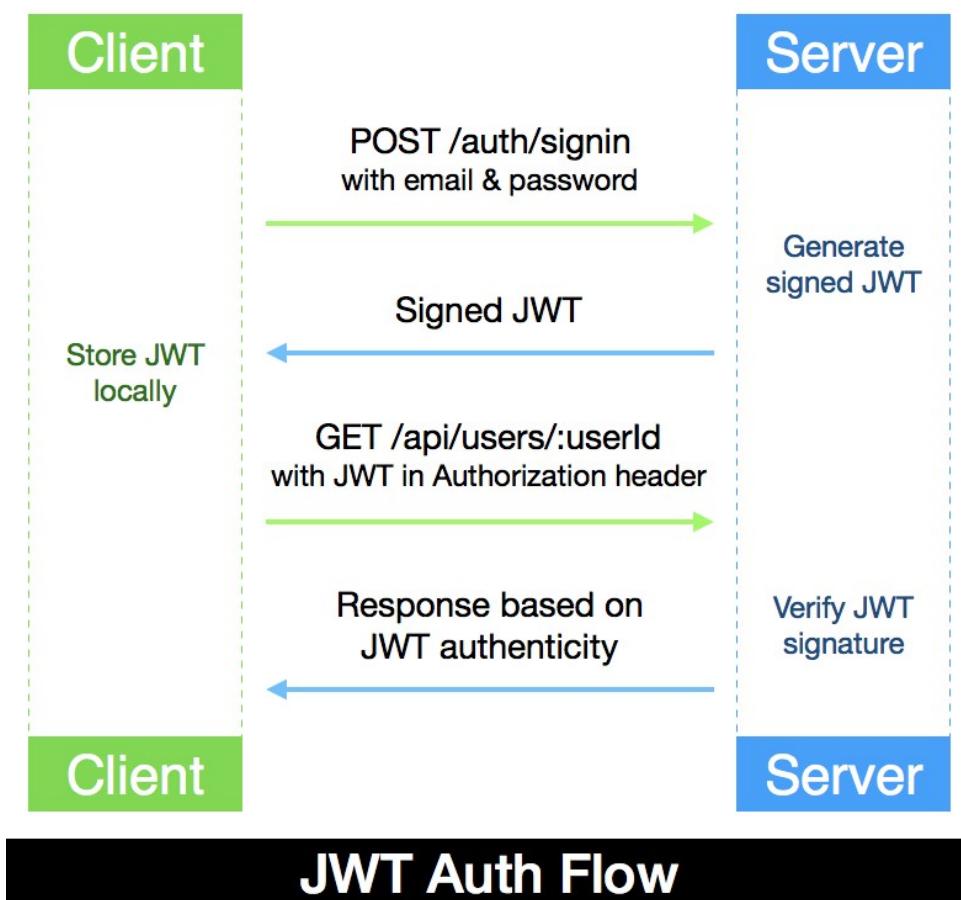
Auth with JSON Web Tokens

To restrict and protect access to user API endpoints according to the skeleton features, the backend will need to incorporate authentication and authorization mechanisms. There are a number of options when it comes to implementing user auth for web applications. The most common and time-tested option is the use of sessions to store user state on both the client and server-side. But a newer approach is the use of **JSON Web Token (JWT)** as a stateless authentication mechanism that does not require storing user state on the server side.

Both approaches have strengths for relevant real-world use cases. However, for the purpose of keeping the code simple in this book, and because it pairs well with the MERN stack and our example applications, we will use JWT for auth implementation.

How JWT works

Before diving into the implementation of authentication with JWT in the MERN stack, we will look at how this mechanism generally works across a client-server application, as outlined in the following diagram:



Initially, when a user signs in using their credentials, the server-side generates a JWT signed with a secret key and a unique user detail. Then, this token is returned to the requesting client to be saved locally either in `localStorage`, `sessionStorage` or a cookie in the browser, essentially handing over the responsibility for maintaining user state to the client-side.

For HTTP requests that are made following a successful sign-in, especially requests for API endpoints that are protected and have restricted access, the client-side has to attach this token to the request. More specifically, the `JSON Web Token` must be included in the `Authorization` header as a `Bearer`:

```
| Authorization: Bearer <JSON Web Token>
```

When the server receives a request for a protected API endpoint, it checks the `Authorization` header of the request for a valid JWT, then verifies the signature to identify the sender and ensures the request data was not corrupted. If the token is valid, the requesting client is given access to the associated operation or resource; otherwise, an authorization error is returned.

In the skeleton application, when a user signs in with their email and password, the backend will generate a signed JWT with the user's ID and with a secret key that's available only on the server. This token will then be required for verification when a user tries to view any user profiles, update their account details, or delete their user account.

Implementing the user model to store and validate user data, then integrating it with APIs to perform CRUD operations based on auth with JWT, will produce a functioning standalone backend. In the rest of this chapter, we will look at how to achieve this in the MERN stack and setup.

Setting up the skeleton backend

To start developing the backend part of the MERN skeleton, we will set up the project folder, install and configure the necessary Node modules, and then prepare run scripts to aid development and run the code. Then, we will go through the code step by step to implement a working Express server, a user model with Mongoose, API endpoints with Express router, and JWT-based auth to meet the specifications we defined earlier for user-oriented features.



The code that will be discussed in this chapter, as well as the complete skeleton application, is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter03%20and%2004/mern-skeleton>. The code for just the backend is available at the same repository in the branch named `mern2-skeleton-backend`. You can clone this code and run the application as you go through the code explanations in the rest of this chapter.

Folder and file structure

As we go through our setup and implementation in the rest of this chapter, we will end up with the following folder structure containing files that are relevant to the MERN skeleton backend. With these files, we will have a functioning, standalone server-side application:

```
| mern_skeleton/
|   -- config/
|     --- config.js
|   -- server/
|     --- controllers/
|       ---- auth.controller.js
|       ---- user.controller.js
|     --- helpers/
|       ---- dbErrorHandler.js
|     --- models/
|       ---- user.model.js
|     --- routes/
|       ---- auth.routes.js
|       ---- user.routes.js
|     --- express.js
|     --- server.js
|   -- .babelrc
|   -- nodemon.json
|   -- package.json
|   -- template.js
|   -- webpack.config.server.js
|   -- yarn.lock
```

We will keep the configuration files in the root directory and the backend-related code in the `server` folder. Within the `server` folder, we will divide the backend code into modules containing models, controllers, routes, helpers, and common server-side code. This folder structure will be further expanded in the next chapter, where we'll complete the skeleton application by adding a React frontend.

Initializing the project

If your development environment is already set up, you can initialize the MERN project to start developing the backend. First, we will initialize `package.json` in the project folder, configure and install any development dependencies, set configuration variables to be used in the code, and update `package.json` with run scripts to help develop and run the code.

Adding package.json

We will need a `package.json` file to store meta information about the project, list module dependencies with version numbers, and to define run scripts. To initialize a `package.json` file in the project folder, go to the project folder from the command line and run `yarn init`, then follow the instructions to add the necessary details. With `package.json` created, we can proceed with setup and development and update the file as more modules are required throughout code implementation.

Development dependencies

In order to begin the development process and run the backend server code, we will configure and install Babel, Webpack, and Nodemon, as discussed in [Chapter 2](#), *Preparing the Development Environment*, and make some minor adjustments to the backend.

Babel

Since we will be using ES6+ and the latest JS features in the backend code, we will install and configure Babel modules to convert ES6+ into older versions of JS so that it's compatible with the Node version being used.

First, we'll configure Babel in the `.babelrc` file with presets for the latest JS features and specify the current version of Node as the target environment.

```
mern-skeleton/.babelrc:
```

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

Setting `targets.node` to `current` instructs Babel to compile against the current version of Node and allows us to use expressions such as `async/await` in our backend code.

Next, we need to install the Babel modules as `devDependencies` from the command line:

```
| yarn add --dev @babel/core babel-loader @babel/preset-env
```

Once the module installations are done, you will notice that the `devDependencies` list has been updated in the `package.json` file.

Webpack

We will need Webpack to compile and bundle the server-side code using Babel. For configuration, we can use the same `webpack.config.server.js` we discussed in [Chapter 2, *Preparing the Development Environment*](#).

From the command line, run the following command to install `webpack`, `webpack-cli`, and the `webpack-node-externals` module:

```
| yarn add --dev webpack webpack-cli webpack-node-externals
```

This will install the Webpack modules and update the `package.json` file.

Nodemon

To automatically restart the Node server as we update our code during development, we will use Nodemon to monitor the server code for changes. We can use the same installation and configuration guidelines we discussed in [Chapter 2, *Preparing the Development Environment*](#).

Before we add run scripts to start developing and running the backend code, we will define configuration variables for values that are used across the backend implementation.

Config variables

In the `config/config.js` file, we will define some server-side configuration-related variables that will be used in the code but should not be hardcoded as a best practice, as well as for security purposes.

mern-skeleton/config/config.js:

```
const config = {
  env: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 3000,
  jwtSecret: process.env.JWT_SECRET || "YOUR_secret_key",
  mongoUri: process.env.MONGODB_URI ||
    process.env.MONGO_HOST ||
    'mongodb://' + (process.env.IP || 'localhost') + ':' +
    (process.env.MONGO_PORT || '27017') +
    '/mernproject'
}
export default config
```

The config variables that were defined are as follows:

- `env`: To differentiate between development and production modes
- `port`: To define the listening port for the server
- `jwtSecret`: The secret key to be used to sign JWT
- `mongoUri`: The location of the MongoDB database instance for the project

These variables will give us the flexibility to change values from a single file and use it across the backend code. Next, we will add the run scripts, which will allow us to run and debug the backend implementation.

Running scripts

To run the server as we develop the code for only the backend, we can start with the `yarn development` script in the `package.json` file. For the complete skeleton application, we will use the same run scripts we defined in [Chapter 2, *Preparing the Development Environment*](#).

mern-skeleton/package.json:

```
| "scripts": {  
|   "development": "nodemon"  
| }
```

With this script added, running `yarn development` in the command line from your project folder will basically start Nodemon according to the configuration in `nodemon.json`. The configuration instructs Nodemon to monitor server files for updates and, on update, to build the files again, then restart the server so that the changes are immediately available. We will begin by implementing a working server with this configuration in place.

Preparing the server

In this section, we will integrate Express, Node, and MongoDB in order to run a completely configured server before we start implementing user-specific features.

Configuring Express

To use Express, we will install it and then add and configure it in the `server/express.js` file.

From the command line, run the following command to install the `express` module and to have the `package.json` file automatically updated:

```
| yarn add express
```

Once Express has been installed, we can import it into the `express.js` file, configure it as required, and make it available to the rest of the app.

mern-skeleton/server/express.js:

```
import express from 'express'  
const app = express()  
/*... configure express ... */  
export default app
```

To handle HTTP requests and serve responses properly, we will use the following modules to configure Express:

- `body-parser`: Request body-parsing middleware to handle the complexities of parsing streamable request objects so that we can simplify browser-server communication by exchanging JSON in the request body. To install the module, run `yarn add body-parser` from the command line. Then, configure the Express app with `bodyParser.json()` and `bodyParser.urlencoded({ extended: true })`.
- `cookie-parser`: Cookie parsing middleware to parse and set cookies in request objects. To install the `cookie-parser` module, run `yarn add cookie-parser` from the command line.
- `compression`: Compression middleware that will attempt to compress response bodies for all requests that traverse through the middleware. To install the `compression` module, run `yarn add compression` from the command line.

- `helmet`: Collection of middleware functions to help secure Express apps by setting various HTTP headers. To install the `helmet` module, run `yarn add helmet` from the command line.
- `cors`: Middleware to enable **cross-origin resource sharing (CORS)**. To install the `cors` module, run `yarn add cors` from the command line.

After the preceding modules have been installed, we can update `express.js` to import these modules and configure the Express app before exporting it for use in the rest of the server code.

The updated `mern-skeleton/server/express.js` code should be as follows:

```
import express from 'express'
import bodyParser from 'body-parser'
import cookieParser from 'cookie-parser'
import compress from 'compression'
import cors from 'cors'
import helmet from 'helmet'

const app = express()

app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }))
app.use(cookieParser())
app.use(compress())
app.use(helmet())
app.use(cors())

export default app
```

The Express app can now accept and process information from incoming HTTP requests, for which we first need to start a server using this app.

Starting the server

With the Express app configured to accept HTTP requests, we can go ahead and use it to implement a server that can listen for incoming requests.

In the `mern-skeleton/server/server.js` file, add the following code to implement the server:

```
import config from '../../config/config'
import app from './express'

app.listen(config.port, (err) => {
  if (err) {
    console.log(err)
  }
  console.info('Server started on port %s.', config.port)
})
```

First, we import the config variables to set the port number that the server will listen on and then import the configured Express app to start the server. To get this code running and continue development, we can run `yarn development` from the command line. If the code has no errors, the server should start running with Nodemon monitoring for code changes. Next, we will update this server code to integrate the database connection.

Setting up Mongoose and connecting to MongoDB

We will be using the `mongoose` module to implement the user model in this skeleton, as well as all future data models for our MERN applications. Here, we will start by configuring Mongoose and utilizing it to define a connection with the MongoDB database.

First, to install the `mongoose` module, run the following command:

```
| yarn add mongoose
```

Then, update the `server.js` file to import the `mongoose` module, configure it so that it uses native ES6 promises, and finally use it to handle the connection to the MongoDB database for the project.

mern-skeleton/server/server.js:

```
import mongoose from 'mongoose'

mongoose.Promise = global.Promise
mongoose.connect(config.mongoUri, { useNewUrlParser: true,
                                     useCreateIndex: true,
                                     useUnifiedTopology: true })

mongoose.connection.on('error', () => {
  throw new Error(`unable to connect to database: ${mongoUri}`)
})
```

If you have the code running in development and also have MongoDB running, saving this update should successfully restart the server, which is now integrated with Mongoose and MongoDB.

 *Mongoose is a MongoDB object modeling tool that provides a schema-based solution to model application data. It includes built-in type casting, validation, query building, and business logic hooks. Using Mongoose with this backend stack provides a higher layer over MongoDB with more functionality, including mapping object models to database documents. This makes it simpler and more productive to develop with a Node and MongoDB backend. To learn more about Mongoose, visit mongoosejs.com.*

With an Express app configured, the database integrated with Mongoose, and a listening server ready, we can add code to load an HTML view from this backend.

Serving an HTML template at a root URL

With a Node- Express- and MongoDB- enabled server now running, we can extend it so that it serves an HTML template in response to an incoming request at the root URL `/`.

In the `template.js` file, add a JS function that returns a simple HTML document that will render `Hello World` on the browser screen.

mern-skeleton/template.js:

```
export default () => {
  return `<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>MERN Skeleton</title>
  </head>
  <body>
    <div id="root">Hello World</div>
  </body>
</html>`
}
```

To serve this template at the root URL, update the `express.js` file to import this template and send it in the response to a GET request for the `'/'` route.

mern-skeleton/server/express.js:

```
import Template from './.../template'
...
app.get('/', (req, res) => {
  res.status(200).send(Template())
})
...
```

With this update, opening the root URL in a browser should show Hello World rendered on the page. If you are running the code on

your local machine, the root URL will be `http://localhost:3000/`.

At this point, the backend Node- Express- and MongoDB-based server that we can build on to add user-specific features.

Implementing the user model

We will implement the user model in the `server/models/user.model.js` file and use Mongoose to define the schema with the necessary user data fields. We're doing this so that we can add built-in validation for the fields and incorporate business logic such as password encryption, authentication, and custom validation.

We will begin by importing the `mongoose` module and use it to generate a `UserSchema`, which will contain the schema definition and user-related business logic to make up the user model. This user model will be exported so that it can be used by the rest of the backend code.

`mern-skeleton/server/models/user.model.js`:

```
import mongoose from 'mongoose'

const UserSchema = new mongoose.Schema({ ... })

export default mongoose.model('User', UserSchema)
```

The `mongoose.Schema()` function takes a schema definition object as a parameter to generate a new Mongoose schema object that will specify the properties or structure of each document in a collection. We will discuss this schema definition for the User collection before we add any business logic code to complete the user model.

User schema definition

The user schema definition object that's needed to generate the new Mongoose schema will declare all user data fields and associated properties. The schema will record user-related information including name, email, created-at and last-updated-at timestamps, hashed passwords, and the associated unique password salt. We will elaborate on these properties next, showing you how each field is defined in the user schema code.

Name

The `name` field is a required field of the `String` type.

mern-skeleton/server/models/user.model.js:

```
name: {  
  type: String,  
  trim: true,  
  required: 'Name is required'  
},
```

This field will store the user's name.

Email

The `email` field is a required field of the `string` type.

mern-skeleton/server/models/user.model.js:

```
email: {  
  type: String,  
  trim: true,  
  unique: 'Email already exists',  
  match: [/.+\@\.+\.+/], 'Please fill a valid email address'],  
  required: 'Email is required'  
},
```

The value to be stored in this email field must have a valid email format and must also be unique in the user collection.

Created and updated timestamps

The `created` and `updated` fields are `Date` values.

mern-skeleton/server/models/user.model.js:

```
  created: {  
    type: Date,  
    default: Date.now  
  },  
  updated: Date,
```

These `Date` values will be programmatically generated to record timestamps that indicate when a user is created and user data is updated.

Hashed password and salt

The `hashed_password` and `salt` fields represent the encrypted user password that we will use for authentication.

mern-skeleton/server/models/user.model.js:

```
  hashed_password: {
    type: String,
    required: "Password is required"
  },
  salt: String
```

The actual password string is not stored directly in the database for security purposes and is handled separately, as discussed in the next section.

Password for auth

The `password` field is very crucial for providing secure user authentication in any application, and each user password needs to be encrypted, validated, and authenticated securely as a part of the user model.

Handling the password string as a virtual field

The `password` string that's provided by the user is not stored directly in the user document. Instead, it is handled as a `virtual` field.

mern-skeleton/server/models/user.model.js:

```
UserSchema
  .virtual('password')
  .set(function(password) {
    this._password = password
    this.salt = this.makeSalt()
    this.hashed_password = this.encryptPassword(password)
  })
  .get(function() {
    return this._password
  })
```

When the `password` value is received on user creation or update, it is encrypted into a new hashed value and set to the `hashed_password` field, along with the unique `salt` value in the `salt` field.

Encryption and authentication

The encryption logic and salt generation logic, which are used to generate the `hashed_password` and `salt` values representing the `password` value, are defined as `UserSchema` methods.

mern-skeleton/server/models/user.model.js:

```
UserSchema.methods = {
  authenticate: function(plainText) {
    return this.encryptPassword(plainText) === this.hashed_password
  },
  encryptPassword: function(password) {
    if (!password) return ''
    try {
      return crypto
        .createHmac('sha1', this.salt)
        .update(password)
        .digest('hex')
    } catch (err) {
      return ''
    }
  },
  makeSalt: function() {
    return Math.round((new Date().valueOf() * Math.random())) + ''
  }
}
```

The `UserSchema` methods can be used to provide the following functionality:

- `authenticate`: This method is called to verify sign-in attempts by matching the user-provided password text with the `hashed_password` stored in the database for a specific user.
- `encryptPassword`: This method is used to generate an encrypted hash from the plain-text password and a unique `salt` value using the `crypto` module from Node.
- `makeSalt`: This method generates a unique and random salt value using the current timestamp at execution and `Math.random()`.

The `crypto` module provides a range of cryptographic functionality, including some standard cryptographic hashing algorithms. In our code, we use the SHA1 hashing algorithm and `createHmac` from `crypto` to generate the cryptographic HMAC hash from the password text



and salt pair.

Hashing algorithms generate the same hash for the same input value. But to ensure two users don't end up with the same hashed password if they happen to use the same password text, we pair each password with a unique salt value before generating the hashed password for each user. This will also make it difficult to guess the hashing algorithm being used because the same user input is seemingly generating different hashes.

These `UserSchema` methods are used to encrypt the user-provided password string into a `hashed_password` with a randomly generated `salt` value. The `hashed_password` and the `salt` are stored in the user document when the user details are saved to the database on a create or update. Both the `hashed_password` and `salt` values are required in order to match and authenticate a password string provided during user sign-in using the `authenticate` method. We should also ensure the user selects a strong password string to begin with, which can done by adding custom validation to the passport field.

Password field validation

To add validation constraints to the actual password string that's selected by the end user, we need to add custom validation logic and associate it with the `hashed_password` field in the schema.

mern-skeleton/server/models/user.model.js:

```
UserSchema.path('hashed_password').validate(function(v) {  
  if (this._password && this._password.length < 6) {  
    this.invalidate('password', 'Password must be at least 6  
characters.')  
  }  
  if (this.isNew && !this._password) {  
    this.invalidate('password', 'Password is required')  
  }  
, null)
```

We will keep the password validation criteria simple in our application and ensure that a password value is provided and it has a length of at least six characters when a new user is created or an existing password is updated. We achieve this by adding custom validation to check the password value before Mongoose attempts to store the `hashed_password` value. If validation fails, the logic will return the relevant error message.

The defined `UserSchema`, along with all the password-related business logic, completes the user model implementation. Now, we can import and use this user model in other parts of the backend code. But before we begin using this model to extend backend functionality, we will add a helper module so that we can parse readable Mongoose error messages, which are thrown against schema validations.

Mongoose error handling

The validation constraints that are added to the user schema fields will throw error messages if they're violated when user data is saved to the database. To handle these validation errors and other errors that the database may throw when we make queries to it, we will define a helper method that will return a relevant error message that can be propagated in the request-response cycle as appropriate.

We will add the `getErrorMessage` helper method to the `server/helpers/dbErrorHandler.js` file. This method will parse and return the error message associated with the specific validation error or other errors that can occur while querying MongoDB using Mongoose.

mern-skeleton/server/helpers/dbErrorHandler.js:

```
const getErrorMessage = (err) => {
  let message = ''
  if (err.code) {
    switch (err.code) {
      case 11000:
      case 11001:
        message = getUniqueErrorMessage(err)
        break
      default:
        message = 'Something went wrong'
    }
  } else {
    for (let errName in err.errors) {
      if (err.errors[errName].message)
        message = err.errors[errName].message
    }
  }
  return message
}

export default {getErrorMessage}
```

Errors that are not thrown because of a Mongoose validator violation will contain an associated error `code`. In some cases, these errors need to be handled differently. For example, errors caused due to a violation of the `unique` constraint will return an error object that is different from Mongoose validation errors. The `unique` option is not a validator but a convenient helper for building MongoDB `unique` indexes, so we will add another `getUniqueErrorMessage` method to parse the `unique` constraint-related error object and construct an appropriate error message.

mern-skeleton/server/helpers/dbErrorHandler.js:

```
const getUniqueErrorMessage = (err) => {
  let output
  try {
    let fieldName =
      err.message.substring(err.message.lastIndexOf('$') + 2,
      err.message.lastIndexOf('_1'))
    output = fieldName.charAt(0).toUpperCase() + fieldName.slice(1) +
      ' already exists'
  } catch (ex) {
    output = 'Unique field already exists'
  }
  return output
}
```

By using the `getErrorMessage` function that's exported from this helper file, we can add meaningful error messages when handling errors that are thrown by Mongoose operations.

With the user model completed, we can perform Mongoose operations that are relevant to achieving user CRUD functionality with the User APIs we'll develop in the next section.

Adding user CRUD APIs

The user API endpoints exposed by the Express app will allow the frontend to perform CRUD operations on documents that are generated according to the user model. To implement these working endpoints, we will write Express routes and the corresponding controller callback functions that should be executed when HTTP requests come in for these declared routes. In this section, we will look at how these endpoints work without any auth restrictions.

Our user API routes will be declared using the Express router in `server/routes/user.routes.js`, and then mounted on the Express app we configured in `server/express.js`.

mern-skeleton/server/express.js:

```
| import userRoutes from './routes/user.routes'  
| ...  
| app.use('/', userRoutes)  
| ...
```

All routes and API endpoints, such as the user-specific routes we'll declare next, need to be mounted on the Express app so that they can be accessed from the client-side.

User routes

The user routes that are defined in the `user.routes.js` file will use `express.Router()` to define route paths with the relevant HTTP methods and assign the corresponding controller function that should be called when these requests are received by the server.

We will keep the user routes simplistic by using the following:

- `/api/users` for the following:
 - Listing users with GET
 - Creating a new user with POST
- `/api/users/:userId` for the following:
 - Fetching a user with GET
 - Updating a user with PUT
 - Deleting a user with DELETE

The resulting `user.routes.js` code will look as follows (without the auth considerations that need to be added for protected routes).

mern-skeleton/server/routes/user.routes.js:

```
import express from 'express'
import userCtrl from '../controllers/user.controller'

const router = express.Router()

router.route('/api/users')
  .get(userCtrl.list)
  .post(userCtrl.create)

router.route('/api/users/:userId')
  .get(userCtrl.read)
  .put(userCtrl.update)
  .delete(userCtrl.remove)

router.param('userId', userCtrl.userByID)

export default router
```

Besides declaring API endpoints that correspond to user CRUD operations, we'll also configure the Express router so that it handles the `userId` parameter in a requested route by executing the `userByID` controller function.

When the server receives requests at each of these defined routes, the corresponding controller functions are invoked. We will define the functionality for each of these controller methods and export it from the `user.controller.js` file in the next subsection.

User controller

The `server/controllers/user.controller.js` file will contain definitions of the controller methods that were used in the preceding user route declarations as callbacks to be executed when a route request is received by the server.

The `user.controller.js` file will have the following structure:

```
import User from '../models/user.model'
import extend from 'lodash/extend'
import errorHandler from './error.controller'

const create = (req, res, next) => { ... }
const list = (req, res) => { ... }
const userByID = (req, res, next, id) => { ... }
const read = (req, res) => { ... }
const update = (req, res, next) => { ... }
const remove = (req, res, next) => { ... }

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

This controller will make use of the `errorHandler` helper to respond to route requests with meaningful messages when a Mongoose error occurs. It will also use a module called `lodash` when updating an existing user with changed values.



`lodash` is a JavaScript library that provides utility functions for common programming tasks, including the manipulation of arrays and objects. To install `lodash`, run `yarn add lodash` from the command line.

Each of the controller functions we defined previously is related to a route request, and will be elaborated on in relation to each API use case.

Creating a new user

The API endpoint to create a new user is declared in the following route.

```
mern-skeleton/server/routes/user.routes.js:
```

```
| router.route('/api/users').post(userCtrl.create)
```

When the Express app gets a POST request at `'/api/users'`, it calls the `create` function we defined in the controller.

```
mern-skeleton/server/controllers/user.controller.js:
```

```
const create = async (req, res) => {
  const user = new User(req.body)
  try {
    await user.save()
    return res.status(200).json({
      message: "Successfully signed up!"
    })
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This function creates a new user with the user JSON object that's received in the POST request from the frontend within `req.body`. The call to `user.save` attempts to save the new user in the database after Mongoose has performed a validation check on the data. Consequently, an error or success response is returned to the requesting client.

The `create` function is defined as an asynchronous function with the **async** keyword, allowing us to use **await** with `user.save()`, which returns a Promise. Using the **await** keyword inside an **async** function causes this function to wait until the returned

Promise resolves, before the next lines of code are executed. If the Promise rejects, an error is thrown and caught in the `catch` block.



Async/await is an addition to ES8 that allows us to write asynchronous JavaScript code in a seemingly sequential or synchronous manner. For controller functions that handle asynchronous behavior such as accessing the database, we will use the `async/await` syntax to implement them.

Similarly, in the next section, we will use `async/await` while implementing the controller function to list all users after querying the database.

Listing all users

The API endpoint to fetch all the users is declared in the following route.

```
mern-skeleton/server/routes/user.routes.js:
```

```
| router.route('/api/users').get(userCtrl.list)
```

When the Express app gets a GET request at `'/api/users'`, it executes the `list` controller function.

```
mern-skeleton/server/controllers/user.controller.js:
```

```
const list = async (req, res) => {
  try {
    let users = await User.find().select('name email updated created')
    res.json(users)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The `list` controller function finds all the users from the database, populates only the `name`, `email`, `created`, and `updated` fields in the resulting user list, and then returns this list of users as JSON objects in an array to the requesting client.

The remaining CRUD operations to read, update, and delete a single user require that we retrieve a specific user by ID first. In the next section, we will implement the controller functions that enable fetching a single user from the database to either return the user, update the user, or delete the user in response to the corresponding requests.

Loading a user by ID to read, update, or delete

All three API endpoints for read, update, and delete require a user to be loaded from the database based on the user ID of the user being accessed. We will program the Express router to do this action first before responding to a specific request to read, update, or delete.

Loading

Whenever the Express app receives a request to a route that matches a path containing the `:userId` parameter in it, the app will execute the `userByID` controller function, which fetches and loads the user into the Express request object, before propagating it to the `next` function that's specific to the request that came in.

mern-skeleton/server/routes/user.routes.js:

```
| router.param('userId', userCtrl.userByID)
```

The `userByID` controller function uses the value in the `:userId` parameter to query the database by `_id` and load the matching user's details.

mern-skeleton/server/controllers/user.controller.js:

```
const userByID = async (req, res, next, id) => {
  try {
    let user = await User.findById(id)
    if (!user)
      return res.status('400').json({
        error: "User not found"
      })
    req.profile = user
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve user"
    })
  }
}
```

If a matching user is found in the database, the user object is appended to the request object in the `profile` key. Then, the `next()` middleware is used to propagate control to the next relevant controller function. For example, if the original request was to read a user profile, the `next()` call in `userByID` would go to the `read` controller function, which is discussed next.

Reading

The API endpoint to read a single user's data is declared in the following route.

```
mern-skeleton/server/routes/user.routes.js:
```

```
| router.route('/api/users/:userId').get(userCtrl.read)
```

When the Express app gets a GET request at `'/api/users/:userId'`, it executes the `userByID` controller function to load the user by the `userId` value, followed by the `read` controller function.

```
mern-skeleton/server/controllers/user.controller.js:
```

```
| const read = (req, res) => {
|   req.profile.hashed_password = undefined
|   req.profile.salt = undefined
|   return res.json(req.profile)
| }
```

The `read` function retrieves the user details from `req.profile` and removes sensitive information, such as the `hashed_password` and `salt` values, before sending the user object in the response to the requesting client. This rule is also followed in implementing the controller function to update a user, as shown next.

Updating

The API endpoint to update a single user is declared in the following route.

```
mern-skeleton/server/routes/user.routes.js:
```

```
| router.route('/api/users/:userId').put(userCtrl.update)
```

When the Express app gets a PUT request at `'/api/users/:userId'`, similar to `read`, it loads the user with the `:userId` parameter value before executing the `update` controller function.

```
mern-skeleton/server/controllers/user.controller.js:
```

```
const update = async (req, res) => {
  try {
    let user = req.profile
    user = extend(user, req.body)
    user.updated = Date.now()
    await user.save()
    user.hashed_password = undefined
    user.salt = undefined
    res.json(user)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The `update` function retrieves the user details from `req.profile` and then uses the `lodash` module to extend and merge the changes that came in the request body to update the user data. Before saving this updated user to the database, the `updated` field is populated with the current date to reflect the last updated timestamp. Upon successfully saving this update, the updated user object is cleaned by removing sensitive data, such as `hashed_password` and `salt`, before sending the user object in the response to the requesting client. Implementation of the

final user controller function to delete a user is similar to the `update` function, as detailed in the next section.

Deleting

The API endpoint to delete a user is declared in the following route.

mern-skeleton/server/routes/user.routes.js:

```
| router.route('/api/users/:userId').delete(userCtrl.remove)
```

When the Express app gets a DELETE request at '/api/users/:userId', similar to read and update, it loads the user by ID and then the `remove` controller function is executed.

mern-skeleton/server/controllers/user.controller.js:

```
const remove = async (req, res) => {
  try {
    let user = req.profile
    let deletedUser = await user.remove()
    deletedUser.hashed_password = undefined
    deletedUser.salt = undefined
    res.json(deletedUser)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The `remove` function retrieves the user from `req.profile` and uses the `remove()` query to delete the user from the database. On successful deletion, the requesting client is returned the deleted user object in the response.

With the implementation of the API endpoints so far, any client can perform CRUD operations on the user model. However, we want to restrict access to some of these operations with authentication and authorization. We'll look at this in the next section.

Integrating user auth and protected routes

To restrict access to user operations such as user profile view, user update, and user delete, we will first implement sign-in authentication with JWT, then use it to protect and authorize the read, update, and delete routes.

The auth-related API endpoints for sign-in and sign-out will be declared in `server/routes/auth.routes.js` and then mounted on the Express app in `server/express.js`.

mern-skeleton/server/express.js:

```
| import authRoutes from './routes/auth.routes'  
| ...  
| app.use('/', authRoutes)  
| ...
```

This will make the routes we define in `auth.routes.js` accessible from the client-side.

Auth routes

The two auth APIs are defined in the `auth.routes.js` file using `express.Router()` to declare the route paths with the relevant HTTP methods. They're also assigned the corresponding controller functions, which should be called when requests are received for these routes.

The auth routes are as follows:

- `'/auth/signin'`: POST request to authenticate the user with their email and password
- `'/auth/signout'`: GET request to clear the cookie containing a JWT that was set on the response object after sign-in

The resulting `mern-skeleton/server/routes/auth.routes.js` file will be as follows:

```
import express from 'express'
import authCtrl from '../controllers/auth.controller'

const router = express.Router()

router.route('/auth/signin')
  .post(authCtrl.signin)
router.route('/auth/signout')
  .get(authCtrl.signout)

export default router
```

A POST request to the `signin` route and a GET request to the `signout` route will invoke the corresponding controller functions defined in the `auth.controller.js` file, as discussed in the next section.

Auth controller

The auth controller functions in `server/controllers/auth.controller.js` will not only handle requests to the `signin` and `signout` routes, but also provide JWT and `express-jwt` functionality to enable authentication and authorization for protected user API endpoints.

The `mern-skeleton/server/controllers/auth.controller.js` file will have the following structure:

```
import User from '../models/user.model'
import jwt from 'jsonwebtoken'
import expressJwt from 'express-jwt'
import config from '../../config/config'

const signin = (req, res) => { ... }
const signout = (req, res) => { ... }
const requireSignin = ...
const hasAuthorization = (req, res) => { ... }

export default { signin, signout, requireSignin, hasAuthorization }
```

The four controller functions are elaborated on in the following sections to show how the backend implements user auth using JSON Web Tokens. We'll start with the `signin` controller function in the next section.

Sign-in

The API endpoint to sign-in a user is declared in the following route.

mern-skeleton/server/routes/auth.routes.js:

```
| router.route('/auth/signin').post(authCtrl.signin)
```

When the Express app gets a POST request at '/auth/signin', it executes the `signin` controller function.

mern-skeleton/server/controllers/auth.controller.js:

```
const signin = async (req, res) => {
  try {
    let user = await User.findOne({ "email": req.body.email })
    if (!user)
      return res.status('401').json({ error: "User not found" })

    if (!user.authenticate(req.body.password)) {
      return res.status('401').send({ error: "Email and
        password don't match." })
    }

    const token = jwt.sign({ _id: user._id }, config.jwtSecret)

    res.cookie('t', token, { expire: new Date() + 9999 })

    return res.json({
      token,
      user: {
        _id: user._id,
        name: user.name,
        email: user.email
      }
    })
  } catch (err) {
    return res.status('401').json({ error: "Could not sign in" })
  }
}
```

The `POST` request object receives the email and password in `req.body`. This email is used to retrieve a matching user from the database. Then, the password authentication method defined in `UserSchema` is used to verify the password that's received in `req.body` from the client.

If the password is successfully verified, the JWT module is used to generate a signed JWT using a secret key and the user's `_id` value.



Install the `jsonwebtoken` module to make it available to this controller in the import by running `yarn add jsonwebtoken` from the command line.

Then, the signed JWT is returned to the authenticated client, along with the user's details. Optionally, we can also set the token to a cookie in the response object so that it is available to the client-side if cookies are the chosen form of JWT storage. On the client-side, this token must be attached as an `Authorization` header when requesting protected routes from the server. To sign-out a user, the client-side can simply delete this token depending on how it is being stored. In the next section, we will learn how to use a `signout` API endpoint to clear the cookie containing the token.

Signout

The API endpoint to sign-out a user is declared in the following route.

```
mern-skeleton/server/routes/auth.routes.js:
```

```
| router.route('/auth/signout').get(authCtrl.signout)
```

When the Express app gets a GET request at '/auth/signout', it executes the `signout` controller function.

```
mern-skeleton/server/controllers/auth.controller.js:
```

```
const signout = (req, res) => {
  res.clearCookie("t")
  return res.status('200').json({
    message: "signed out"
  })
}
```

The `signout` function clears the response cookie containing the signed JWT. This is an optional endpoint and not really necessary for auth purposes if cookies are not used at all in the frontend.

With JWT, user state storage is the client's responsibility, and there are multiple options for client-side storage besides cookies. On signout, the client needs to delete the token on the client-side to establish that the user is no longer authenticated. On the server-side, we can use and verify the token that's generated at sign-in to protect routes that should not be accessed without valid authentication. In the next section, we will learn how to implement these protected routes using JWT.

Protecting routes with express-jwt

To protect access to the read, update, and delete routes, the server will need to check that the requesting client is actually an authenticated and authorized user.

To check whether the requesting user is signed in and has a valid JWT when a protected route is accessed, we will use the `express-jwt` module.



The `express-jwt` module is a piece of middleware that validates JSON Web Tokens. Run `yarn add express-jwt` from the command line to install `express-jwt`.

Protecting user routes

We will define two auth controller methods called `requireSignin` and `hasAuthorization`, both of which will be added to the user route declarations that need to be protected with authentication and authorization.

The read, update, and delete routes in `user.routes.js` need to be updated as follows.

mern-skeleton/server/routes/user.routes.js:

```
import authCtrl from '../controllers/auth.controller'  
...  
router.route('/api/users/:userId')  
  .get(authCtrl.requireSignin, userCtrl.read)  
  .put(authCtrl.requireSignin, authCtrl.hasAuthorization,  
    userCtrl.update)  
  .delete(authCtrl.requireSignin, authCtrl.hasAuthorization,  
    userCtrl.remove)  
...  
...
```

The route to read a user's information only needs authentication verification, whereas the update and delete routes should check for both authentication and authorization before these CRUD operations are executed. We will look into the implementation of the `requireSignin` method, which checks authentication, and the `hasAuthorization` method, which checks authorization, in the next section.

Requiring sign-in

The `requireSignin` method in `auth.controller.js` uses `express-jwt` to verify that the incoming request has a valid JWT in the `Authorization` header. If the token is valid, it appends the verified user's ID in an `'auth'` key to the request object; otherwise, it throws an authentication error.

mern-skeleton/server/controllers/auth.controller.js:

```
const requireSignin = expressJwt({
  secret: config.jwtSecret,
  userProperty: 'auth'
})
```

We can add `requireSignin` to any route that should be protected against unauthenticated access.

Authorizing signed in users

For some of the protected routes, such as update and delete, on top of checking for authentication we also want to make sure the requesting user is only updating or deleting their own user information.

To achieve this, the `hasAuthorization` function defined in `auth.controller.js` will check whether the authenticated user is the same as the user being updated or deleted before the corresponding CRUD controller function is allowed to proceed.

`mern-skeleton/server/controllers/auth.controller.js`:

```
const hasAuthorization = (req, res, next) => {
  const authorized = req.profile && req.auth
    && req.profile._id == req.auth._id
  if (!authorized) {
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}
```

The `req.auth` object is populated by `express-jwt` in `requireSignin` after authentication verification, while `req.profile` is populated by the `userByID` function in `user.controller.js`. We will add the `hasAuthorization` function to routes that require both authentication and authorization.

Auth error handling for express-jwt

To handle auth-related errors thrown by `express-jwt` when it tries to validate JWT tokens in incoming requests, we need to add the following error-catching code to the Express app configuration in `mern-skeleton/server/express.js`, near the end of the code, after the routes are mounted and before the app is exported:

```
app.use((err, req, res, next) => {
  if (err.name === 'UnauthorizedError') {
    res.status(401).json({ "error": err.name + ": " + err.message })
  } else if (err) {
    res.status(400).json({ "error": err.name + ": " + err.message })
    console.log(err)
  }
})
```

`express-jwt` throws an error named `UnauthorizedError` when a token cannot be validated for some reason. We catch this error here to return a `401` status back to the requesting client. We also add a response to be sent if other server-side errors are generated and caught here.

With user auth implemented for protecting routes, we have covered all the desired features of a working backend for our skeleton MERN application. In the next section, we will look at how we can check whether this standalone backend is functioning as desired without implementing a frontend.

Checking the standalone backend

There are a number of options when it comes to selecting tools to check backend APIs, ranging from the command-line tool curl ([http://github.com/curl/curl](https://github.com/curl/curl)) to **Advanced REST Client (ARC)** (<https://chrome.google.com/webstore/detail/advanced-rest-client/hgmlloofddfdnphfgcellkdfbfbjeloo>), a Chrome extension app with an interactive user interface.

To check the APIs that were implemented in this chapter, first, have the server running from the command line and use either of these tools to request the routes. If you are running the code on your local machine, the root URL is <http://localhost:3000/>.

Using ARC, we will showcase the expected behavior for five use cases so that we can check the implemented API endpoints.

Creating a new user

First, we will create a new user with the `/api/users` POST request and pass name, email, and password values in the request body. When the user is successfully created in the database without any validation errors, we will see a 200 OK success message, as shown in the following screenshot:

The screenshot shows a REST client interface with the following details:

- Method:** POST
- Request URL:** `http://localhost:3000/api/users/`
- Parameters:** A dropdown menu is open.
- Headers:** Body content type is set to `application/json`.
- Body:** Contains three properties:
 - name:** Jane Smith
 - email:** jane@smith.info
 - password:** abc987
- Variables:** None listed.
- Buttons:** SEND and more options (three dots).
- Status:** 200 OK (green button)
- Time:** 62.33 ms
- Details:** A dropdown menu.
- Tools:** Copy, Download, Refresh, Stop.
- Response Body:**

```
{  
  "message": "Successfully signed up!"  
}
```

You can also try to send the same request with invalid values for name, email, and password to check whether the relevant error messages are returned by the backend. Next, we will check whether the users were successfully created and stored in the database by calling the list users API.

Fetching the user list

We can see whether a new user is in the database by fetching a list of all users with a `GET` request to `/api/users`. The response should contain an array of all the user objects stored in the database:

Method Request URL
GET http://localhost:3000/api/users/ ▼ SEND ⋮

Parameters ^

Headers	Variables
<input type="checkbox"/> <input type="checkbox"/>	<input checked="" type="checkbox"/> Headers sets
Header name	Header value
Content-Type	application/json

[ADD HEADER](#)

A 30 bytes

200 OK 18.64 ms DETAILS ▾

```
[Array[2]
-0: {
  "_id": "5a1c7e411a692aa19c3e7b32",
  "name": "Jane Smith",
  "email": "jane@smith.info",
  "created": "2017-11-27T21:06:09.432Z"
},
-1: {
  "_id": "5a1c7ead1a692aa19c3e7b33",
  "name": "John Smith",
  "email": "john@smith.info",
  "created": "2017-11-27T21:07:57.922Z"
}
]
```

Notice how the returned user objects only show the `_id`, `name`, `email`, and `created` field values, and not the `salt` or `hashed_password` values, which are also present in the actual documents stored in the database. The request only retrieves the selected fields we specified in the

Mongoose `find` query that we made in the list controller method. This omission is also in place when fetching a single user.

Trying to fetch a single user

Next, we will try to access a protected API without signing in first. A `GET` request to read any one of the users will return a 401 Unauthorized error, such as in the following example. Here, a `GET` request to `/api/users/5a1c7ead1a692aa19c3e7b33` returns a 401 error:

Method Request URL
GET ▼ http://localhost:3000/api/users/5a1c7ead1a692aa19c3e7b33

SEND



Parameters ^

Headers

Variables



Headers sets

Header name

Header value

Content-Type

application/json



ADD HEADER



30 bytes

401 Unauthorized

14.92 ms

DETAILS ▼



{

 "message": "UnauthorizedError: No authorization token was found"

}

To make this request return a successful response with user details, a valid authorization token needs to be provided in the request header. We can generate a valid token by successfully calling the sign-in request.

Signing in

To be able to access the protected route, we will sign-in using the credentials of the user we created in the first example. To sign-in, a POST request is sent to `/auth/signin` with the email and password in the request body, as shown in the following screenshot:

Method Request URL
POST ▼ http://localhost:3000/auth/signin ▼ SEND :

Parameters ^

Headers	Body	Variables
Body content type application/json	Editor view JSON editor	
email	jane@smith.info	
password	abc987	

ADD PROPERTY

200 OK 18.10 ms

DETAILS ▾



```
{  
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI1YTFjN2U0MTFhNjkyYWExOWMzZTdiMzIiLC  
  JpYXQiOjE1MTE4MTcxNDd9.Y_267SQNjNHFTrgN-9FFxup9qy1jWeVAWoW7OrcFZGQ",  
  "user": {  
    "_id": "5a1c7e411a692aa19c3e7b32",  
    "name": "Jane Smith",  
    "email": "jane@smith.info"  
  }  
}
```

On successful sign-in, the server returns a signed JWT and user details. We will need this token to access the protected route for fetching a single user.

Fetching a single user successfully

Using the token received after sign-in, we can now access the protected route that failed previously. The token is set in the `Authorization` header in the Bearer scheme when making the GET request to `/api/users/5a1c7ead1a692aa19c3e7b33`. This time, the user object is returned successfully:

Method Request URL
GET <http://localhost:3000/api/users/5a1c7ead1a692aa19c3e7b33>

SEND



Parameters ^

Headers

Variables



Headers sets

Header name	Header value	X	Pencil	?
Content-Type	application/json	X	Pencil	?
Header name	Header value	X	Pencil	?
Authorization	Bearer eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9eyJfaWQiOiI1YTFjN2	X	Pencil	?

ADD HEADER



202 bytes

200 OK 12.61 ms

DETAILS ▾



```
{  
  "_id": "5a1c7ead1a692aa19c3e7b33",  
  "name": "John Smith",  
  "email": "john@smith.info",  
  "_v": 0,  
  "created": "2017-11-27T21:07:57.922Z"  
}
```

Using ARC as demonstrated in this section, you can also check the implementation of the other API endpoints for updating and deleting a user. With all these API endpoints working as expected, we have a complete working backend for MERN-based applications.

Summary

In this chapter, we developed a fully functioning standalone server-side application using Node, Express, and MongoDB and covered the first part of the MERN skeleton application. In the backend, we implemented a user model for storing user data, implemented with Mongoose; user API endpoints to perform CRUD operations, which were implemented with Express; and user auth for protected routes, which was implemented with JWT and `express-jwt`.

We also set up the development flow by configuring Webpack so that it compiles ES6+ code using Babel. We also configured Nodemon so that it restarts the server when the code changes. Finally, we checked the implementation of the APIs using the Advanced Rest API Client extension app for Chrome.

Now, we are ready to extend this backend application code and add the React frontend, which will complete the MERN skeleton application. We will do this in the next chapter.

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

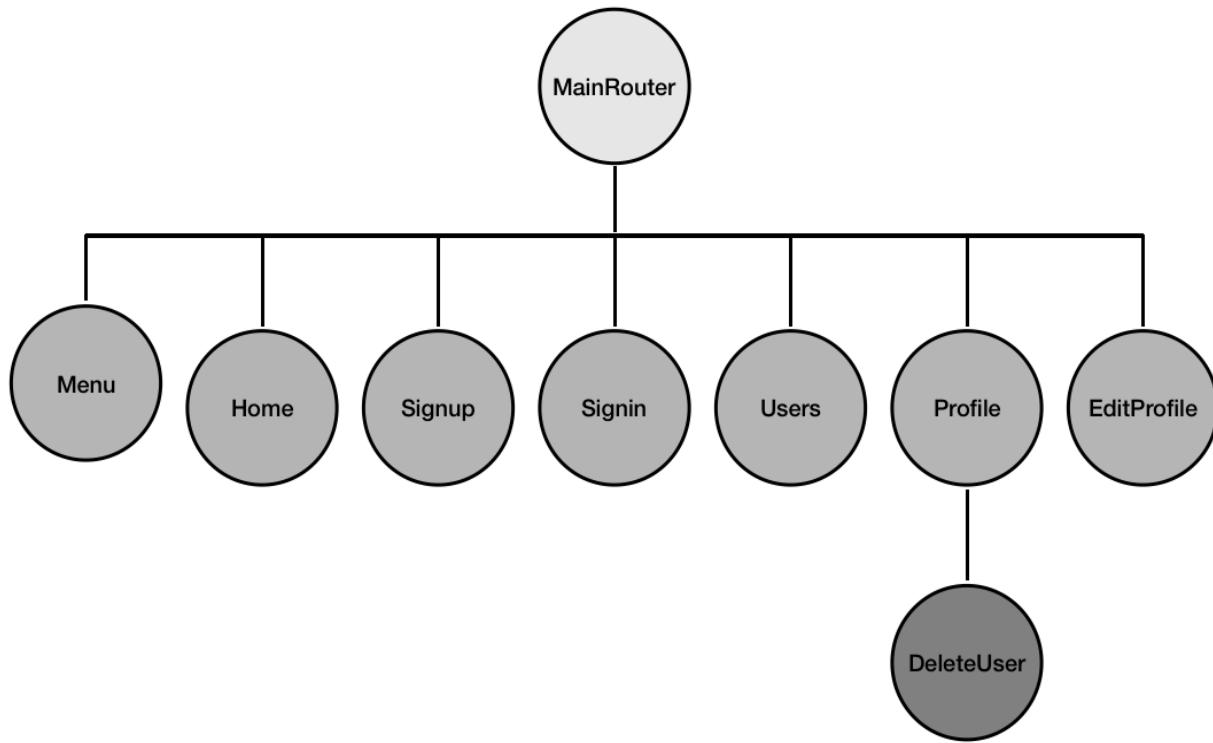
Defining the skeleton application frontend

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/Chapter%203%20and%204/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 file-loader  
| 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, 'dist')))
```

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"></script>
</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?  
| family=Roboto:100,300,400">  
| <link href="https://fonts.googleapis.com/icon?family=Material+Icons"  
| rel="stylesheet">
```

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.

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.



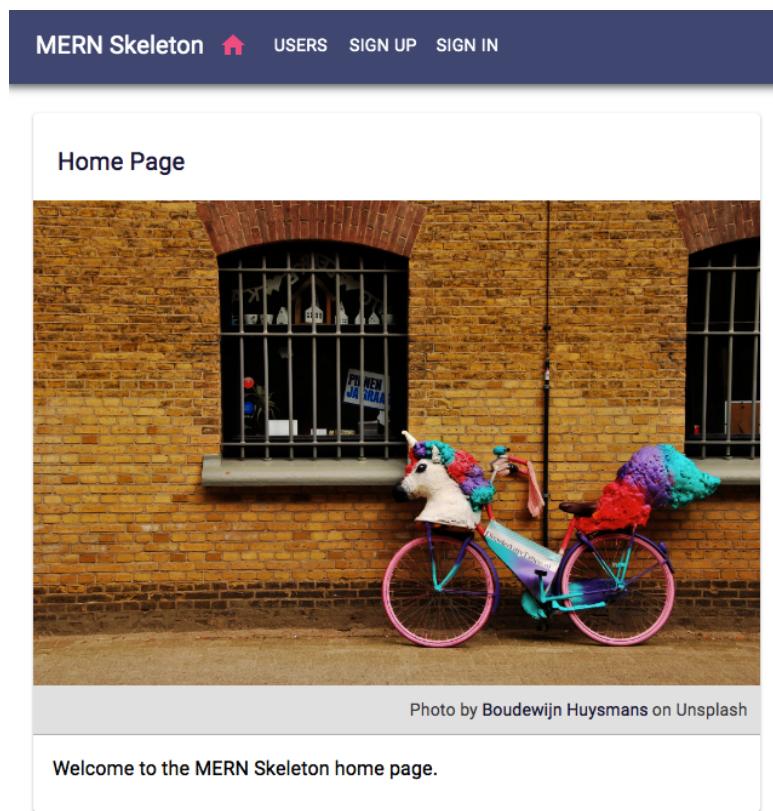
The `Switch` component in React Router renders a route exclusively. In other words, it only renders the first child that matches the requested route path. On the other hand, without being nested in a `Switch`, every `Route` component renders inclusively when there is a path match; for example, a request 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
${theme.spacing(2)}px`,
    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 Bicycle"/>
      <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 `window.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 00:00:00
                      UTC; path=/;"
  })
}
```

This `clearJWT` method takes a callback function as an argument, and it removes the JWT credential from `window.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 `window.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-workflow>, 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
- `Siginin`: 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:

The screenshot shows a dark blue header bar with the text "MERN Skeleton" and icons for Home, USERS, MY PROFILE, and SIGN OUT. Below the header is a white card with a title "All Users". The card lists seven user profiles, each consisting of a small circular icon with a person symbol, the user's name in blue text, and a right-pointing arrow indicating a link.

User Name	Action
Jane Smith	→
John Smith	→
Amy Pond	→
Jack Harkness	→
Donna Noble	→
Rory Williams	→
Bill Potts	→

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>
                        <ListItemIcon>
                            <Avatar>
                                <Person/>
                            </Avatar>
                        </ListItemIcon>
                        <ListItemText primary={item.name}>/>
                        <ListItemIcon>
                            <IconButton>
                                <ArrowForward/>
                            </IconButton>
                        </ListItemIcon>
                    </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}/>
```

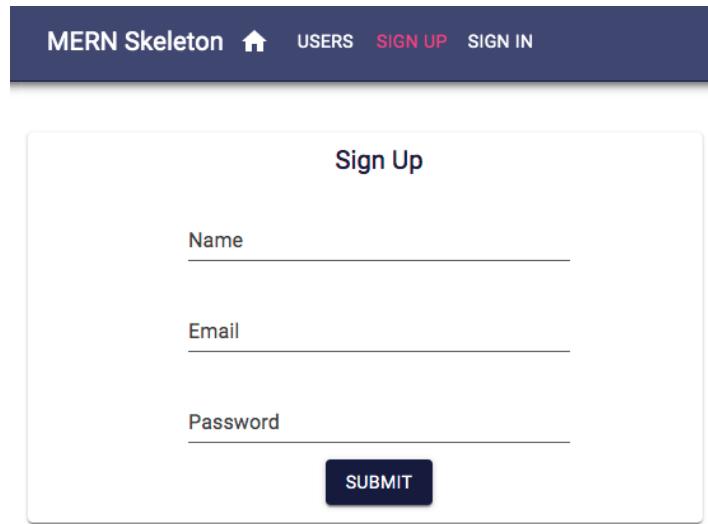
To see this view rendered in the browser, you can temporarily add a `Link` component to the `Home` component to be able to route to the `Users` component:

```
| <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="Password"
      className={classes.textField} value={values.password}
      onChange={handleChange('password')} margin="normal"/>
    <br/>
    {
      values.error && (<Typography component="p" color="error">
        <Icon color="error" className={classes.error}>error</Icon>
        {values.error}</Typography>
      )
    }
  </CardContent>
  <CardActions>
    <Button color="primary" variant="contained" onClick=
{clickSubmit}
      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`:

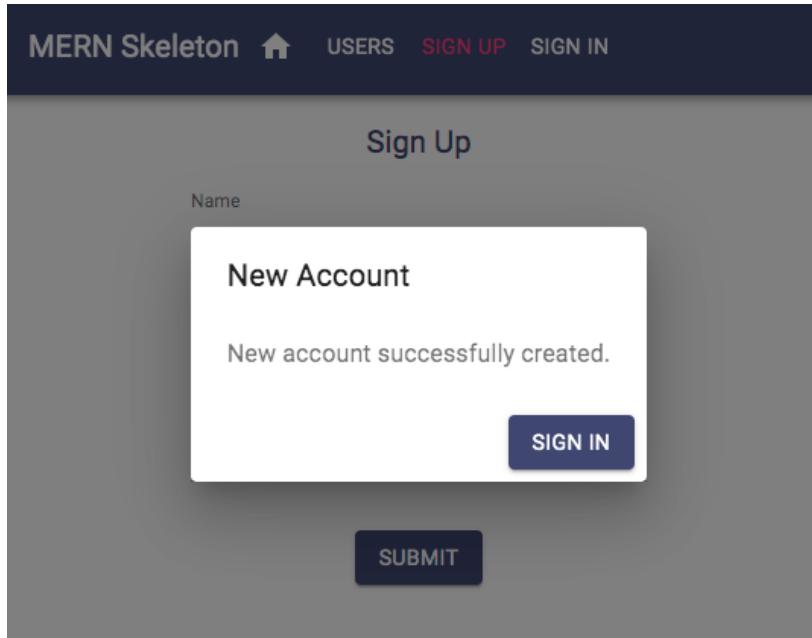
```

<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>

```

```
|           </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.

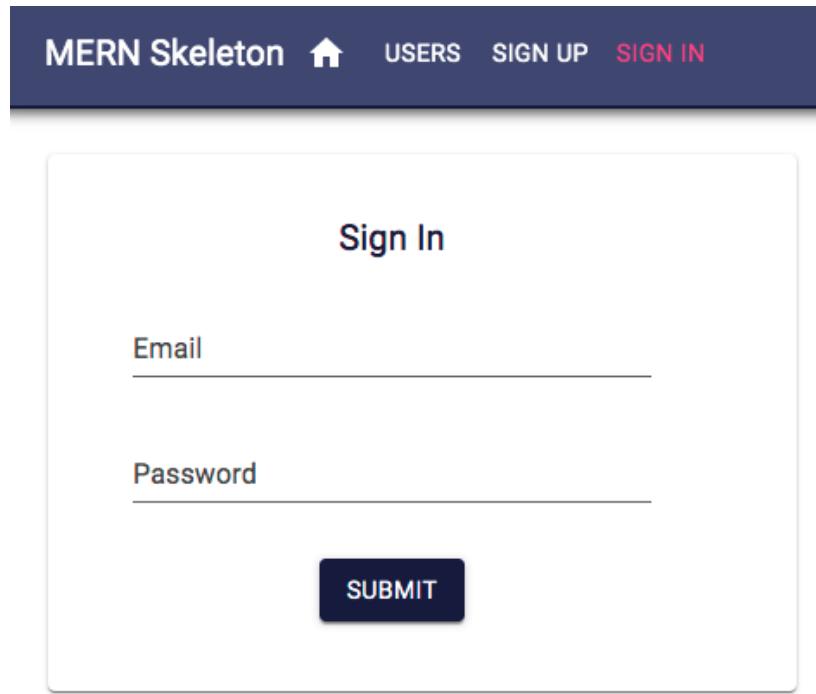
`mern-skeleton/client/MainRouter.js`:

```
| <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: true })  
      })  
    }  
  })  
}
```

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.

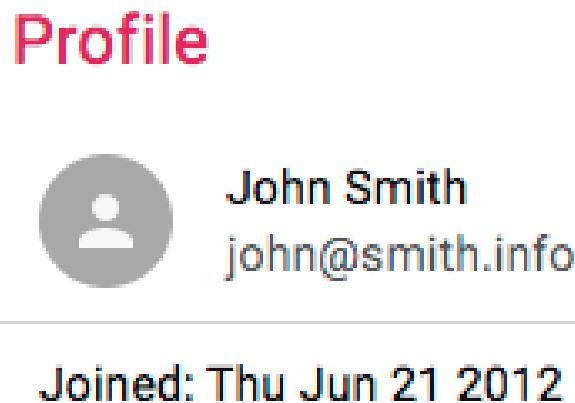
mern-skeleton/client/MainRouter.js:

```
| <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(false)
  ...
}
```

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' />  
| }  
|  
|
```

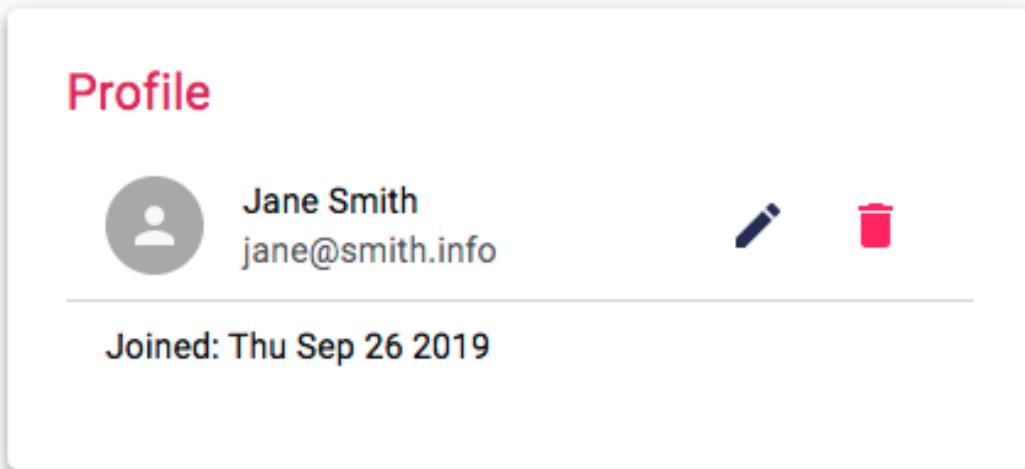
The function will return the `Profile` view with the following elements if the user who's currently signed in is viewing another user's profile.

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

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

MERN Skeleton  USERS MY PROFILE SIGN OUT



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._id ==  
  user._id &&  
  (<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.

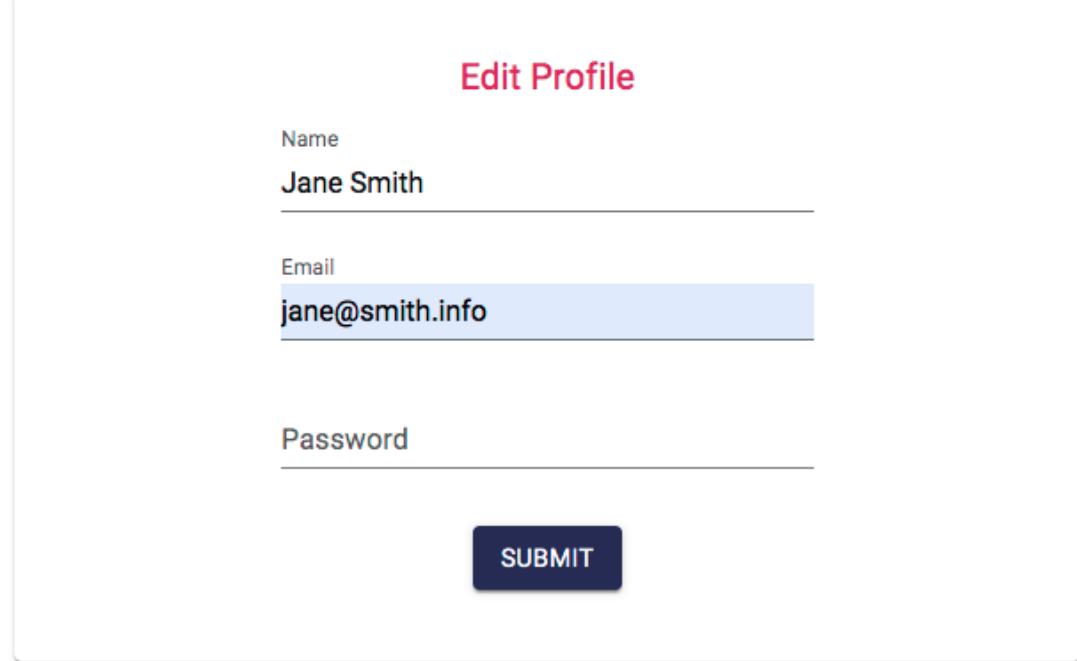
mern-skeleton/client/MainRouter.js:

```
| <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:



A screenshot of a web application interface. At the top, there is a dark blue header bar with the text "MERN Skeleton" and a house icon on the left, and "USERS", "MY PROFILE", and "SIGN OUT" on the right. Below the header is a white rectangular form with rounded corners. The form has a title "Edit Profile" in red at the top center. It contains three input fields: "Name" with the value "Jane Smith", "Email" with the value "jane@smith.info" (which is highlighted with a light blue background), and "Password" (which is currently empty). At the bottom of the form is a dark blue "SUBMIT" button.

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, redirectToProfile: true})
    }
  })
}
```

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:

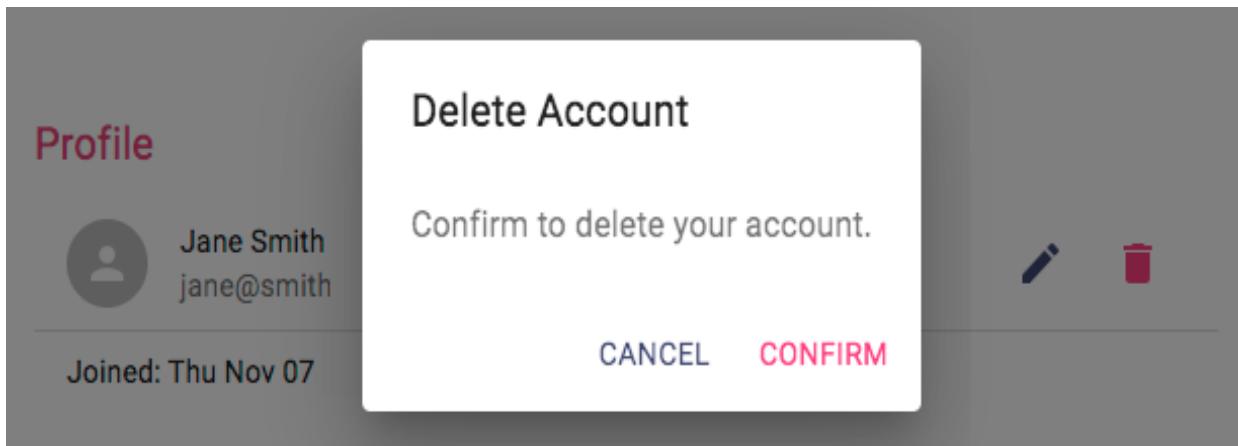
```
| <Switch>
|   ...
|     <PrivateRoute path="/user/edit/:userId" component={EditProfile}/>
|     <Route path="/user/:userId" component={Profile}/>
|   </Switch>
```

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</Button>
      </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.

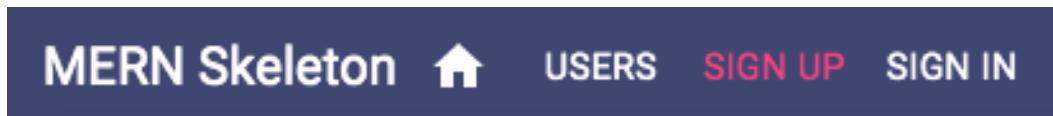
mern-skeleton/client/core/Menu.js:

```
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 </Button>  
    </Link>  
    <Link to="/signin">  
      <Button style={isActive(history, "/signin")}> SIGN IN </Button>  
    </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 ServerStyleSheets
    // 2. Use renderToString to generate markup which renders
        // components specific to the route requested
    // 3. Return template with markup and CSS styles in the response
})
```

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/styles'  
| 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 `ServerStyleSheet`s and React's `renderToString`.

On every request received by the Express app, we will create a new `ServerStyleSheet` 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 ServerStyleSheet()
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.

Growing the Skeleton into a Social Media Application

Social media is an integral part of the web these days, and many of the user-centric web applications we build end up requiring a social component down the line to drive user engagement.

For our first real-world MERN application, we will modify the MERN skeleton application we developed in [Chapter 3](#), *Building a Backend with MongoDB, Express, and Node*, and [Chapter 4](#), *Adding a React Frontend to Complete MERN*, to build a simple social media application in this chapter. While doing this, you will learn how to extend the integration of the MERN stack technologies and add new features to grow your own full-stack web applications.

In this chapter, we will go over the following topics:

- Introducing MERN Social
- Updating the user profile
- Following users in MERN Social
- Posting messages with photos
- Implementing interactions on posts

Introducing MERN Social

MERN Social is a social media application with rudimentary features inspired by existing social media platforms such as Facebook and Twitter. The main purpose of this application is to demonstrate how to use the MERN stack technologies to implement features that allow users to connect or follow each other, and interact over shared content. While building out MERN Social in this chapter, we will go over the implementation of the following social media-flavored features:

- User profile with a description and a photo
- Users following each other
- Who to follow suggestions
- Posting messages with photos
- Newsfeed with posts from followed users
- Listing posts by user
- Liking posts
- Commenting on posts

You can extend these implementations further, as desired, for more complex features. The MERN Social home page looks as follows:

Newsfeed



Cecil McQueen

Share your thoughts ...

0

POST



Violet Bernstein

Fri Dec 22 2017

"Injustice anywhere is a threat to justice everywhere." - Martin Luther King, Jr.

1

3



1



Write something ...



Cecil McQueen



Truth!

Fri Dec 22 2017 | 

Who to follow



Dominic Kotsopolis

0

FOLLOW



Daniel Nakamura

0

FOLLOW



Delbert Catessen

0

FOLLOW



Antonia Thatcher

0

FOLLOW



Jack Michaels

0

FOLLOW



Thomas Brogan

0

FOLLOW



Katherine Hemstridge

0

FOLLOW



Jamie Woolcraft

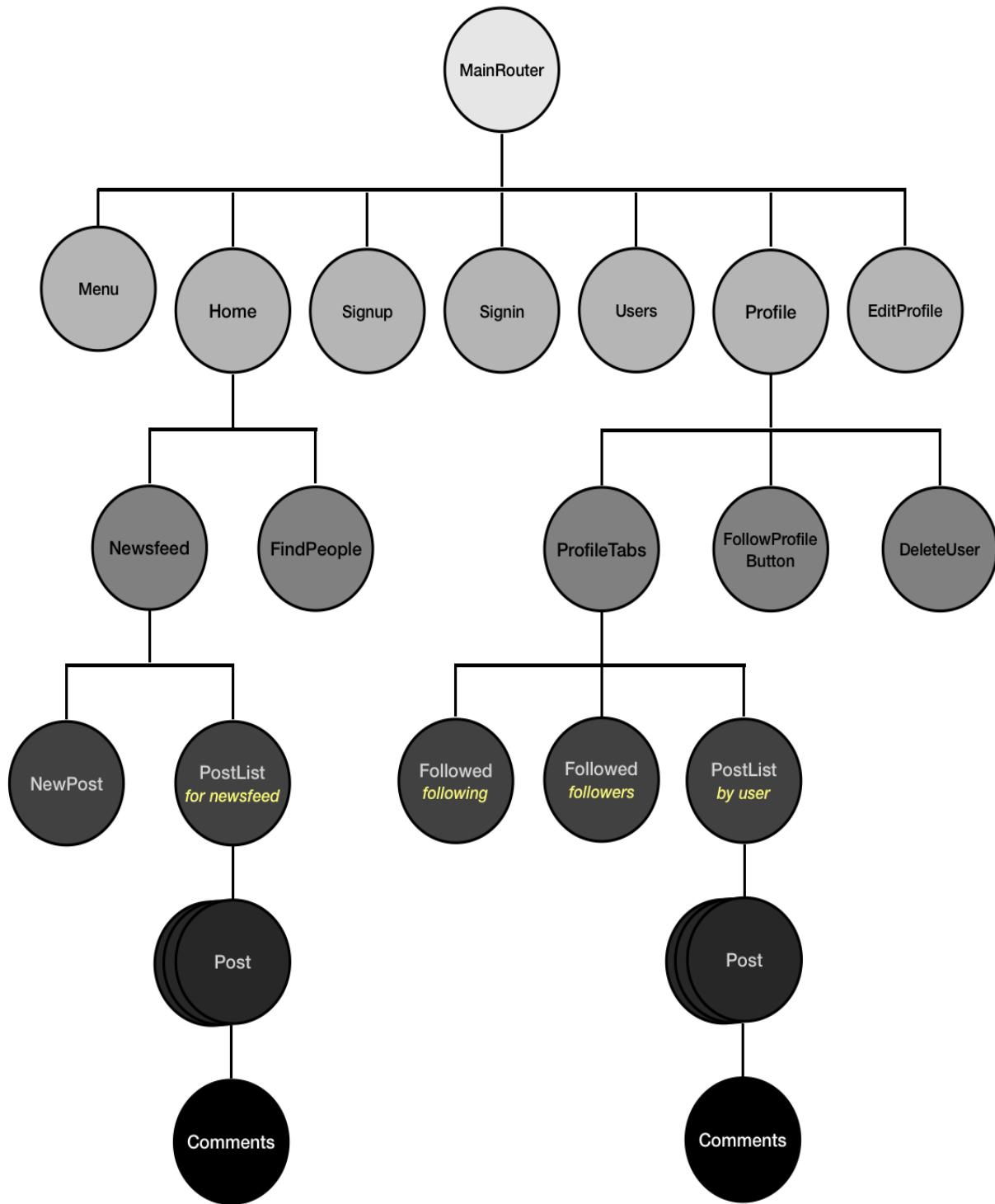
0

FOLLOW



The code for the complete MERN Social application is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter05/mern-social>. You can clone this code and run the application as you go through the code explanations for the rest of this chapter.

The views needed for the MERN Social application will be developed by extending and modifying the existing React components in the MERN skeleton application. The following component tree shows all the custom React components that make up the MERN Social frontend and also exposes the composition structure we will use to build out the views in the rest of this chapter:



Besides updating the existing components, we will also add new custom components to compose views, including a Newsfeed view

where the user can create a new post and also browse a list of all the posts from people they follow on MERN Social. In the next section, we will begin by updating the user profile to demonstrate how to upload a profile photo and add a short bio for each user on the platform.

Updating the user profile

The existing skeleton application only has support for a user's name, email, and password. But in MERN Social, we will allow users to add a description about themselves, as well as upload a profile photo while editing the profile after signing up, as shown in the following screenshot:

Edit Profile



UPLOAD 

Jamie.png

Name
Jamie Woolcraft

About
An expert combatant with a prodigious talent for solving mysteries

Email
jamie@wool.info

Password

In order to implement this feature update, we need to modify both the user backend and frontend. In the following sections, we will learn how to update the user model and user update API in the backend, and then the user profile and user profile edit form views in

the frontend to add a short description and a profile photo for users in MERN Social.

Adding an about description

To store the short description that's entered in the `about` field by a user, we need to add an `about` field to the user model in

`server/models/user.model.js`:

```
| about: {  
|   type: String,  
|   trim: true  
| }
```

Then, to get the description as input from the user, we need to add a multiline `TextField` to the `EditProfile` form and handle the value change the same way we did for the user's name input.

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

```
| <TextField  
|   id="multiline-flexible"  
|   label="About"  
|   multiline  
|   rows="2"  
|   value={values.about}  
|   onChange={handleChange('about')}  
| />
```

Finally, to show the description text that was added to the `about` field on the user profile page, we can add it to the existing profile view.

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

```
| <ListItem> <ListItemText primary={this.state.user.about}/> </ListItem>
```

With this modification to the user feature in the MERN skeleton code, users can now add and update a description about themselves to be displayed on their profiles. Next, we will add the ability to upload a photo to complete the user profile.

Uploading a profile photo

Allowing a user to upload a profile photo will require that we store the uploaded image file and retrieve it on request to load it in the view. There are multiple ways of implementing this upload feature while considering the different file storage options:

- **Server filesystem:** Upload and save files to a server filesystem and store the URL in MongoDB.
- **External file storage:** Save files to external storage such as Amazon S3 and store the URL in MongoDB.
- **Store as data in MongoDB:** Save files that are small in size (less than 16 MB) to MongoDB as data of the Buffer type.

For MERN Social, we will assume that the photo files that are uploaded by the user will be small in size and demonstrate how to store these files in MongoDB for the profile photo upload feature. In [Chapter 8](#), *Extending the Marketplace for Orders and Payments*, we will discuss how to store larger files in MongoDB using GridFS.

To implement this photo upload feature, in the following sections, we will do the following:

- Update the user model to store the photo.
- Integrate updated frontend views to upload the photo from the client- side.
- Modify the user update controller in the backend to process the uploaded photo.

Updating the user model to store a photo in MongoDB

In order to store the uploaded profile photo directly in the database, we will update the user model to add a `photo` field that stores the file as data of the `Buffer` type, along with the file's `contentType`.

mern-social/server/models/user.model.js:

```
| photo: {  
|   data: Buffer,  
|   contentType: String  
| }
```

An image file that's uploaded by the user from the client- side will be converted into binary data and stored in this `photo` field for documents in the Users collection in MongoDB. Next, we will look at how to upload the file from the frontend.

Uploading a photo from the edit form

Users will be able to upload an image file from their local files when editing the profile. In order to implement this interaction, we will update the `EditProfile` component in `client/user/EditProfile.js` with an upload photo option and then attach the user selected file in the form data that's submitted to the server. We will discuss this in the following sections.

File input with Material-UI

We will utilize the HTML5 file input type to let the user select an image from their local files. The file input will return the filename in the change event when the user selects a file. We will add the file input element to the edit profile form as follows:

mern-social/client/user/EditProfile.js:

```
| <input accept="image/*" type="file"
|   onChange={handleChange('photo')}
|   style={{display:'none'}}
|   id="icon-button-file" />
```

To integrate this file `input` element with Material-UI components, we apply `display:none` to hide the `input` element from the view, then add a Material-UI button inside the label for this file input. This way, the view displays the Material-UI button instead of the HTML5 file input element. The `label` is added as follows:

mern-social/client/user/EditProfile.js:

```
| <label htmlFor="icon-button-file">
|   <Button variant="contained" color="default" component="span">
|     Upload <FileUpload/>
|   </Button>
| </label>
```

When the Button's `component` prop is set to `span`, the `Button` component renders as a `span` element inside the `label` element. A click on the `Upload` span or label is registered by the file input with the same ID as the label, and as a result, the file select dialog is opened. Once the user selects a file, we can set it to state in the call to `handleChange(...)` and display the name in the view, as shown in the following code.

mern-social/client/user/EditProfile.js:

```
| <span className={classes.filename}>
|   {values.photo ? values.photo.name : ''}
```

|

This way, the user will see the name of the file they are trying to upload as the profile photo. With the file selected for uploading, next, we have to attach and send this file with the request to the server to update the user information in the database.

Form submission with the file attached

Uploading files to the server with a form requires a multipart form submission. This is in contrast to the stringified object we sent in previous implementations of `fetch`. We will modify the `EditProfile` component so that it uses the `FormData` API to store the form data in the format needed for encoding in the `multipart/form-data` type.



You can learn more about the `FormData` API at developer.mozilla.org/en-US/docs/Web/API/FormData.

First, we will update the `input` `handleChange` function so that we can store input values for both the text fields and the file input, as shown in the following code.

mern-social/client/user/EditProfile.js:

```
const handleChange = name => event => {
  const value = name === 'photo'
    ? event.target.files[0]
    : event.target.value
  setValues({...values, [name]: value })
}
```

Then, on form submission, we need to initialize `FormData` and append the values from the fields that were updated, as shown here.

mern-social/client/user/EditProfile.js:

```
const clickSubmit = () => {
  let userData = new FormData()
  values.name && userData.append('name', values.name)
  values.email && userData.append('email', values.email)
  values.password && userData.append('password', values.password)
  values.about && userData.append('about', values.about)
  values.photo && userData.append('photo', values.photo)
  ...
}
```

After appending all the fields and values to it, `userData` is sent with the `fetch` API call to update the user, as shown in the following code.

mern-social/client/user/EditProfile.js:

```
update({
  userId: match.params.userId
}, {
  t: jwt.token
}, userData).then((data) => {
  if (data && data.error) {
    setValues({...values, error: data.error})
  } else {
    setValues({...values, 'redirectToProfile': true})
  }
})
```

Since the content type of the data that's sent to the server is no longer `'application/json'`, we also need to modify the `update` `fetch` method in `api-user.js` to remove `Content-Type` from the headers in the `fetch` call, as shown here.

mern-social/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',
        'Authorization': 'Bearer ' + credentials.t
      },
      body: user
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

Now, if the user chooses to upload a profile photo when editing the profile, the server will receive a request with the file attached, along with the other field values. Next, we need to modify the server-side code to be able to process this request.

Processing a request containing a file upload

On the server, to process the request to the update API that may now contain a file, we will use the `formidable` Node module. Run the following command from the command line to install `formidable`:

```
| yarn add formidable
```

The `formidable` will allow the server to read the `multipart` form data and give us access to the fields and the file, if there are any. If there is a file, `formidable` will store it temporarily in the filesystem. We will read it from the filesystem using the `fs` module, which will retrieve the file type and data, and store it in the `photo` field in the user model. The `formidable` code will go in the `update` controller in `user.controller.js`, as follows.

mern-social/server/controllers/user.controller.js:

```
import formidable from 'formidable'
import fs from 'fs'
const update = async (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        error: "Photo could not be uploaded"
      })
    }
    let user = req.profile
    user = extend(user, fields)
    user.updated = Date.now()
    if(files.photo) {
      user.photo.data = fs.readFileSync(files.photo.path)
      user.photo.contentType = files.photo.type
    }
    try {
      await user.save()
      user.hashed_password = undefined
      user.salt = undefined
      res.json(user)
    } catch (err) {
      console.log(err)
      res.status(500).json({
        error: "Database error"
      })
    }
  })
}
```

```
    } catch (err) {
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    })
}
```

This will store the uploaded file as data in the database. Next, we will set up file retrieval so that we can access and display the photo that's uploaded by the user in the frontend views.

Retrieving a profile photo

The simplest option to retrieve the image stored in the database and then show it in a view is to set up a route that will fetch the data and return it as an image file to the requesting client. In this section, we will learn how to set up this route to expose a photo URL, as well as how to use this URL to display the photo in the frontend views.

Profile photo URL

We will set up a route to the photo stored in the database for each user, and also add another route that will fetch a default photo if the given user did not upload a profile photo. These routes will be defined as follows.

mern-social/server/routes/user.routes.js:

```
router.route('/api/users/photo/:userId')
  .get(userCtrl.photo, userCtrl.defaultPhoto)
router.route('/api/users/defaultphoto')
  .get(userCtrl.defaultPhoto)
```

We will look for the photo in the `photo` controller method and, if found, send it in the response to the request at the photo route; otherwise, we'll call `next()` to return the default photo, as shown in the following code.

mern-social/server/controllers/user.controller.js:

```
const photo = (req, res, next) => {
  if(req.profile.photo.data) {
    res.set("Content-Type", req.profile.photo.contentType)
    return res.send(req.profile.photo.data)
  }
  next()
}
```

The default photo is retrieved and sent from the server's file system, as shown here.

mern-social/server/controllers/user.controller.js:

```
import profileImage from './../../../client/assets/images/profile-pic.png'
const defaultPhoto = (req, res) => {
  return res.sendFile(process.cwd() + profileImage)
}
```

We can use the route defined here to display the photo in the views, as described in the next section.

Showing a photo in a view

With the photo URL routes set up to retrieve the photo, we can simply use these in the `img` element's `src` attribute to load the photo in the view. For example, in the `Profile` component, we use the user ID from the `values` in the state to construct the photo URL, as shown in the following code.

mern-social/client/user/Profile.js:

```
| const photoUrl = values.user._id  
|   ? `/api/users/photo/${values.user._id}?${new Date().getTime()}`  
|   : '/api/users/defaultphoto'
```

To ensure the `img` element reloads in the `Profile` view after the photo is updated, we have to add a time value to the photo URL to bypass the browser's default image caching behavior.

Then, we can set the `photoUrl` to the Material-UI `Avatar` component, which renders the linked image in the view:

```
| <Avatar src={photoUrl}/>
```

The updated user profile in MERN Social can now display a user uploaded profile photo and an `about` description, as shown in the following screenshot:

Profile



Jamie Woolcraft
jamie@wool.info



An expert combatant with a prodigious talent for solving mysteries

Joined: Sat Dec 23 2017

We have successfully updated the MERN skeleton application code to let users upload a profile photo and add a short bio description to their profiles. In the next section, we will update this further and implement the social media flavored feature that allows users to follow each other.

Following users in MERN Social

In MERN Social, users will be able to follow each other. Each user will have a list of followers and a list of people they follow. Users will also be able to see a list of users they can follow; in other words, the users in MERN Social they are not already following. In the following sections, we will learn how to update the full-stack code to implement these features.

Following and unfollowing

In order to keep track of which user is following which other users, we will have to maintain two lists for each user. When one user follows or unfollows another user, we will update one's `following` list and the other's `followers` list. First, we will update the backend to store and update these lists, then modify the frontend views to allow users to perform follow and unfollow actions.

Updating the user model

To store the list of `following` and `followers` in the database, we will need to update the user model with two arrays of user references, as shown in the following code.

`mern-social/server/models/user.model.js`:

```
| following: [{type: mongoose.Schema.ObjectId, ref: 'User'}],  
| followers: [{type: mongoose.Schema.ObjectId, ref: 'User'}]
```

These references will point to the users in the collection being followed by or following the given user. Next, we will update the user controllers to ensure the details of the users that are referenced in these lists are returned in a response to client-side requests.

Updating the userByID controller method

When a single user is retrieved from the backend, we want the `user` object to include the names and IDs of the users referenced in the `following` and `followers` arrays. To retrieve these details, we need to update the `userByID` controller method so that it populates the returned `user` object, as shown in the highlighted code.

mern-social/server/controllers/user.controller.js:

```
const userByID = async (req, res, next, id) => {
  try {
    let user = await User.findById(id)
      .populate('following', '_id name')
      .populate('followers', '_id name')
      .exec()
    if (!user)
      return res.status('400').json({
        error: "User not found"
      })
    req.profile = user
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve user"
    })
  }
}
```

We use the Mongoose `populate` method to specify that the `user` object that's returned from the query should contain the name and ID of the users referenced in the `following` and `followers` lists. This will give us the names and IDs of the user references in the `followers` and `following` lists when we fetch the user with the read API call.

With the user model updated, we are ready to add API endpoints that will update these lists to either add or remove users from the lists, as discussed in the next section.

Adding APIs to follow and unfollow

When a user follows or unfollows another user from the view, both users' records in the database will be updated in response to the `follow` or `unfollow` requests.

Set up `follow` and `unfollow` routes in `user.routes.js` as follows.

mern-social/server/routes/user.routes.js:

```
router.route('/api/users/follow')
  .put(authCtrl.requireSignin,
        userCtrl.addFollowing,
        userCtrl.addFollower)
router.route('/api/users/unfollow')
  .put(authCtrl.requireSignin,
        userCtrl.removeFollowing,
        userCtrl.removeFollower)
```

The `addFollowing` controller method in the user controller will update the `following` array for the current user by pushing the followed user's reference into the array, as shown in the following code.

mern-social/server/controllers/user.controller.js:

```
const addFollowing = async (req, res, next) => {
  try{
    await User.findByIdAndUpdate(req.body.userId,
                            {$push: {following: req.body.followId}})
    next()
  }catch(err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

On successful update of the `following` array, `next()` is invoked, and as a result, the `addFollower` method is executed to add the current user's

reference to the followed user's `followers` array. The `addFollower` method is defined as follows.

mern-social/server/controllers/user.controller.js:

```
const addFollower = async (req, res) => {
  try{
    let result = await User.findByIdAndUpdate(req.body.followId,
                                              {$push: {followers: req.body.userId}},
                                              {new: true})
      .populate('following', '_id name')
      .populate('followers', '_id name')
      .exec()
    result.hashed_password = undefined
    result.salt = undefined
    res.json(result)
  }catch(err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

For unfollowing, the implementation is similar. The `removeFollowing` and `removeFollower` controller methods update the respective `'following'` and `'followers'` arrays by removing the user references with `$pull` instead of `$push`. `removeFollowing` and `removeFollower` will look as follows.

mern-social/server/controllers/user.controller.js:

```
const removeFollowing = async (req, res, next) => {
  try{
    await User.findByIdAndUpdate(req.body.userId,
                             {$pull: {following: req.body.unfollowId}})
    next()
  }catch(err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
const removeFollower = async (req, res) => {
  try{
    let result = await User.findByIdAndUpdate(req.body.unfollowId,
                                           {$pull: {followers: req.body.userId}},
                                           {new: true})
      .populate('following', '_id name')
      .populate('followers', '_id name')
      .exec()
    result.hashed_password = undefined
    result.salt = undefined
  }
```

```
    res.json(result)
  }catch(err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The user backend on the server-side is ready for the follow and unfollow features. Next, we will update the frontend to utilize these new backend APIs and complete this feature.

Accessing the follow and unfollow APIs in views

In order to access these API calls in the views, we will update `api-user.js` with the `follow` and `unfollow` fetch methods. The `follow` and `unfollow` methods will be similar, making calls to the respective routes with the current user's ID and credentials, and the followed or unfollowed user's ID. The `follow` method will be as follows.

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

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

The `unfollow` fetch method is similar; it takes the unfollowed user's ID and calls the `unfollow` API, as shown in the following code.

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

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

```
    unfollowId})
  })
  return await response.json()
} catch(err) {
  console.log(err)
}
}
```

With the API fetch code implemented, we can use these two methods to integrate the backend updates in the views, as discussed in the next section, which will allow users to follow or unfollow another user in the application.

Follow and unfollow buttons

The button that will allow a user to follow or unfollow another user will appear conditionally, depending on whether the user is already followed or not by the current user, as shown in the following screenshot:



In the following sections, we will add this button in a separate React component, integrate it with the existing user profile view, and connect it to the follow and unfollow fetch methods.

The FollowProfileButton component

We will create a separate component for the follow button called `FollowProfileButton`, which will be added to the `Profile` component. This component will show the `Follow` or `Unfollow` button, depending on whether the current user is already a follower of the user in the profile. The `FollowProfileButton` component will look as follows.

mern-social/client/user/FollowProfileButton.js:

```
export default function FollowProfileButton (props) {
  const followClick = () => {
    props.onButtonClick(follow)
  }
  const unfollowClick = () => {
    props.onButtonClick(unfollow)
  }
  return (<div>
    { props.following
      ? (<Button variant="contained" color="secondary"
          onClick={unfollowClick}>Unfollow</Button>)
      : (<Button variant="contained" color="primary"
          onClick={followClick}>Follow</Button>)
    }
  </div>
)
FollowProfileButton.propTypes = {
  following: PropTypes.bool.isRequired,
  onButtonClick: PropTypes.func.isRequired
}
```

When `FollowProfileButton` is added to the profile, the `following` value will be determined and sent from the `Profile` component as a prop to `FollowProfileButton`, along with the click handler that takes the specific `follow` or `unfollow` fetch API to be called as a parameter. The resulting profile views will look as follows:

Profile



Thomas Brogan
ace@brogan.info

FOLLOW

A hard-as-nails, feet-on-the-desk private detective - the guy you call when the wolf's at the door and there's nowhere left to run.
Joined: Thu Oct 19 2017

Profile



Antonia Thatcher
iron@lady.info

UNFOLLOW

Iron Lady
Joined: Mon Sep 04 2017

In order to integrate this `FollowProfileButton` component with the profile view, we need to update the existing `Profile` component, as discussed next.

Updating the Profile component

In the `Profile` view, `FollowProfileButton` should only be shown when the user views the profile of other users, so we need to modify the condition for showing the `Edit` and `Delete` buttons when viewing a profile, as follows:

```
{auth.isAuthenticated().user &&
  auth.isAuthenticated().user._id == values.user._id
  ? (edit and delete buttons)
  : (follow button)
}
```

In the `Profile` component, after the user data is successfully fetched in `useEffect`, we will check whether the signed-in user is already following the user in the profile or not and set the `following` value to the respective state, as shown in the following code.

mern-social/client/user/Profile.js:

```
let following = checkFollow(data)
setValues({...values, user: data, following: following})
```

To determine the value to set in `following`, the `checkFollow` method will check if the signed-in user exists in the fetched user's `followers` list, then return `match` if found; otherwise, it will return `undefined` if a match is not found. The `checkFollow` method is defined as follows.

mern-social/client/user/Profile.js:

```
const checkFollow = (user) => {
  const match = user.followers.some((follower)=> {
    return follower._id == jwt.user._id
  })
  return match
}
```

The `Profile` component will also define the click handler for `FollowProfileButton` so that the state of the `Profile` can be updated when

the follow or unfollow action completes, as shown in the following code.

mern-social/client/user/Profile.js:

```
const clickFollowButton = (callApi) => {
  callApi({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, values.user._id).then((data) => {
    if (data.error) {
      setValues({...values, error: data.error})
    } else {
      setValues({...values, user: data, following: !values.following})
    }
  })
}
```

The click handler definition takes the fetch API call as a parameter and is passed as a prop to `FollowProfileButton`, along with the `following` value when it is added to the `Profile` view, as follows.

mern-social/client/user/Profile.js:

```
<FollowProfileButton following={this.state.following} onClick={this.clickFollowButton} />
```

This will load `FollowProfileButton` into the profile view, with all the necessary conditions accounted for, and provide the current user the option to follow or unfollow other users in the MERN Social application. Next, we will extend this feature to allow users to view the list of followings or followers in the user profile view.

Listing followings and followers

In order to give users easy access to the users they are following and the users who are following them on MERN Social, we will add these lists to their profile views. In each user's profile, we will add a list of their followers and the people they are following, as shown in the following screenshot:

Profile

The screenshot shows a user profile for 'Daniel Nakamura' (black@cobra.info). It includes a profile picture, the user's name and email, and a 'FOLLOW' button. Below the profile information, there is a bio ('Fight for freedom') and a 'Joined' date ('Mon Sep 04 2017'). At the bottom, there are three tabs: 'POSTS' (disabled), 'FOLLOWING' (selected), and 'FOLLOWERS'. Under the 'FOLLOWING' tab, four user profiles are listed with their names and pictures: Thomas Brogan, Jamie Woolcraft, Cecil McQueen, and Jack Michaels.

Following User	Profile Picture	Name
Thomas Brogan		Thomas Brogan
Jamie Woolcraft		Jamie Woolcraft
Cecil McQueen		Cecil McQueen
Jack Michaels		Jack Michaels

The details of the users referenced in the `following` and `followers` lists are already in the user object that is fetched using the `read` API when the profile is loaded. In order to render these separate lists of followers and followings, we will create a new component called

`FollowGrid`.

Making a FollowGrid component

The `FollowGrid` component will take a list of users as props, display the avatars of the users with their names, and link them to each user's own profile. We can add this component to the `Profile` view to display followings or followers. The `FollowGrid` component is defined as follows.

mern-social/client/user/FollowGrid.js:

```
export default function FollowGrid (props) {
  const classes = useStyles()
  return (<div className={classes.root}>
    <GridList cellHeight={160} className={classes.gridList} cols={4}>
      {props.people.map((person, i) => {
        return <GridListTile style={{'height':120}} key={i}>
          <Link to={"/user/" + person._id}>
            <Avatar src={'/api/users/photo/' + person._id}
              className={classes.bigAvatar}/>
            <Typography className={classes.tileText}>
              {person.name}
            </Typography>
          </Link>
        </GridListTile>
      ))}
    </GridList>
  </div>
}

FollowGrid.propTypes = {
  people: PropTypes.array.isRequired
}
```

To add the `FollowGrid` component to the `Profile` view, we can place it as desired in the view and pass the list of `followers` or `followings` as the `people` prop:

```
<FollowGrid people={props.user.followers}/>
<FollowGrid people={props.user.following}/>
```

As shown previously, in MERN Social, we chose to display the `FollowGrid` components in tabs within the `Profile` component. We created a separate `ProfileTabs` component using Material-UI tab components and added that to the `Profile` component. This `ProfileTabs` component contains the two `FollowGrid` components with following and followers lists, along with a `PostList` component that shows the posts by the user.

This `PostList` component will be discussed later in this chapter. In the next section, we will add a feature that will allow a user to discover other users on the platform who they are not following yet.

Finding people to follow

The Who to follow feature will show the signed-in user a list of people in MERN Social that they are not currently following, thus giving them the option to follow them or view their profiles, as shown in the following screenshot:

Who to follow

	Dominic Kotsopolis	 FOLLOW
	Antonia Thatcher	 FOLLOW
	Jack Michaels	 FOLLOW
	Thomas Brogan	 FOLLOW
	Katherine Hemstridge	 FOLLOW

To implement this feature, we need to add a backend API that returns the list of users not followed by the currently signed-in user, and then update the frontend by adding a component that loads and displays this list of users.

Fetching users not followed

We will implement a new API on the server to query the database and fetch the list of users the current user is not following. This route will be defined as follows.

mern-social/server/routes/user.routes.js:

```
| router.route('/api/users/findpeople/:userId')
|   .get(authCtrl.requireSignin, userCtrl.findPeople)
```

In the `findPeople` controller method, we will query the User collection in the database to find the users that are not in the current user's following list.

mern-social/server/controllers/user.controller.js:

```
| const findPeople = async (req, res) => {
|   let following = req.profile.following
|   following.push(req.profile._id)
|   try {
|     let users = await User.find({ _id: { $nin: following } })
|       .select('name')
|     res.json(users)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

This query will return an array of users that are not followed by the current user. To use this list of users in the frontend, we will update the `api-user.js` file and add a fetch for this API. The `findPeople` fetch method is defined as follows.

mern-social/client/user/api-user.js:

```
| const findPeople = async (params, credentials, signal) => {
|   try {
|     let response = await fetch('/api/users/findpeople/' + 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)
}
}
```

We can use this `findPeople` fetch method in the component that will display this list of users. In the next section, we will create the `FindPeople` component for this purpose.

The FindPeople component

To display the *who to follow* feature, we will create a component called `FindPeople`, which can be added to any of the views or rendered on its own. In this component, we will fetch the users not being followed by calling the `findPeople` method in `useEffect`, as shown in the following code.

mern-social/client/user/FindPeople.js:

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

  findPeople({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, signal).then((data) => {
    if (data && data.error) {
      console.log(data.error)
    } else {
      setValues({...values, users:data})
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

The fetched list of users will be iterated over and rendered in a Material-UI `List` component, with each list item containing the user's avatar, name, a link to the profile page, and a `Follow` button, as shown in the following code.

mern-social/client/user/FindPeople.js:

```
<List>
  {values.users.map((item, i) => {
    return <span key={i}>
      <ListItemIcon>
        <ListItemIconAvatar className={classes.avatar}>
          <Avatar src={'/api/users/photo/' + item._id}/>
        </ListItemIconAvatar>
```

```

        <ListItemText primary={item.name}/>
        <ListItemSecondaryAction className={classes.follow}>
            <Link to={"/user/" + item._id}>
                <IconButton variant="contained" color="secondary"
                    className={classes.viewButton}>
                    <ViewIcon/>
                </IconButton>
            </Link>
            <Button aria-label="Follow" variant="contained"
                color="primary"
                onClick={()=> {clickFollow(item, i)}}>
                Follow
            </Button>
        </ListItemSecondaryAction>
    </ListItem>
</span>
)
}
</List>

```

Clicking the `Follow` button will make a call to the follow API and update the list of users to follow by splicing out the newly followed user. The `clickFollow` method implements this behavior as follows.

mern-social/client/user/FindPeople.js:

```

const clickFollow = (user, index) => {
    follow({
        userId: jwt.user._id
    }, {
        t: jwt.token
    }, user._id).then((data) => {
        if (data.error) {
            console.log(data.error)
        } else {
            let toFollow = values.users
            toFollow.splice(index, 1)
            setValues({...values, users: toFollow, open: true,
                followMessage: `Following ${user.name}!`})
        }
    })
}

```

We will also add a Material-UI `Snackbar` component that will open temporarily when the user is successfully followed in order to tell the user that they started following this new user. `Snackbar` will be added to the view code as follows.

mern-social/client/user/FindPeople.js:

```
<Snackbar
  anchorOrigin={{
    vertical: 'bottom',
    horizontal: 'right',
  }}
  open={values.open}
  onClose={handleRequestClose}
  autoHideDuration={6000}
  message={<span className={classes.snack}>{values.followMessage}</span>}
/>
```

As shown in the following screenshot, `Snackbar` will display the `message` containing the followed user's name at the bottom-right corner of the page, and then auto-hide it after the set duration:



Following Dominic Kotsopolis!

MERN Social users can now follow each other, view lists of followings and followers for each user, and also see a list of people they can follow. The main purpose of following another user in MERN Social is to see and interact with their shared posts. In the next section, we will look at the implementation of the post feature.

Posting on MERN Social

The post feature in MERN Social will allow users to share content on the MERN Social application platform and also interact with each other over the content by commenting on or liking a post, as shown in the following screenshot:



Daniel Nakamura

Sun Dec 24 2017

"For the things we have to learn before we can do them, we learn by doing them." - Aristotle



4



2



Write something ...



Cecil McQueen

No better way!

Sun Dec 24 2017 |



Thomas Brogan

Practice, practice, practice

Sun Dec 24 2017 |

For this feature, we will implement a complete full-stack slice containing the post backend and frontend. The post backend will be comprised of a new Mongoose model for structuring the post data to be stored in the database, while the post CRUD API endpoints will allow the frontend to interact with the Post collection in the database. The post frontend will consist of post-related React components that will allow users to view posts, add new posts, interact with posts, and delete their own posts. In the following sections, we will define the data structure for posts in the Post schema, and then learn how to incrementally add the post backend APIs and frontend components according to the specific post-related feature we are implementing.

Mongoose schema model for Post

To define the structure for storing details about each post and to store each post as a document in a collection in MongoDB, we will define the Mongoose schema for a post in `server/models/post.model.js`. The Post schema will store a post's text content, a photo, a reference to the user who posted, time of creation, likes on the post from users, and comments on the post by users. The schema will store these details in the following fields, each defined as shown with the corresponding code.

- **Post text:** `text` will be a required field that needs to be provided by the user on new post creation from the view:

```
text: {  
  type: String,  
  required: 'Text is required'  
}
```

- **Post photo:** `photo` will be uploaded from the user's local files during post creation and stored in MongoDB, similar to the user profile photo upload feature. The photo will be optional for each post:

```
photo: {  
  data: Buffer,  
  contentType: String  
}
```

- **Post by:** Creating a post will require a user to be signed-in first so that we can store a reference to the user who is posting in the `postedBy` field:

```
|  postedBy: {type: mongoose.Schema.ObjectId, ref: 'User'}
```

- **Created time:** The `created` time will be generated automatically at the time of post creation in the database:

```
|     created: { type: Date, default: Date.now }
```

- **Likes:** References to the users who liked a specific post will be stored in a `likes` array:

```
|     likes: [{type: mongoose.Schema.ObjectId, ref: 'User'}]
```

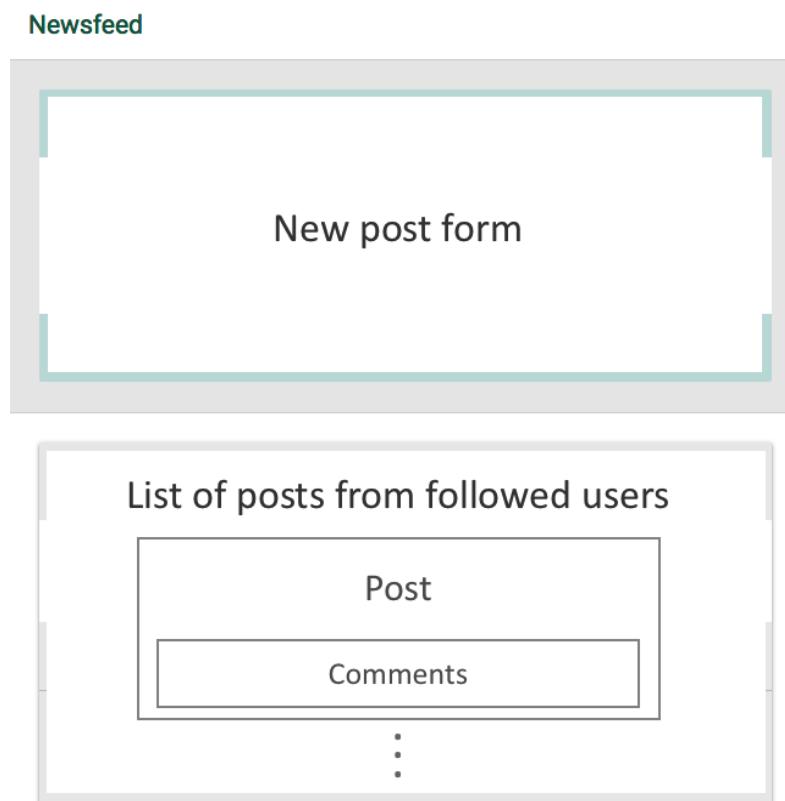
- **Comments:** Each comment on a post will contain text content, the time of creation, and a reference to the user who posted the comment. Each post will have an array of `comments`:

```
|     comments: [{  
|         text: String,  
|         created: { type: Date, default: Date.now },  
|         postedBy: { type: mongoose.Schema.ObjectId, ref: 'User' }  
|     }]
```

This schema definition will enable us to implement all the post-related features in MERN Social. Next, we will start with a discussion of the Newsfeed feature to learn how to compose frontend React components.

The Newsfeed component

On MERN Social, each user will see posts that have been shared by people they follow, along with posts that they themselves share, all aggregated in a Newsfeed view. Before delving further into the implementations of the post-related features in MERN Social, we will look at the composition of this Newsfeed view to showcase a basic example of how to design nested UI components that share state. The `Newsfeed` component will contain two main child components – a new post form and a list of posts from followed users, as shown in the following screenshot:



The basic structure of the `Newsfeed` component will be as follows, with the `NewPost` component and the `PostList` component inside it.

mern-social/client/post/Newsfeed.js:

```
| <Card>
|   <Typography type="title"> Newsfeed </Typography>
|   <Divider/>
|   <NewPost addUpdate={addPost}/>
|   <Divider/>
|   <PostList removeUpdate={removePost} posts={posts}/>
| </Card>
```

As the parent component, `Newsfeed` will control the state of the posts' data that's rendered in the child components. It will provide a way to update the state of posts across the components when the post data is modified within the child components, such as the addition of a new post in the `NewPost` component or the removal of a post from the `PostList` component.

Here specifically, in the `Newsfeed` component we initially make a call to the server to fetch a list of posts from people that the currently signed-in user follows. Then we set this list of posts to the state to be rendered in the `PostList` component.. The `Newsfeed` component provides the `addPost` and `removePost` functions to `NewPost` and `PostList`, which will be used when a new post is created or an existing post is deleted to update the list of posts in the Newsfeed's state and ultimately reflect it in the `PostList`.

The `addPost` function defined in the `Newsfeed` component will take the new post that was created in the `NewPost` component and add it to the posts in the state. The `addPost` function will look as follows.

mern-social/client/post/Newsfeed.js:

```
| const addPost = (post) => {
|   const updatedPosts = [...posts]
|   updatedPosts.unshift(post)
|   setPosts(updatedPosts)
| }
```

The `removePost` function defined in the `Newsfeed` component will take the deleted post from the `Post` component in `PostList` and remove it from the posts in the state. The `removePost` function will look as follows.

mern-social/client/post/Newsfeed.js:

```
const removePost = (post) => {
  const updatedPosts = [...posts]
  const index = updatedPosts.indexOf(post)
  updatedPosts.splice(index, 1)
  setPosts(updatedPosts)
}
```

As the posts are updated in the Newsfeed's state this way, the `PostList` will render the changed list of posts to the viewer. This mechanism of relaying state updates from parent to child components and back will be applied across other features, such as comment updates in a post and when a `PostList` is rendered for an individual user in the `Profile` component.

To begin the complete implementation of the `Newsfeed`, we need to be able to fetch a list of posts from the server and display it in the `PostList`. In the next section, we will make this `PostList` component for the frontend and add PostList API endpoints to the backend.

Listing posts

In MERN Social, we list posts in the `Newsfeed` and in the profile of each user. We will create a generic `PostList` component that will render any list of posts provided to it, which we can use in both the `Newsfeed` and the `Profile` components. The `PostList` component is defined as follows.

`mern-social/client/post/PostList.js`:

```
export default function PostList (props) {
  return (
    <div style={{marginTop: '24px'}}>
      {props.posts.map((item, i) => {
        return <Post post={item} key={i}
          onRemove={props.removeUpdate}/>
      })
    }
  )
}
PostList.propTypes = {
  posts: PropTypes.array.isRequired,
  removeUpdate: PropTypes.func.isRequired
}
```

The `PostList` component will iterate through the list of posts passed to it as `props` from the `Newsfeed` or the `Profile`, and pass the data of each post to a `Post` component that will render details of the post. `PostList` will also pass the `removeUpdate` function that was sent as a prop from the parent component to the `Post` component so that the state can be updated when a single post is deleted. Next, we will complete the post lists in the `Newsfeed` view after fetching the relevant posts from the backend.

Listing posts in Newsfeed

We will set up an API on the server that queries the Post collection and returns a list of posts from the people a specified user is following. Then, to populate the Newsfeed view, these posts will be retrieved in the frontend by calling this API and they will be displayed in the `PostList` in `Newsfeed`.

Newsfeed API for posts

To implement the Newsfeed-specific API, we need to add the route endpoint that will receive the request for Newsfeed posts and respond accordingly to the requesting client-side.

On the backend, we need to define the route path that will receive the request for retrieving Newsfeed posts for a specific user, as shown in the following code.

```
server/routes/post.routes.js
```

```
| router.route('/api/posts/feed/:userId')
  .get(authCtrl.requireSignin, postCtrl.listNewsFeed)
```

We are using the `:userId` parameter in this route to specify the currently signed-in user. We will utilize the `userByID` controller method in `user.controller` to fetch the user details, as we did previously, and append these to the request object that is accessed in the `listNewsFeed` post controller method. Add the following to `mern-social/server/routes/post.routes.js`:

```
| router.param('userId', userCtrl.userByID)
```

The `post.routes.js` file will be very similar to the `user.routes.js` file. To load these new routes in the Express app, we need to mount the post routes in `express.js`, just like we did for the auth and user routes. The post-related routes are mounted as follows.

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

```
| app.use('/', postRoutes)
```

The `listNewsFeed` controller method in `post.controller.js` will query the Post collection in the database to get the matching posts. The `listNewsFeed` controller method is defined as follows.

mern-social/server/controllers/post.controller.js:

```
const listNewsFeed = async (req, res) => {
  let following = req.profile.following
  following.push(req.profile._id)
  try {
    let posts = await Post.find({postedBy: { $in : req.profile.following
}})
      .populate('comments.postedBy', '_id name')
      .populate('postedBy', '_id name')
      .sort('-created')
      .exec()
    res.json(posts)
  } catch(err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

In the query to the Post collection, we find all the posts that have `postedBy` user references that match the current user's followings and the current user. The posts that are returned will be sorted by the `created` timestamp, with the most recent post listed first. Each post will also contain the `id` and `name` of the user who created the post and of the users who left comments on the post. Next, we will fetch this API in the frontend `Newsfeed` component and render the list details.

Fetching Newsfeed posts in the view

We will use the Newsfeed API in the frontend to fetch the related posts and display these posts in the Newsfeed view. First, we will add a fetch method to make a request to the API, as shown in the following code.

client/post/api-post.js:

```
const listNewsFeed = async (params, credentials, signal) => {
  try {
    let response = await fetch('/api/posts/feed/' + 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)
  }
}
```

This is the fetch method that will load the posts that are rendered in `PostList`, which is added as a child component to the `Newsfeed` component. So, this fetch needs to be called in the `useEffect` hook in the `Newsfeed` component, as shown in the following code.

mern-social/client/post/Newsfeed.js:

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

  listNewsFeed({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, signal).then((data) => {
```

```
    if (data.error) {
      console.log(data.error)
    } else {
      setPosts(data)
    }
  })
return function cleanup() {
  abortController.abort()
}

}, []))
```

This will retrieve the list of posts from the backend and set it to the state of the `Newsfeed` component to initially load the posts that are rendered in the `PostList` component, as shown in the following screenshot:

Newsfeed



Jamie Woolcraft

Share your thoughts ...



POST



Jamie Woolcraft

Sat Dec 23 2017



Woah! That was something else ...

 1  0

 Write something ...



Violet Bernstein

Fri Dec 22 2017

"Injustice anywhere is a threat to justice everywhere." - Martin Luther King, Jr.

 4  0

 Write something ...

The implementation of how the individual post details are rendered in this list will be discussed later in this chapter. In the next section, we will render this same `PostList` for the `Profile` component and render the posts that are shared by a specific user.

Listing posts by user in Profile

The implementation of getting a list of posts created by a specific user and showing it in `Profile` will be similar to what we discussed in the previous section regarding listing posts in the Newsfeed. First, we will set up an API on the server that queries the Post collection and returns posts from a specific user to the `Profile` view.

API for posts by a user

To retrieve posts that have been shared by a specific user, we need to add a route endpoint that will receive the request for these posts and respond accordingly to the requesting client-side.

On the backend, we will define another post-related route that will receive a query to return posts by a specific user, as follows.

mern-social/server/routes/post.routes.js:

```
| router.route('/api/posts/by/:userId')
|   .get(authCtrl.requireSignin, postCtrl.listByUser)
```

The `listByUser` controller method in `post.controller.js` will query the Post collection to find posts that have a matching reference in the `postedBy` field to the user specified in the `userId` param in the route. The `listByUser` controller method will look as follows.

mern-social/server/controllers/post.controller.js:

```
| const listByUser = async (req, res) => {
|   try {
|     let posts = await Post.find({postedBy: req.profile._id})
|       .populate('comments.postedBy', '_id name')
|       .populate('postedBy', '_id name')
|       .sort('-created')
|       .exec()
| 
|     res.json(posts)
|   } catch(err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

This query will return the list of posts that were created by a specific user. We need to call this API from the frontend, which we will do in the next section.

Fetching user posts in the view

We will use the list-posts-by-user API in the frontend to fetch the related posts and display these posts in the profile view. To use this API, we will add a fetch method to the frontend, as follows.

mern-social/client/post/api-post.js:

```
const listByUser = async (req, res) => {
  try {
    let posts = await Post.find({postedBy: req.profile._id})
      .populate('comments.postedBy', '_id name')
      .populate('postedBy', '_id name')
      .sort('-created')
      .exec()

    res.json(posts)
  } catch(err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This `fetch` method will load the required posts for `PostList`, which is added to the `Profile` view. We will update the `Profile` component so that it defines a `loadPosts` method that calls the `listByUser` `fetch` method. The `loadPosts` method will look as follows.

mern-social/client/user/Profile.js:

```
const loadPosts = (user) => {
  listByUser({
    userId: user
  }, {
    t: jwt.token
  }).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      setPosts(data)
    }
  })
}
```

In the `Profile` component, the `loadPosts` method will be called with the user ID of the user whose profile is being loaded, after the user details have been fetched from the server in the `useEffect()` hook function. The posts that are loaded for the specific user are set to the state and rendered in the `PostList` component that's added to the `Profile` component. The `Profile` component also provides a `removePost` function, similar to the `Newsfeed` component, as a prop to the `PostList` component so that the list of posts can be updated if a post is removed. The resulting `PostList` in the `Profile` component will render similar to what can be seen in the following screenshot:

Profile



Katherine Hemstridge
kitty@hems.info

[UNFOLLOW](#)

Goody two shoes

Joined: Thu Nov 30 2017

[POSTS](#)

[FOLLOWING](#)

[FOLLOWERS](#)



Katherine Hemstridge

Fri Dec 08 2017

Déjà vu!



2



0



Write something ...



Katherine Hemstridge

Thu Nov 30 2017

Today was an exceptionally good day!



4



0



Write something ...

The features that list posts that have been shared on MERN Social are now complete. But before these can be tested out, we need to implement the feature that will allow users to create new posts. We will do this in the next section.

Creating a new post

The create new post feature will allow a signed-in user to post a message and optionally add an image to the post by uploading it from their local files. To implement this feature, in the following sections, we will add a create post API endpoint to the backend that allows uploading an image file, as well as add a `NewPost` component to the frontend that will utilize this endpoint to let users create new posts.

Creating the post API

On the server, we will define an API to create the post in the database, starting by declaring a route to accept a POST request at `/api/posts/new/:userId` in `mern-social/server/routes/post.routes.js`:

```
| router.route('/api/posts/new/:userId')
|   .post(authCtrl.requireSignin, postCtrl.create)
```

The `create` method in `post.controller.js` will use the `formidable` module to access the fields and the image file, if any, as we did for the user profile photo update. The `create` controller method will look as follows.

`mern-social/server/controllers/post.controller.js`:

```
const create = (req, res, next) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        error: "Image could not be uploaded"
      })
    }
    let post = new Post(fields)
    post.postedBy= req.profile
    if(files.photo){
      post.photo.data = fs.readFileSync(files.photo.path)
      post.photo.contentType = files.photo.type
    }
    try {
      let result = await post.save()
      res.json(result)
    } catch (err) {
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}
```

Similar to the profile photo upload, the photo that's uploaded with a new post will be stored in the `Post` document in binary format. We

need to add a route to retrieve and return this photo to the frontend, which we will do next.

Retrieving a post's photo

To retrieve the uploaded photo, we will also set up a `photo` route endpoint that, on request, will return the photo associated with a specific post. The photo URL route will be defined with the other post-related routes, as follows.

mern-social/server/routes/post.routes.js:

```
| router.route('/api/posts/photo/:postId').get(postCtrl.photo)
```

The `photo` controller will return the `photo` data stored in MongoDB as an image file. This is defined as follows.

mern-social/server/controllers/post.controller.js:

```
| const photo = (req, res, next) => {
|   res.set("Content-Type", req.post.photo.contentType)
|   return res.send(req.post.photo.data)
| }
```

Since the photo route uses the `:postId` parameter, we will set up a `postByID` controller method to fetch a specific post by its ID before returning it to the photo request. We will add the `param` call to `post.routes.js`, as shown in the following code.

mern-social/server/routes/post.routes.js:

```
|   router.param('postId', postCtrl.postByID)
```

`postByID` will be similar to the `userByID` method, and it will attach the post retrieved from the database to the request object so that it can be accessed by the `next` method. The `postByID` method is defined as follows.

mern-social/server/controllers/post.controller.js:

```
const postByID = async (req, res, next, id) => {
  try{
    let post = await Post.findById(id)
      .populate('postedBy', '_id name')
      .exec()
    if (!post)
      return res.status('400').json({
        error: "Post not found"
      })
    req.post = post
    next()
  }catch(err){
    return res.status('400').json({
      error: "Could not retrieve use post"
    })
  }
}
```

The attached post data in this implementation will also contain the ID and name of the `postedBy` user reference since we are invoking `populate()`. In the next section, we will add a fetch method to access this API endpoint in the frontend.

Fetching the create post API in the view

We will update `api-post.js` by adding a `create` method to make a `fetch` call to the create API. The `create` fetch method will look as follows.

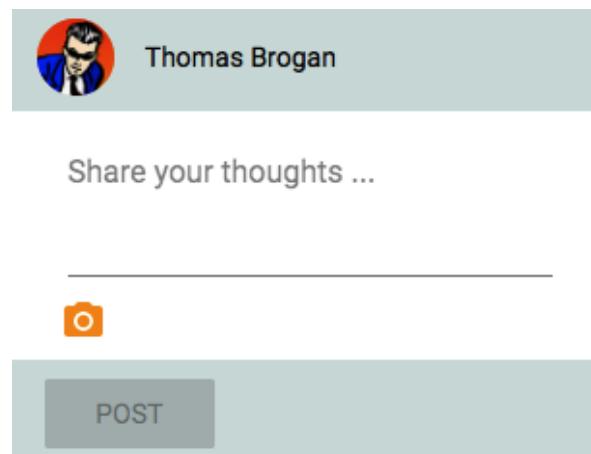
mern-social/client/post/api-post.js:

```
const create = async (params, credentials, post) => {
  try {
    let response = await fetch('/api/posts/new/' + params.userId, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      },
      body: post
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

This method, like the user `edit` fetch method, will send a multipart form submission using a `FormData` object that will contain the text field and the image file. Finally, we are ready to integrate this create new post feature in the backend with a frontend component that will allow users to compose a post and submit it to the backend.

Making the NewPost component

The `NewPost` component that we added in the `Newsfeed` component will allow users to compose a new post containing a text message and, optionally, an image, as shown in the following screenshot:



The `NewPost` component will be a standard form with a Material-UI `TextField` and a file upload button, as implemented in `EditProfile`, that takes the values and sets them in a `FormData` object to be passed in the call to the `create` fetch method on post submission. Post submission will invoke the following `clickPost` method.

mern-social/client/post/NewPost.js:

```
const clickPost = () => {
  let postData = new FormData()
  postData.append('text', values.text)
  postData.append('photo', values.photo)
  create({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, postData).then((data) => {
    if (data.error) {
      setValues({...values, error: data.error})
    } else {
```

```
    setValues({...values, text:'', photo: ''})
    props.addUpdate(data)
  })
}
```

The `NewPost` component is added as a child component in the `Newsfeed` and given the `addUpdate` method as a prop, as we discussed earlier. On successful post creation, the form view is emptied and `addUpdate` is executed so that the post list in the `Newsfeed` is updated with the new post. In the next section, we will add the `Post` component in order to display each post and its details.

The Post component

Post details in each post will be rendered in the `Post` component, which will receive the post data as props from the `PostList` component, as well as the `onRemove` prop, which needs to be applied if a post is deleted. In the following sections, we will look at the different parts of the Post interface and how to implement each.

Layout

The `Post` component layout will have a header showing details of the poster, the content of the post, an actions bar with a count of likes and comments, and a comments section, as shown in the following screenshot:

The screenshot displays a user interface for a social media post. At the top left is a circular profile picture of a person with purple hair. To its right, the name "Violet Bernstein" is written in purple, followed by the date "Fri Dec 22 2017" in blue. On the far right of the header is a small trash can icon. Below the header, a quote by Martin Luther King, Jr. is shown in a grey box: "Injustice anywhere is a threat to justice everywhere." - Martin Luther King, Jr. In the main content area, there is a large orange heart icon with the number "3" next to it, and an orange speech bubble icon with the number "1". Below this is a text input field with a placeholder "Write something ...". A smaller box contains another user's profile picture, the name "Cecil McQueen" in purple, the word "Truth!" in blue, and the date "Fri Dec 22 2017 |".

Next, we will look into the implementation details of the header, content, actions, and comments sections of this Post component.

Header

The header will contain information such as the name, avatar, and link to the profile of the user who posted, as well as the date the post was created. The code to display these details in the header section will be as follows.

mern-social/client/post/Post.js:

```
<CardHeader
  avatar={ 
    <Avatar src={'/api/users/photo/' + props.post.postedBy._id} />
  }
  action={ props.post.postedBy._id === auth.isAuthenticated().user._id
&&
    <IconButton onClick={deletePost}>
      <DeleteIcon />
    </IconButton>
  }
  title={<Link to={"/user/" + props.post.postedBy._id}>
{props.post.postedBy.name}</Link>}
  subheader={ (new Date(props.post.created)).toDateString() }
  className={classes.cardHeader}
/>
```

The header will also conditionally show a `delete` button if the signed-in user is viewing their own post. This header section will be above the main content section, which is discussed next.

Content

The content section will show the text of the post and the image if the post contains a photo. The code to display these details in the content section will be as follows.

mern-social/client/post/Post.js:

```
<CardContent className={classes.cardContent}>
  <Typography component="p" className={classes.text}>
    {props.post.text}
  </Typography>
  {props.post.photo &&
    (<div className={classes.photo}>
      <img className={classes.media}
           src={'/api/posts/photo/' + props.post._id}/>
    </div>)
  }
</CardContent>
```

The image is loaded by adding the photo API to the `src` attribute in the `img` tag if the given post contains a photo. Followed by this content section is the actions section.

Actions

The actions section will contain an interactive "like" option with a display of the total number of likes on the post and a comment icon with the total number of comments on the post. The code to display these actions will be as follows.

mern-social/client/post/Post.js:

```
<CardActions>
  { values.like
    ? <IconButton onClick={clickLike} className={classes.button}
      aria-label="Like" color="secondary">
      <FavoriteIcon />
    </IconButton>
    : <IconButton onClick={clickLike} className={classes.button}
      aria-label="Unlike" color="secondary">
      <FavoriteBorderIcon />
    </IconButton> } <span>{values.likes}</span>
  <IconButton className={classes.button}
    aria-label="Comment" color="secondary">
    <CommentIcon/>
  </IconButton> <span>{values.comments.length}</span>
</CardActions>
```

We will discuss the implementation of the "like" button later in this chapter. The details of the likes for each post are retrieved within the `post` object that's received in the props.

In the Post component, the final section will display the comments that have been left on the given post. We'll discuss this next.

Comments

The comments section will contain all the comment-related elements in the `Comments` component and will get `props` such as the `postId` and the `comments` data, along with a `state` updating method that can be called when a comment is added or deleted in the `Comments` component.

The comments section will be rendered in the view with the following code.

`mern-social/client/post/Post.js`:

```
| <Comments postId={props.post._id}
|   comments={values.comments}
|   updateComments={updateComments}/>
```

The implementation of this `Comments` component will be detailed later in this chapter. These four sections make up the individual post view that we implemented in the `Post` component, which is rendered in `PostList` component. Each post's header also shows a `delete` button to the creator of the post. We will implement this remove post functionality next.

Deleting a post

The `delete` button is only visible if the signed-in user and `postedBy` user are the same for the specific post being rendered. For the post to be deleted from the database, we will have to set up a delete post API in the backend which will also have a fetch method in the frontend that will be applied when `delete` is clicked. The route for the delete post API endpoint will be as follows.

`mern-social/server/routes/post.routes.js`:

```
router.route('/api/posts/:postId')
    .delete(authCtrl.requireSignin,
            postCtrl.isPoster,
            postCtrl.remove)
```

The delete route will check for authorization before calling `remove` on the post by ensuring the authenticated user and `postedBy` user are the same users. The `isPoster` method, which is implemented in the following code, checks whether the signed-in user is the original creator of the post before executing the `next` method.

`mern-social/server/controllers/post.controller.js`:

```
const isPoster = (req, res, next) => {
  let isPoster = req.post && req.auth &&
  req.post.postedBy._id == req.auth._id
  if(!isPoster){
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}
```

The rest of the implementation for the delete API with a `remove` controller method and fetch method for the frontend are the same as for the other API implementations. The important difference here, in the delete post feature, is the call to the `onRemove` update method in the `Post` component when delete succeeds. The `onRemove` method is sent as

a prop from either `Newsfeed` or `Profile` to update the list of posts in the state when the delete is successful.

The following `deletePost` method, which is defined in the `Post` component, is called when the `delete` button is clicked on a post.

mern-social/client/post/Post.js:

```
const deletePost = () => {
  remove({
    postId: props.post._id
  }, {
    t: jwt.token
  }).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      props.onRemove(props.post)
    }
  })
}
```

This method makes a fetch call to the delete post API and on success, updates the list of posts in the state by executing the `onRemove` method, which is received as a prop from the parent component.

This wraps up the implementation of the Post CRUD features in the backend and the frontend. However, we have not completed the features that will allow MERN Social users to interact with these posts. In the next section, we will add the ability to like posts and comment on posts.

Interacting with Posts

A core feature of any social media platform is the ability for users to interact with shared content. For the posts that are created in the MERN Social application, we will add the options to like and leave comments on individual posts.

To complete the implementation of this feature, first, we will have to modify the backend so that we can add API endpoints that update an existing post with details of who liked the post and details of comments left on the post.

Then, in the frontend, we will have to modify the UI so that users can like and leave a comment on a post.

Likes

The like option in the `Post` component's action bar section will allow the user to like or unlike a post, and also show the total number of likes for the post. To record a "like", we will have to set up like and unlike APIs that can be called in the view when the user interacts with the action bar that's rendered in each post.

The Like API

The like API will be a PUT request that will update the `likes` array in the `Post` document. The request will be received at the `api/posts/like` route, which is defined as follows.

mern-social/server/routes/post.routes.js:

```
| router.route('/api/posts/like')
|   .put(authCtrl.requireSignin, postCtrl.like)
```

In the `like` controller method, the post ID that's received in the request body will be used to find the specific `Post` document and update it by pushing the current user's ID to the `likes` array, as shown in the following code.

mern-social/server/controllers/post.controller.js:

```
| const like = async (req, res) => {
|   try {
|     let result = await Post.findByIdAndUpdate(req.body.postId,
|                                           {$push: {likes: req.body.userId}},
|                                           {new: true})
|     res.json(result)
|   } catch(err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

To use this API, a fetch method called `like` will be added to `api-post.js`, which will be used when the user clicks the `like` button. The `like` fetch is defined as follows.

mern-social/client/post/api-post.js:

```
| const like = async (params, credentials, postId) => {
|   try {
|     let response = await fetch('/api/posts/like/', {
|       method: 'PUT',
|       headers: {
```

```
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + credentials.token
    },
    body: JSON.stringify({userId:params.userId, postId: postId})
})
return await response.json()
} catch(err) {
    console.log(err)
}
}
```

Similarly, in the next section, we will also implement an unlike API endpoint so that a user can unlike a previously liked post.

The Unlike API

The `unlike` API will be implemented similar to the like API, with its own route. This will be declared as follows.

mern-social/server/routes/post.routes.js:

```
|   router.route('/api/posts/unlike')
|     .put(authCtrl.requireSignin, postCtrl.unlike)
```

The `unlike` method in the controller will find the post by its ID and update the `likes` array by removing the current user's ID using `$pull` instead of `$push`. The `unlike` controller method will look as follows.

mern-social/server/controllers/post.controller.js:

```
| const unlike = async (req, res) => {
|   try {
|     let result = await Post.findByIdAndUpdate(req.body.postId,
|                                           {$pull: {likes: req.body.userId}},
|                                           {new: true})
|     res.json(result)
|   } catch(err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

The unlike API will also have a corresponding fetch method, similar to the `like` method in `api-post.js`.

These APIs will be called when the user interacts with the like button in the view. But whether the like button should allow a like or an unlike action needs to be determined first. We will take a look at this in the next section.

Checking if a post has been liked and counting likes

When the `Post` component is rendered, we need to check if the currently signed-in user has liked the post or not so that the appropriate `like` option can be shown. The following `checkLike` method checks whether the currently signed-in user is referenced in the post's `likes` array or not.

mern-social/client/post/Post.js:

```
const checkLike = (likes) => {
  let match = likes.indexOf(jwt.user._id) !== -1
  return match
}
```

This `checkLike` function can be called while setting the initial value of the `like` state variable, which keeps track of whether the current user liked the given post or not. The following screenshot shows how the like button renders when a post has not been liked versus when it has been liked by the current user:



The `like` value that's set in the state using the `checkLike` method can be used to render a heart outline button or a full heart button. A heart outline button will render if the user has not liked the post; clicking it will make a call to the `like` API, show the full heart button, and increment the `likes` count. The full heart button will indicate the current user has already liked this post; clicking this will call the `unlike` API, render the heart outline button, and decrement the `likes` count.

The `likes` count is also set initially when the `Post` component mounts and props are received by setting the `likes` value to the state with

`props.post.likes.length`, as shown in the following code.

mern-social/client/post/Post.js:

```
| const [values, setValues] = useState({  
|   like: checkLike(props.post.likes),  
|   likes: props.post.likes.length,  
|   comments: props.post.comments  
| })
```

The likes-related values are updated again when a "like" or "unlike" action takes place, and the updated post data is returned from the API call. Next, we will look at how to handle these clicks on the like button.

Handling like clicks

To handle clicks on the `like` and `unlike` buttons, we will set up a `clickLike` method that will call the appropriate fetch method based on whether it is a "like" or "unlike" action, and then update the state of the `like` and `likes` count for the post. This `clickLike` method will be defined as follows.

mern-social/client/post/Post.js:

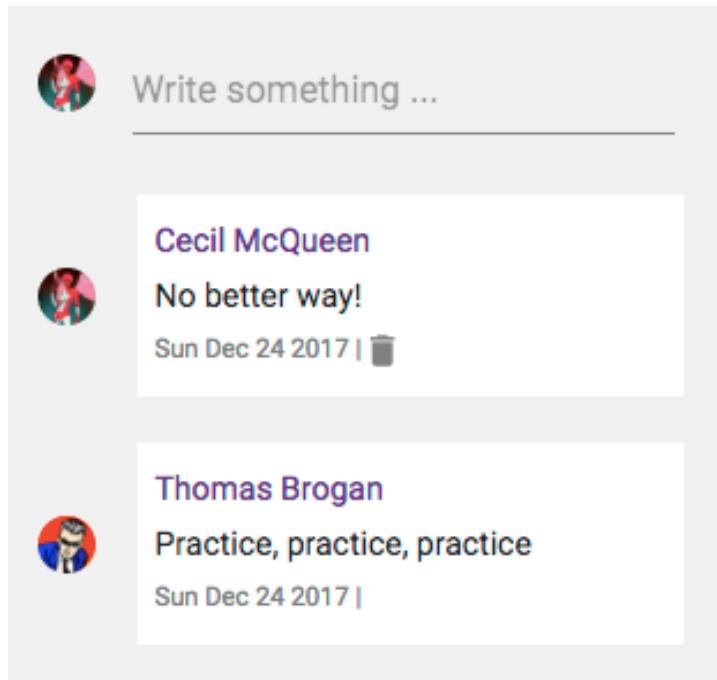
```
const clickLike = () => {
  let callApi = values.like ? unlike : like
  const jwt = auth.isAuthenticated()
  callApi({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, props.post._id).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      setValues({...values, like: !values.like,
                likes: data.likes.length})
    }
  })
}
```

Which of the like or unlike API endpoints will be called on a click depends on the value of the `like` variable in the state. Once the chosen API endpoint is called successfully, the values are updated in the state so that they can be reflected in the view.

This completes the likes feature implementation, complete with backend APIs integrated with the frontend to enable liking and unliking a given post. Next, we will add the comments feature to complete the social media application features we had set out for MERN Social.

Comments

The comments section in each post will allow signed-in users to add comments, see the list of comments, and delete their own comments. Any changes to the comment list, such as a new addition or removal, will update the comments, as well as the comment count in the action bar section of the `Post` component. The resulting comments section can be seen in the following screenshot:



To implement a functional comments section, we will update the backend with the corresponding comment and uncomment API endpoints, and also create this `Comments` component so that it integrates with the backend update.

Adding a comment

When a user adds a comment, the `Post` document will be updated in the database with the new comment. First, we need to implement an API that receives the comment details from the client-side and updates the `Post` document. Then, we need to create the UI in the frontend, which allows us to compose a new comment and submit it to the backend API.

The Comment API

To implement the add comment API, we will set up a `PUT` route as follows to update the post.

mern-social/server/routes/post.routes.js:

```
| router.route('/api/posts/comment')
|   .put(authCtrl.requireSignin, postCtrl.comment)
```

The `comment` controller method, which is defined in the following code, will find the relevant post to be updated by its ID and push the `comment` object that's received in the request body to the `comments` array of the post.

mern-social/server/controllers/post.controller.js:

```
const comment = async (req, res) => {
  let comment = req.body.comment
  comment.postedBy = req.body.userId
  try {
    let result = await Post.findByIdAndUpdate(req.body.postId,
      {$push: {comments: comment}},
      {new: true})
      .populate('comments.postedBy', '_id name')
      .populate('postedBy', '_id name')
      .exec()
    res.json(result)
  } catch(err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

In the response, the updated post object will be sent back with details of the `postedBy` users populated in the post and in the comments.

To use this API in the view, we will set up a fetch method in `api-post.js`, which takes the current user's ID, the post ID, and the `comment`

object from the view, and sends it with the add comment request. The `comment` fetch method will look as follows.

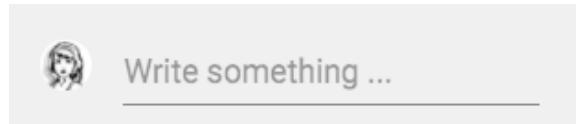
mern-social/client/post/api-post.js:

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

We can use this fetch method in the UI when the user submits a new comment, as discussed in the next section.

Writing something in the view

The *add comment* section in the `Comments` component will allow the signed-in user to type in the comment text:



This will contain an avatar showing the user's photo and a text field, which will add the comment when the user presses the *Enter* key. This add comment section will be rendered in the view with the following code.

`mern-social/client/post/Comments.js`:

```
<CardHeader
  avatar={
    <Avatar className={classes.smallAvatar}
      src= {'/api/users/photo/' +
        +auth.isAuthenticated().user._id}/>
  }
  title={ <TextField
    onKeyDown={addComment}
    multiline
    value={text}
    onChange={handleChange}
    placeholder="Write something ..."
    className={classes.commentField}
    margin="normal"
  />
}
  className={classes.cardHeader}
/>
```

The text will be stored in the state when the value changes, and on the `onKeyDown` event, the `addComment` method will call the `comment` fetch method if the *Enter* key is pressed. The *Enter* key corresponds to `keyCode 13`, as shown in the following code.

`mern-social/client/post/Comments.js`:

```
const addComment = (event) => {
  if(event.keyCode == 13 && event.target.value) {
    event.preventDefault()
    comment({
      userId: jwt.user._id
    }, {
      t: jwt.token
    }, props.postId, {text: text}).then((data) => {
      if (data.error) {
        console.log(data.error)
      } else {
        setText('')
        props.updateComments(data.comments)
      }
    })
  }
}
```

The `Comments` component receives the `updateComments` method (as discussed in the previous section) as a prop from the `Post` component. This will be executed when the new comment is added in order to update the comments list and the comment count in the Post view. The part of `Comments` that lists all the comments for the post will be added in the next section.

Listing comments

The `Comments` component receives the list of comments for the specific post as props from the `Post` component. Then, it iterates over the individual comments to render the details of the commenter and the comment content. This view is implemented with the following code.

mern-social/client/post/Comments.js:

```
{ props.comments.map((item, i) => {
    return <CardHeader
        avatar={
            <Avatar className={classes.smallAvatar}
                src={'/api/users/photo/' + item.postedBy._id}/>
        }
        title={commentBody(item)}
        className={classes.cardHeader}
        key={i} />
    )
})}
```

`commentBody` renders the content, including the name of the commenter linked to their profile, the comment text, and the date of comment creation. `commentBody` is defined as follows.

mern-social/client/post/Comments.js:

```
const commentBody = item => {
    return (
        <p className={classes.commentText}>
            <Link to={"/user/" + item.postedBy._id}>
                {item.postedBy.name} </Link><br/>
            {item.text}
            <span className={classes.commentDate}>
                { (new Date(item.created)).toDateString() } |
                { auth.isAuthenticated().user._id === item.postedBy._id &&
                    <Icon onClick={deleteComment(item)}
                        className={classes.commentDelete}>delete</Icon> }
            </span>
        </p>
    )
}
```

`commentBody` will also render a delete option for the comment if the `postedBy` reference of the comment matches the currently signed-in user. We will look at the implementation of this comment deletion option in the next section.

Deleting a comment

Clicking the delete button in a comment will update the post in the database by removing the comment from the `comments` array in the corresponding post. The delete button can be seen underneath the comment shown in the following screenshot:



To implement this delete comment feature, we need to add an `uncomment` API to the backend and then use it in the frontend.

The Uncomment API

We will implement an `uncomment` API at the following PUT route.

`mern-social/server/routes/post.routes.js`:

```
| router.route('/api/posts/uncomment')
|   .put(authCtrl.requireSignin, postCtrl.uncomment)
```

The `uncomment` controller method will find the relevant post by ID and pull the comment with the deleted comment's ID from the `comments` array in the post, as implemented in the following code.

`mern-social/server/controllers/post.controller.js`:

```
const uncomment = async (req, res) => {
  let comment = req.body.comment
  try{
    let result = await Post.findByIdAndUpdate(req.body.postId,
                                           {$pull: {comments: {_id:
comment._id}}},
                                           {new: true})
    .populate('comments.postedBy', '_id name')
    .populate('postedBy', '_id name')
    .exec()
    res.json(result)
  } catch(err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The updated post will be returned in the response, similar to the comment API.

To use this API in the view, we will also set up a fetch method in `api-post.js`, similar to the `addComment` fetch method, that takes the current user's ID, the post ID, and the deleted `comment` object to send with the `uncomment` request. Next, we will learn how to use this fetch method when the delete button is clicked.

Removing a comment from the view

When a comment's delete button is clicked by the commenter, the `Comments` component will call the `deleteComment` method to fetch the `uncomment` API and update the comments, along with the comment count, when the comment is successfully removed from the server.

The `deleteComment` method is defined as follows.

mern-social/client/post/Comments.js:

```
const deleteComment = comment => event => {
  uncomment({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, props.postId, comment).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      props.updateComments(data.comments)
    }
  })
}
```

On successfully removing a comment from the backend, the `updateComments` method that's sent in the props from the `Post` component will be invoked. This will update the state of the `Post` component to update the view. This will be discussed in the next section.

Comment count update

The `updateComments` method, which will allow the `comments` and comment count to be updated when a comment is added or deleted, is defined in the `Post` component and passed as a prop to the `Comments` component. The `updateComments` method will look as follows:

mern-social/client/post/Post.js:

```
| const updateComments = (comments) => {
|   setValues({...values, comments: comments})
| }
```

This method takes the updated list of comments as a parameter and updates the state that holds the list of comments rendered in the view. The initial state of comments in the `Post` component is set when the `Post` component mounts and receives the post data as props. The comments that are set here are sent as props to the `Comments` component and used to render the comment count next to the likes action in the action bar of the Post layout, as follows.

mern-social/client/post/Post.js:

```
| <IconButton aria-label="Comment" color="secondary">
|   <CommentIcon/>
| </IconButton> <span>{values.comments.length}</span>
```

This relationship between the comment count in the `Post` component and the comments that are rendered and updated in the `Comments` component gives a simple demonstration of how changing data is shared among nested components in React to create dynamic and interactive user interfaces.

The MERN Social application now contains the set of features we defined earlier for the application. Users are able to update their profiles with a photo and description, follow each other on the application, and create posts with photos and text, as well as like

and comment on posts. The implementations shown here can be tuned and extended further to add more features in order to utilize the revealed mechanisms of working with the MERN stack.

Summary

The MERN Social application we developed in this chapter demonstrated how the MERN stack technologies can be used together to build out a fully-featured and functioning web application with social media features.

We began by updating the user feature in the skeleton application to allow anyone with an account on MERN Social to add a description about themselves, as well as upload a profile picture from their local files. In the implementation of uploading a profile picture, we explored how to upload multipart form data from the client, then receive it on the server to store the file data directly in the MongoDB database, and then be able to retrieve it back for viewing.

Next, we updated the user feature further to allow users to follow each other on the MERN Social platform. In the user model, we added the capability to maintain arrays of user references to represent lists of followers and followings for each user. Extending this capability, we incorporated follow and unfollow options in the view and displayed lists of followers, followings, and even lists of users not followed yet.

Then, we added the ability to allow users to post content and interact over the content by liking or commenting on the post. On the backend, we set up the Post model and corresponding APIs, which are capable of storing the post content that may or may not include an image, as well as maintaining records of likes and comments that are incurred on a post by any user.

Finally, while implementing the views for the posting, liking, and commenting features, we explored how to use component composition and share changing state values across the components to create complex and interactive views.

By completing this MERN Social application implementation, we learned how to extend and modify the base application code to grow it into a full-fledged web application according to our desired features. You can apply similar strategies to grow the skeleton application into any real-world application of your choosing.

In the next chapter, we will expand further on these abilities in the MERN stack and unlock new possibilities as we develop an online classroom application by extending the skeleton application.

Developing Web Applications with MERN

In this part, we develop two different web applications using the MERN skeleton application from the previous section, demonstrating how basic to complex features can be implemented and added on to growing MERN applications.

This section comprises the following chapters:

- [Chapter 6](#), *Building a Web-Based Classroom Application*
- [Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*
- [Chapter 8](#), *Extending the Marketplace for Orders and Payments*
- [Chapter 9](#), *Adding Real-Time Bidding Capabilities to the Marketplace*

Building a Web-Based Classroom Application

As the world is moving to the internet, so are our tools for learning and acquiring knowledge in different disciplines. Right now on the web, there is a plethora of online platforms that offer both educators and students options to teach and learn different topics remotely, without the necessity to be physically co-located in a classroom.

In this chapter, we will build a simple online classroom application, by extending the MERN stack skeleton application. This classroom application will support multiple user roles, the addition of course content and lessons, student enrollments, progress tracking, and course enrollment statistics. While building out this application, we will uncover more capabilities of this stack, such as how to implement role-based access to resources and actions, how to combine multiple schemas, and how to run different query operations in order to gather stats. By the end of this chapter, you will be familiar with the techniques that are needed to easily integrate new full-stack features in any MERN-based application.

We will build out the online classroom application by covering the following topics in this chapter:

- Introducing MERN Classroom
- Adding an educator role to users
- Adding courses to the classroom
- Updating courses with lessons
- Publishing courses
- Enrolling in courses
- Tracking progress and enrollment stats

Introducing MERN Classroom

MERN Classroom is a simple online classroom application, which allows educators to add courses that are made up of various lessons, while students can enroll on these courses. Additionally, the application will allow students to track their progress throughout the course, whereas instructors can monitor how many students have enrolled in/on a course, and how many have completed each course. The completed application, with all these features, will end up with a home page as shown in the following screenshot:

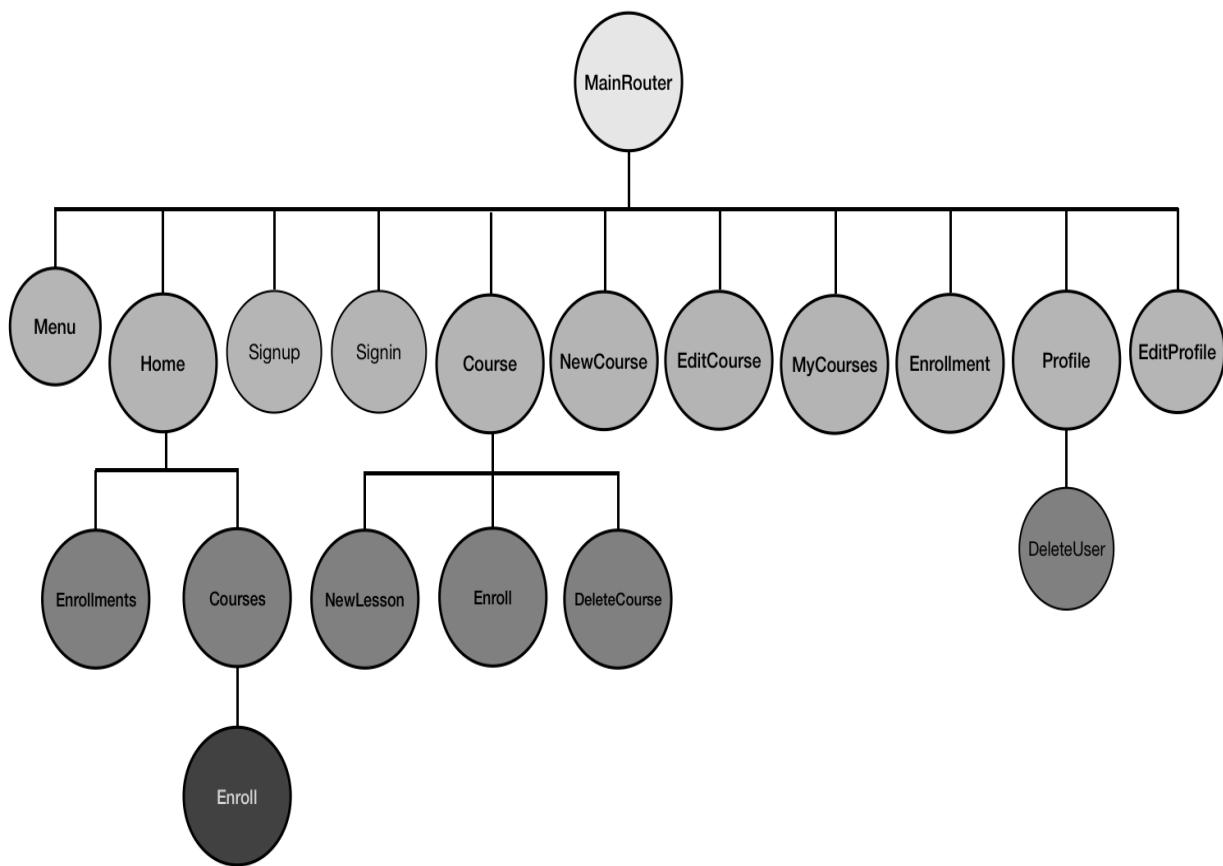
The screenshot shows the MERN Classroom application interface. At the top, there is a navigation bar with the title "MERN Classroom" and icons for "TEACH", "MY PROFILE", and "SIGN OUT". Below the navigation bar, a section titled "Courses you are enrolled in" displays four course cards: "Node.js Fundamentals", "React for Beginners", "Learn Git", and "JavaScript Pro". Each card includes a small icon, the course name, and a progress indicator. Below this, a section titled "All Courses" displays four course cards: "Probability & Statistics" (Mathematics), "Introduction to Machine Learning" (Computer Science), "Search Engine Optimization" (Digital Marketing), and "Basics of Microeconomics" (Economics). Each card features an image, the course name, a brief description, and an "ENROLL" button.



The code for the complete MERN Classroom application is available on GitHub in the repository at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter06/mern-classroom>. You can clone this code and run the application as you go through the code explanations for the rest of this chapter.

The views needed for the MERN Classroom application will be developed by extending and modifying the

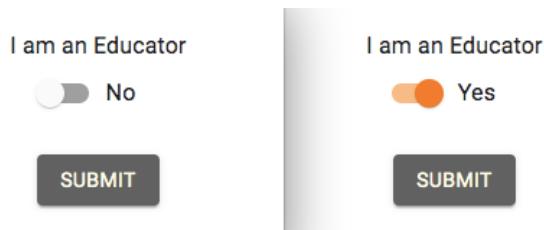
existing React components in the MERN skeleton application. The component tree in the following diagram lays out all the custom React components that make up the MERN Classroom frontend, and also exposes the composition structure that we will use to build out the views in the rest of the chapter:



We will add new React components that are related to courses, lessons, and enrollments; and we will also modify existing components such as the EditProfile, Menu, and Home components as we build out the different features of the MERN Classroom application in the rest of the chapter. Most of these features in the Classroom application will depend on the user's ability to become an educator. In the next section, we will begin implementing the MERN Classroom application by updating the user in order to give them the option to choose an educator role.

Updating the user with an educator role

Users who sign up to the MERN Classroom application will have the choice to become an educator on the platform by selecting this option in the `EditProfile` form component. This option in the form will look as follows—showing when the user isn't an educator, versus when they choose to be an educator:



When a user chooses to be an educator, in contrast to being a regular user, they will be allowed to create and manage their own courses. As pictured in the following screenshot, MERN Classroom will display a TEACH option in the navigation menu for educators only, that is, it won't be shown to regular users:



In the following sections, we will add this educator feature, by first updating the user model, then the `EditProfile` view, and finally by adding a `TEACH` link to the menu that will only be visible to educators.

Adding a role to the user model

The existing user model in the MERN skeleton application will need an educator value that will be set to `false` by default in order to represent regular users, but that can be set to `true` to represent the users who are also educators. To add this new field to the user schema, we will add the following code.

mern-classroom/server/models/user.model.js:

```
|   educator: {  
|     type: Boolean,  
|     default: false  
|   }
```

This `educator` value must be sent to the frontend, with the user details received once the user has successfully signed in, so that the view can be rendered accordingly to show information that is relevant to the educator. To make this change, we need to update the response that was sent back in the `signin` controller method as highlighted in the following code:

mern-classroom/server/controllers/auth.controller.js

```
| ...  
|  
|   token,  
|   user: {  
|     _id: user._id,  
|     name: user.name,  
|     email: user.email,  
|     educator: user.educator  
|   }  
| })  
| ...  
| }
```

By sending this `educator` field value back in the response, we can render the frontend views with role-specific authorization considerations.

But before getting to these conditionally rendered views, we first need to implement the option to select an educator role in the `EditProfile` view, as discussed in the next section.

Updating the EditProfile view

In order to become an educator in the MERN Classroom application, a signed-in user will need to update their profile. They will see a toggle in the EditProfile view, which will either activate or deactivate the educator feature. To implement this, first, we will update the `EditProfile` component in order to add a Material-UI `Switch` component in `FormControlLabel`, as shown in the following code.

mern-classroom/client/user/EditProfile.js:

```
<Typography variant="subtitle1" className={classes.subheading}>
  I am an Educator
</Typography>
<FormControlLabel
  control={
    <Switch classes={{
      checked: classes.checked,
      bar: classes.bar,
    }}>
      checked={values.educator}
      onChange={handleCheck}
    />
  }
  label={values.educator? 'Yes' : 'No'}
/>
```

Any changes to the switch will be set to the value of the `educator` variable in the state by calling the `handleCheck` method, as defined in the following code.

mern-classroom/client/user/EditProfile.js:

```
const handleCheck = (event, checked) => {
  setValues({...values, 'educator': checked})
```

The `handleCheck` method receives the `checked` Boolean value to indicate whether the switch has been selected or not, and this value is set to `educator`.

On form submit, the `educator` value is added to the details that were sent in the update to the server, as highlighted in the following code.

mern-classroom/client/user/EditProfile.js:

```
clickSubmit = () => {
  const jwt = auth.isAuthenticated()
  const user = {
    name: this.state.name || undefined,
    email: this.state.email || undefined,
    password: this.state.password || undefined,
    educator: values.educator || undefined
  }
  update({
    userId: this.match.params.userId
  }, {
    t: jwt.token
  }, user).then((data) => {
    if (data.error) {
      this.setState({error: data.error})
    } else {
      auth.updateUser(data, ()=> {
        setValues({...values, userId: data._id, redirectToProfile:
true})
      })
    }
  })
}
```

Once the EditProfile view has been successfully updated, the user details that are stored in `sessionStorage` for auth purposes should also be updated. The `auth.updateUser` method is called to do this `sessionStorage` update. It is defined with the other `auth-helper.js` methods, and the parameters that are passed are the updated user data and a callback function that updates the view. This `updateUser` method is defined as follows.

mern-classroom/client/auth/auth-helper.js:

```
updateUser(user, cb) {
  if(typeof window !== "undefined"){
    if(sessionStorage.getItem('jwt')){
      let auth = JSON.parse(sessionStorage.getItem('jwt'))
      auth.user = user
      sessionStorage.setItem('jwt', JSON.stringify(auth))
      cb()
    }
  }
}
```

Once the updated educator role is available in the frontend, we can use it to render the frontend accordingly. In the next section, we will see how to render the menu differently, based on whether an educator or a regular user is viewing the application.

Rendering an option to teach

In the frontend of the classroom application, we can render different options based on whether an educator is currently browsing the application. In this section, we will add the code to conditionally display a link to *TEACH* on the navigation bar, which will only be visible to the signed-in users who are also educators.

We will update the `Menu` component, as follows, within the previous code that only renders when a user is signed in.

mern-classroom/client/core/Menu.js:

```
{auth.isAuthenticated() && (<span>
  {auth.isAuthenticated().user.educator &&
    (<Link to="/teach/courses">
      <Button style={isPartActive(history, "/teach/")}>
        <Library/> Teach </Button>
      </Link>
    )
  ...
}
```

This link, which is only visible to educators, will take them to the educator dashboard view, where they can manage the courses that they are instructing.

This section has taught us how to update a user role to an educator role in the application, and we can now begin incorporating features that will allow an educator to add courses to the classroom.

Adding courses to the classroom

Educators in the MERN Classroom can create courses and add lessons to each course. In this section, we will walk through the implementations of the course-related features, such as adding new courses, listing courses by a specific instructor, and displaying the details of a single course. To store the course data and enable course management, we will start by implementing a Mongoose schema for courses, then backend APIs to create and list the courses, along with frontend views for both authorized educators and for regular users interacting with courses in the application.

Defining a Course model

The Course schema—defined in `server/models/course.model.js`—will have simple fields to store course details, along with an image, a category, whether the course is published or not, and a reference to the user who created the course. The code defining the course fields are given in the following list with explanations:

- **Course name and description:** `name` and `description` fields will have string types, with `name` as a required field:

```
name: {
  type: String,
  trim: true,
  required: 'Name is required'
},
description: {
  type: String,
  trim: true
},
```

- **Course image:** The `image` field will store the course image file to be uploaded by the user as binary data in the MongoDB database:

```
image: {
  data: Buffer,
  contentType: String
},
```

- **Course category:** The `category` field will store the category value of the course as a string, and it will be a required field:

```
category: {
  type: String,
  required: 'Category is required'
},
```

- **Course published state:** The `published` field will be a Boolean value, indicating whether the course is published or not:

```
published: {  
    type: Boolean,  
    default: false  
},
```

- **Course instructor:** The `instructor` field will reference the user who created the course:

```
instructor: {  
    type: mongoose.Schema.ObjectId,  
    ref: 'User'  
}
```

- **Created and updated at times:** The `created` and `updated` fields will be `Date` types, with `created` generated when a new course is added, and `updated` changed when any course details are modified:

```
updated: Date,  
created: {  
    type: Date,  
    default: Date.now  
},
```

The fields in this schema definition will enable us to implement the course-related features in MERN Classroom. To start off these features, in the next section, we will implement the full-stack slice that will allow educators to create new courses.

Creating a new course

In MERN Classroom, a user who is signed in—and who is also an educator—will be able to create new courses. To implement this feature, in the following sections we will add a create course API in the backend, along with a way to fetch this API in the frontend, and a create new course form view that takes user input for course fields.

The create course API

In order to start the implementation of the create course API in the backend, we will add a `POST` route that verifies that the current user is an educator, and then creates a new course with the course data passed in the request body. The route is defined as follows:

mern-classroom/server/routes/course.routes.js:

```
| router.route('/api/courses/by/:userId')
|   .post(authCtrl.requireSignin, authCtrl.hasAuthorization,
|         userCtrl.isEducator,
|         courseCtrl.create)
```

The `course.routes.js` file will be very similar to the `user.routes` file, and to load these new routes in the Express app, we need to mount the course routes in `express.js`, in the same way that we did for the auth and user routes, as shown in the following code:

mern-classroom/server/express.js:

```
| app.use('/', courseRoutes)
```

Next, we will update the user controller to add the `isEducator` method—which will ensure that the current user is actually an educator—before creating the new course. The `isEducator` method is defined as follows:

mern-classroom/server/controllers/user.controller.js:

```
| const isEducator = (req, res, next) => {
|   const isEducator = req.profile && req.profile.educator
|   if (!isEducator) {
|     return res.status('403').json({
|       error: "User is not an educator"
|     })
|   }
|   next()
| }
```

The `create` method, in the course controller, uses the `formidable` Node module to parse the multipart request that may contain an image file that has been uploaded by the user for the course image. If there is a file, `formidable` will store it temporarily in the filesystem, and we will read it using the `fs` module to retrieve the file type and data, and then it will be stored to the `image` field in the course document. The `create` controller method will look as shown in the following code:

mern-classroom/server/controllers/course.controller.js:

```
const create = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        error: "Image could not be uploaded"
      })
    }
    let course = new Course(fields)
    course.instructor= req.profile
    if(files.image){
      course.image.data = fs.readFileSync(files.image.path)
      course.image.contentType = files.image.type
    }
    try {
      let result = await course.save()
      res.json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}
```

The image file for the course, if uploaded by the user, is stored in MongoDB as data. Then, in order to be shown in the views, it is retrieved from the database as an image file at a separate `GET` API route. The `GET` API is set up as an Express route at `/api/courses/photo/:courseId`, which gets the image data from MongoDB and sends it as a file in the response. The implementation steps for file upload, storage, and retrieval are outlined in detail in the *Upload profile photo* section in [Chapter 5](#), *Starting with a Simple Social Media Application*.

With the create course API endpoint ready on the server, next, we can add a `fetch` method in the frontend in order to utilize it.

Fetching the create API in the view

In order to use the create API in the frontend, we will set up a `fetch` method on the client- side to make a `POST` request to the create API, by passing the multipart form data, as shown in the following code:

`mern-classroom/client/course/api-course.js`

```
const create = async (params, credentials, course) => {
  try {
    let response = await fetch('/api/courses/by/' + params.userId, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      },
      body: course
    })
    return response.json()
  } catch(err) {
    console.log(err)
  }
}
```

This method will be used in the new course form view to submit the user-entered course details to the backend to create a new course in the database. In the next section, we will implement this new course form view in a React component.

The NewCourse component

In order to allow educators to create new courses, we will add a React component that contains a form to the frontend of the application. This form view will look as shown in the following screenshot:

The screenshot shows a user interface for creating a new course. At the top center is the title "New Course". Below it is an orange button labeled "UPLOAD PHOTO" with a camera icon. The main area contains three input fields: "Name" (with a placeholder line), "Description" (with a placeholder line), and "Category" (with a placeholder line). At the bottom left is a dark button labeled "SUBMIT", and at the bottom right is a light gray button labeled "CANCEL".

The form will contain an option to upload the course image, input fields for entering the course Name, Description, and Category; and the SUBMIT button, which will save the details that have been entered into the database.

We will define the `NewCourse` React component in order to implement this form. As shown next, we first initialize the state by using the `useState` hook; with empty input field values, an empty error message, and a `redirect` variable that is initialized to `false`.

mern-classroom/client/course/NewCourse.js:

```
export default function NewCourse() {  
  ...  
  const [values, setValues] = useState({
```

```

        name: '',
        description: '',
        image: '',
        category: '',
        redirect: false,
        error: ''
    })
...
}

```

In the form view, we first give the user an option to upload a course image file. To render this option, we will add the file upload elements using a Material-UI button and an HTML5 file input element in the return function for `NewCourse`, as shown in the following code.

`mern-classroom/client/course/NewCourse.js`:

```

<input accept="image/*" onChange={handleChange('image')}
       type="file" style={{display:'none'}} />
<label htmlFor="icon-button-file">
    <Button variant="contained" color="secondary" component="span">
        Upload Photo <FileUpload/>
    </Button>
</label>
<span>{values.image ? values.image.name : ''}</span>

```

Then, we add the `name`, `description`, and `category` form fields using the `TextField` components from Material-UI.

`mern-classroom/client/course/NewCourse.js`:

```

<TextField
    id="name"
    label="Name"
    value={values.name} onChange={handleChange('name')}/> <br/>
<TextField
    id="multiline-flexible"
    label="Description"
    multiline
    rows="2"
    value={values.description}
    onChange={handleChange('description')}/> <br/>
<TextField
    id="category"
    label="Category"
    value={values.category}
    onChange={handleChange('category')}/>

```

We will define a handler function in `NewCourse` so that we can track changes to these fields in the form view. The `handleChange` function will be defined as shown in the following code:

mern-classroom/client/course/NewCourse.js

```
const handleChange = name => event => {
  const value = name === 'image'
    ? event.target.files[0]
    : event.target.value
  setValues({...values, [name]: value })
}
```

This `handleChange` function takes the new value that has been entered into the input field and sets it to state, including the name of the file if one is uploaded by the user.

Finally, in the view, you can add the Submit button, which, when clicked, should call a click-handling function. We will define a function for this purpose in `NewCourse` as follows.

mern-classroom/client/course/NewCourse.js:

```
const clickSubmit = () => {
  let courseData = new FormData()
  values.name && courseData.append('name', values.name)
  values.description && courseData.append('description',
    values.description)
  values.image && courseData.append('image', values.image)
  values.category && courseData.append('category', values.category)
  create({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, courseData).then((data) => {
    if (data.error) {
      setValues({...values, error: data.error})
    } else {
      setValues({...values, error: '', redirect: true})
    }
  })
}
```

This `clickSubmit` function will be called when the form is submitted. It first takes the input values from the state and sets it to a `FormData` object. This ensures that the data is stored in the correct

format that is needed for the `multipart/form-data` encoding type that is necessary for sending requests containing file uploads. Then, the `create` fetch method is called to create a new course in the backend. Finally, depending on the response from the server, either an error message is shown, or the user is redirected to the `MyCourses` view with the following code.

mern-classroom/client/course/NewCourse.js:

```
| if (values.redirect) {  
|     return (<Redirect to={'/teach/courses'} />)  
| }
```

The `NewCourse` component can only be viewed by a signed-in user who is also an educator. So, we will add a `PrivateRoute` to the `MainRouter` component, which will render this form only for authorized users at `/teach/course/new`.

mern-classroom/client/MainRouter.js:

```
| <PrivateRoute path="/teach/course/new" component={NewCourse} />
```

This link can be added to any of the view components that may be accessed by the educator, such as the `MyCourses` view, which will be implemented in the next section in order to list the courses that have been created by an educator.

Listing courses by educator

Authorized educators will be able to see a list of the courses that they have created on the platform. In order to implement this feature, in the following sections we will add a backend API that retrieves the list of courses for a specific instructor, and then we will call this API in the frontend to render this data in a React component.

The list course API

In order to implement the API to return the list of courses that have been created by a specific instructor, first, we will add a route in the backend to retrieve all the courses that have been created by a given user when the server receives a `GET` request at `/api/courses/by/:userId`. This route will be declared as shown next.

mern-classroom/server/routes/course.routes.js:

```
| router.route('/api/courses/by/:userId')
|   .get(authCtrl.requireSignin,
|       authCtrl.hasAuthorization,
|       courseCtrl.listByInstructor)
```

To process the `:userId` param in the route and retrieve the associated user from the database, we will utilize the `userByID` method in our user controller. We will add the following code to the Course routes in `course.routes.js`, so that the user is available in the `request` object as `profile`.

mern-classroom/server/routes/course.routes.js:

```
| router.param('userId', userCtrl.userByID)
```

The `listByInstructor` controller method in `course.controller.js` will query the `Course` collection in the database in order to get the matching courses, as shown next.

mern-classroom/server/controllers/course.controller.js:

```
| const listByInstructor = (req, res) => {
|   Course.find({instructor: req.profile._id}, (err, courses) => {
|     if (err) {
|       return res.status(400).json({
|         error: errorHandler.getErrorMessage(err)
|       })
|     }
|     res.json(courses)
```

```
|   }).populate('instructor', '_id name')  
| }  
|  
| }
```

In the query to the Course collection, we find all the courses that have an `instructor` field that matches the user specified with the `userId` param. Then, the resulting courses are sent back in the response to the client. In the next section, we will see how to call this API from the frontend.

Fetching the list API in the view

In order to use the list API in the frontend, we will define a `fetch` method that can be used by the React components to load this list of courses. The `fetch` method that is needed in order to retrieve a list of courses by a specific instructor will be defined as follows.

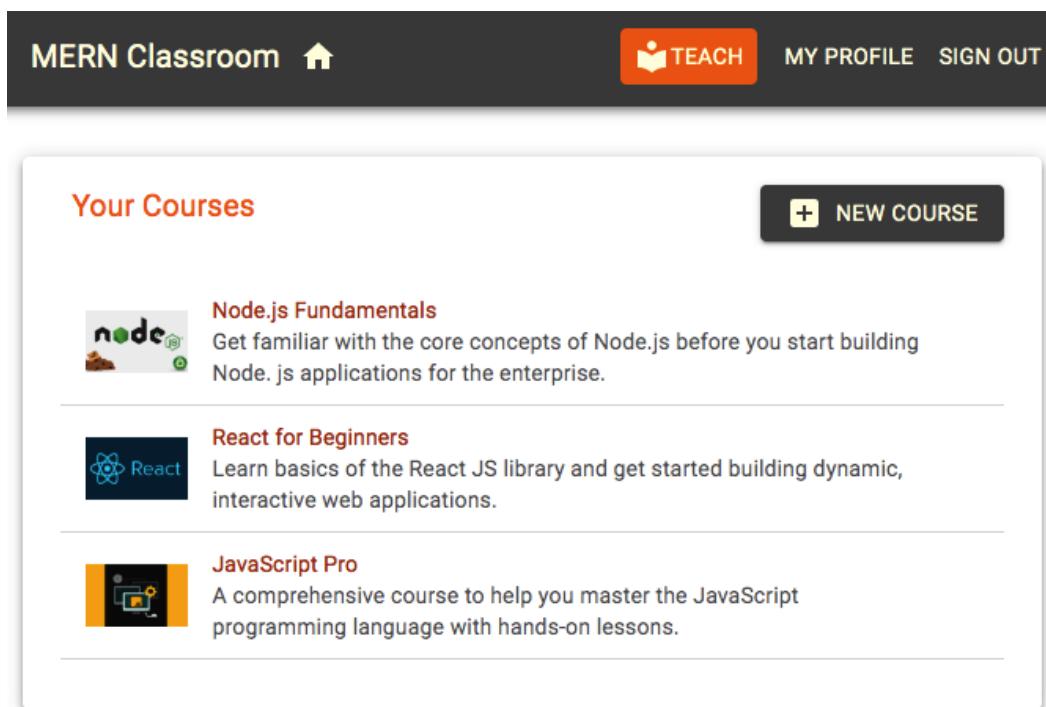
```
mern-classroom/client/course/api-course.js
```

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

This `listByInstructor` method will take the `userId` value in order to generate the API route to be called, and will receive the list of courses that were created by the user associated with the provided `userId` value. In the classroom application, we will utilize this method in the `MyCourses` component, which is discussed in the next section.

The MyCourses component

In the `MyCourses` component, we will render the list of courses in a Material-UI `List`, after fetching the data from the server using the `listByInstructor` API. This component, as pictured in the following image, will function as the educator's dashboard, where their courses are listed and they have an option to add new courses:



In order to implement this component, we first need to fetch and render the list of courses. We will make the fetch API call in the `useEffect` hook, and set the received courses array in the state, as shown next.

`mern-classroom/client/course/MyCourses.js`

```
export default function MyCourses() {
  const [courses, setCourses] = useState([])
  const [redirectToSignin, setRedirectToSignin] = useState(false)
  const jwt = auth.isAuthenticated()
```

```

useEffect(() => {
  const abortController = new AbortController()
  const signal = abortController.signal
  listByInstructor({
    userId: jwt.user._id
  }, {t: jwt.token}, signal).then((data) => {
    if (data.error) {
      setRedirectToSignin(true)
    } else {
      setCourses(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
if (redirectToSignin) {
  return <Redirect to='/signin' />
}
...
}

```

When the `listByInstructor` API is fetched, we pass the currently signed-in user's auth token to check for authorization on the server-side. Users should only be able to see their own courses, and if the current user is not authorized to make this fetch call, the view will be redirected to the sign-in page. Otherwise, the list of courses will be returned and displayed in the view.

In this `MyCourses` component's view, we will render the retrieved courses array by iterating over it using `map`, with each course data rendered in the view in a Material-UI `ListItem`. Each `ListItem` will also be linked to the individual course view, as shown in the following code:

mern-classroom/client/course/MyCourses.js

```

{courses.map((course, i) => {
  return <Link to={"/teach/course/" + course._id} key={i}>
    <ListItem button>
      <ListItemIcon>
        <Avatar src={'/api/courses/photo/' + course._id + "?" +
          new Date().getTime()} />
      </ListItemIcon>
      <ListItemText primary={course.name}
        secondary={course.description} />
    </ListItem>
    <Divider/>
  </Link>}

```

```
| }     )
```

The `MyCourses` component can only be viewed by a signed-in user who is also an educator. So, we will add a `PrivateRoute` in the `MainRouter` component, which will render this component only for authorized users, at `/seller/courses`.

mern-classroom/client/MainRouter.js:

```
| <PrivateRoute path="/seller/courses" component={MyCourses} />
```

We use this frontend route in the "*TEACH*" link on the menu, which directs a signed-in educator to this `MyCourses` view. In this view, users can click on each course in the list, and go to the page that shows the details of a specific course. In the next section, we will implement the feature to render an individual course.

Display a course

Users on the MERN Classroom application, including visitors, signed-in students, and educators alike, will all be able to browse through course pages, with interactions that are specific to their authorization level. In the following sections, we will start implementing the individual course view feature by adding a read course API to the backend, a way to call this API from the frontend, and the React component that will house the course detail view.

A read course API

In order to implement a read course API in the backend, we will start by declaring the `GET` route and the parameter-handling trigger, as shown in the following code.

mern-classroom/server/routes/course.routes.js:

```
router.route('/api/courses/:courseId')
  .get(courseCtrl.read)
  router.param('courseId', courseCtrl.courseByID)
```

We are adding this `GET` route to query the `Course` collection with an ID and return the corresponding course in the response. The `:courseId` param in the route URL will call the `courseByID` controller method, which is similar to the `userByID` controller method. It retrieves the course from the database, and attaches it to the request object that is to be used in the `next` method, as shown in the following code.

mern-classroom/server/controllers/course.controller.js:

```
const courseByID = async (req, res, next, id) => {
  try {
    let course = await Course.findById(id)
      .populate('instructor', '_id name')
    if (!course)
      return res.status('400').json({
        error: "Course not found"
      })
    req.course = course
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve course"
    })
  }
}
```

The course object that is queried from the database will also contain the name and ID details of the instructor, as we specified in the

`populate()` method. The call to `next()` after this course object is attached to the request object invokes the `read` controller method.

The `read` controller method then returns this `course` object in the response to the client, as shown in the following code.

mern-classroom/server/controllers/course.controller.js:

```
| const read = (req, res) => {  
|   req.course.image = undefined  
|   return res.json(req.course)  
| }
```

We are removing the `image` field before sending the response, since images will be retrieved as files in separate routes. With this API ready in the backend, you can now add the implementation in order to call it in the frontend, by adding a `fetch` method in `api-course.js`, which is similar to other `fetch` methods that have already been added. We will use the `fetch` method to call the read course API in the React component that will render the course details, as discussed in the next section.

The Course component

The `Course` component will render the individual course-specific details and user interactions, as pictured in the following screenshot:

The screenshot shows a course page for "Node.js Fundamentals". At the top, it displays the course title, author ("By Jane"), category ("Software Development"), and three action buttons: a pencil icon, "PUBLISH", and a trash bin icon. Below this is a large image featuring the Node.js logo and some brown cardboard boxes. To the right of the image is a description: "Get familiar with the core concepts of Node.js before you start building Node.js applications for the enterprise." Below the image, there's a section titled "Lessons" with a "NEW LESSON" button. It lists five lessons numbered 1 to 5: "Install Node.js", "Node.js Module System", "Asynchronous Node", "HTTP Module", and "File System".

The completed `Course` component will contain the following parts:

- A section showing course details, which is visible to all visitors to this page. We will implement this part in this section.
- A "`Lessons`" section, which contains a list of lessons and is visible to all visitors, and the option to add a new lesson, which will be

visible only to the instructor of this course. We will implement the lessons part in the next section.

- Edit, delete, and publish options, which are visible only to the instructor. This will be discussed later in the chapter.
- An "*Enroll*" option, not pictured in the previous image, which will only be visible after the course has been published by the instructor. This will be implemented later in the chapter.

To start off the implementation of this `Course` component, we will first retrieve the course details with a `fetch` call to the read API in a `useEffect` hook, and then we will set the received values to state, as shown in the following code.

```
mern-classroom/client/course/Course.js
```

```
export default function Course ({match}) {
  const [course, setCourse] = useState({instructor:{}})
  const [values, setValues] = useState({
    error: ''
  })
  useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal

    read({courseId: match.params.courseId}, signal).then((data) => {
      if (data.error) {
        setValues({...values, error: data.error})
      } else {
        setCourse(data)
      }
    })
    return function cleanup() {
      abortController.abort()
    }
  }, [match.params.courseId])
...
}
```

`useEffect` will only run when `courseId` changes in the route params.

In the view, we will render the received details, such as course name, description, category, image, and a link to the instructor's user profile in a Material-UI `Card` component, as shown in the following code:

```
mern-classroom/client/course/Course.js
```

```
<Card>
  <CardHeader
    title={course.name}
    subheader={<div>
      <Link to={"/user/" + course.instructor._id}>
        By {course.instructor.name}
      </Link>
      <span>{course.category}</span>
    </div>
  }
/>
<CardMedia image={imageUrl} title={course.name} />
<div>
  <Typography variant="body1">
    {course.description}
  </Typography>
</div>
</Card>
```

`imageUrl` consists of the route that will retrieve the course image as a file response, and it is constructed as follows:

```
mern-classroom/client/course/Course.js
```

```
const imageUrl = course._id
  ? `/api/courses/photo/${course._id}?${new Date().getTime()}`
  : '/api/courses/defaultphoto'
```

When the course instructor is signed in and views the course page, we will render the edit and other course data-modifying options in the Course component. For now, we will only look at how the `edit` option is added conditionally to the view code:

```
mern-classroom/client/course/Course.js
```

```
{auth.isAuthenticated().user && auth.isAuthenticated().user._id ==
course.instructor._id &&
  (<span><Link to={"/teach/course/edit/" + course._id}>
    <IconButton aria-label="Edit" color="secondary">
      <Edit/>
    </IconButton>
  </Link>
</span>)
}
```

If the current user is signed in, and their ID matches with the course instructor's ID, only then will the "Edit" option be rendered. This part will be edited further in upcoming sections, in order to show the publish and delete options.

In order to load this Course component in the frontend, we will add a route to `MainRouter` as follows:

```
| <Route path="/course/:courseId" component={Course} />
```

This route URL (`/course/:courseId`) can now be added into any component to link to a specific course, with the `:courseId` param replaced with the course's ID value. Clicking on the link will take the user to the corresponding Course view.

We now have the relevant backend model and API endpoints integrated with the frontend views, meaning that we have a functioning implementation of the new course creation, a course listing by the instructor, and single-course display features. We can now move on to extending these implementations further, giving instructors the ability to add lessons to each course and update the course as desired, before publishing it.

Updating courses with lessons

Each course in the MERN Classroom will contain a list of lessons that make up the course content and what the students need to cover when they enroll. We will keep the lesson structure simple for this application, putting more emphasis on the implementation of managing lessons and allowing students to go through lessons in order to complete a course. In the following sections, we will focus on the implementation of managing lessons for a course, and we will also look at how to update an existing course—either to edit details or to delete the course. First, we will look into how to store lesson details, then we will implement the full-stack features to allow instructors to add lessons, update lessons, update details of the course, and delete a course.

Storing lessons

We need to define the lesson data structure and associate it with the course data structure before we can store and retrieve lesson details for each course.

We will start by defining the Lesson model, with a schema containing the title, the content, and the resource URL fields of the string type, as shown in the following code.

```
mern-classroom/server/models/course.model.js
```

```
| const LessonSchema = new mongoose.Schema ( {  
|   title: String,  
|   content: String,  
|   resource_url: String  
| })  
const Lesson = mongoose.model ('Lesson', LessonSchema)
```

These schemas will let educators create and store basic lessons for their courses. To integrate lessons with the course structure, we will add a field called `lessons` in the Course model, which will store an array of lesson documents, as shown in the following code:

```
mern-classroom/server/models/course.model.js
```

```
|   lessons: [LessonSchema]
```

With this updated Course schema and model, we can now proceed with the implementations that will allow educators to add lessons to their course, as discussed in the next section.

Adding new lessons

Educators on the MERN Classroom application will be able to add new lessons to the courses that they are still building and have not yet published. In the following sections, we will make this feature possible, first by implementing a backend API that adds lessons to an existing course, then by creating a frontend form view for entering and sending the new lesson details, and finally, by displaying the newly added lessons on the Course page.

Adding a lesson API

In order to implement a backend API that will allow us to add and store new lessons for a given course, we first need to declare a `PUT` route as follows:

```
mern-classroom/server/routes/course.routes.js:
```

```
router.route('/api/courses/:courseId/lesson/new')
  .put(authCtrl.requireSignin,
        courseCtrl.isInstructor,
        courseCtrl.newLesson)
```

When this route gets a `PUT` request with the course ID in the URL, we will first use the `isInstructor` method to check whether the current user is the instructor for the course, and then we will save the lesson in the database with the `newLesson` method. The `isInstructor` controller method will be defined as follows:

```
mern-classroom/server/controllers/course.controller.js:
```

```
const isInstructor = (req, res, next) => {
  const isInstructor = req.course && req.auth &&
    req.course.instructor._id == req.auth._id
  if(!isInstructor){
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}
```

With the `isInstructor` method, we first check whether the signed-in user has the same user ID as the instructor of the given course. If the user is not authorized, an error is returned in the response, otherwise the `next()` middleware is invoked in order to execute the `newLesson` method. This `newLesson` controller method is defined as follows:

```
mern-classroom/server/controllers/course.controller.js:
```

```
const newLesson = async (req, res) => {
  try {
    let lesson = req.body.lesson
    let result = await Course.findByIdAndUpdate(req.course._id,
      {$push: {lessons: lesson},
       updated: Date.now(),
       new: true})
      .populate('instructor', '_id name')
      .exec()

    res.json(result)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

In this `newLesson` controller method, we use `findByIdAndUpdate` (from MongoDB) to find the corresponding course document, and we update its `lessons` array field by pushing the new lesson object that was received in the request body.

In order to access this API to add a new lesson in the frontend, you will also need to add a corresponding fetch method, as we did for other API implementations.

This API will be used in a form-based component that will take input from the user for each new lesson and then send it to the backend. We will implement this form-based component in the next section.

The NewLesson component

In each course, while it is still unpublished, the instructor will be able to add a lesson by filling out a form. In order to implement this form view to add new lessons, we will create a React component called `NewLesson`, which will be added to the `Course` component. This component will render the following form in a dialog box in the Course page:

Add New Lesson

Title

Content

Resource link

CANCEL ADD

While defining the `NewLesson` component, we will first initialize the form values in the state with the `useState` hook. This component will also receive `props` from the `Course` component, as shown in the following code.

mern-classroom/client/course/NewLesson.js

```
export default function NewLesson(props) {
  const [open, setOpen] = useState(false)
  const [values, setValues] = useState({
    title: '',
    content: '',
    resource_url: ''
  })
  ...
}
NewLesson.propTypes = {
  courseId: PropTypes.string.isRequired,
  addLesson: PropTypes.func.isRequired
}
```

The `NewLesson` component will receive the `courseId` value and an `addLesson` function as `props` from the parent component to which it will be added; in this case from the `Course` component. We make these required `props` by adding `PropTypes` validation to `NewLesson`. These `props` will be needed in this component when the form is submitted.

Next, we will add the button to toggle the dialog that will contain the form, as shown in the following code.

`mern-classroom/client/course/NewLesson.js`

```
<Button aria-label="Add Lesson" color="primary" variant="contained"
       onClick={handleClickOpen}>
    <Add/> New Lesson
</Button>
<Dialog open={open} onClose={ handleClose } aria-labelledby="form-dialog-title">
  <div className={classes.form}>
    <DialogTitle id="form-dialog-title">Add New Lesson</DialogTitle>
    ...
    <DialogActions>
      <Button onClick={ handleClose }
             color="primary" variant="contained">
        Cancel
      </Button>
      <Button onClick={ clickSubmit }
             color="secondary" variant="contained">
        Add
      </Button>
    </DialogActions>
  </div>
</Dialog>
```

The Material-UI `Dialog` component stays open or closed, depending on the state of the `open` variable. We update the `open` value in the following functions, which are called on dialog open and close actions.

`mern-classroom/client/course/NewLesson.js`

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

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

The form fields for entering the new lesson's title, content, and resource URL values are added inside the `Dialog` component using `TextFields` in `DialogContent`, as shown with the following code:

```
mern-classroom/client/course/NewLesson.js
```

```
<DialogContent>
  <TextField label="Title" type="text" fullWidth
    value={values.title} onChange={handleChange('title')} />
  <br/>
  <TextField label="Content" type="text" multiline rows="5" fullWidth
    value={values.content} onChange={handleChange('content')} />
  <br/>
  <TextField label="Resource link" type="text" fullWidth
    value={values.resource_url}
    onChange={handleChange('resource_url')} />
  <br/>
</DialogContent>
```

Values that are entered in the input fields are captured with the `handleChange` function, which is defined as follows:

```
mern-classroom/client/course/NewLesson.js
```

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

Finally, when the form is submitted, we will send the new lesson details to the server in the `clickSubmit` function, as shown in the following code.

```
mern-classroom/client/course/NewLesson.js
```

```
const clickSubmit = () => {
  const jwt = auth.isAuthenticated()
  const lesson = {
    title: values.title || undefined,
    content: values.content || undefined,
    resource_url: values.resource_url || undefined
  }
  newLesson({
    courseId: props.courseId
  }, {
    t: jwt.token
  }, lesson).then((data) => {
    if (data && data.error) {
      setValues({...values, error: data.error})
```

```

        } else {
          props.addLesson(data)
          setValues({...values, title: '',
          content: '',
          resource_url: ''})
          setOpen(false)
        }
      })
}

```

The lesson details are sent in the request to the add lesson API with the course ID received as a prop from the Course component. On a successful update response from the server, besides emptying the form fields, the `addLesson` update function, which was passed as a prop, is executed to render the latest lessons in the Course component. The `addLesson` function to be passed in from the `Course` component is defined as follows:

mern-classroom/client/course/Course.js

```

const addLesson = (course) => {
  setCourse(course)
}

```

The `NewLesson` component that is added to the Course component should only render if the current user is the instructor of the course, and if the course is still unpublished. To do this check and conditionally render the `NewLesson` component, we can add the following code to the Course component:

mern-classroom/client/course/Course.js

```

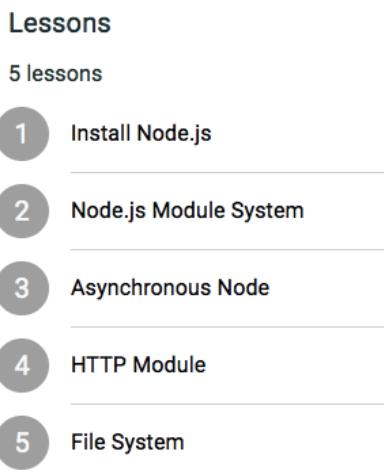
{ auth.isAuthenticated().user &&
  auth.isAuthenticated().user._id == course.instructor._id &&
  !course.published &&
    (<NewLesson courseId={course._id} addLesson={addLesson}/>)
}

```

This will allow educators on the application to add lessons to their courses. Next, we will add the code to render these lessons on the Course page.

Displaying lessons

The lessons for a specific course will be rendered in a list—along with a tally of the total number of lessons—on the Course page below the other course details, as pictured in the following screenshot:



To render this list of lessons, we will update the `Course` component to iterate over the array of lessons with a `map` function, and each lesson will be displayed in a Material-UI `ListItem` component, as shown in the following code.

`mern-classroom/client/course/Course.js`

```
<List>
  {course.lessons && course.lessons.map((lesson, index) => {
    return(<span key={index}>
      <ListItem>
        <ListItemIconAvatar>
          <Avatar> {index+1} </Avatar>
        </ListItemIconAvatar>
        <ListItemText primary={lesson.title} />
      </ListItem>
      <Divider variant="inset" component="li" />
    </span>
  ))}
</List>
```

The number beside each list item is calculated using the current index value of the array. The total number of lessons can also be displayed by accessing `course.lessons.length`.

Now that instructors can add and view lessons for each course, in the next section we will implement the ability to update these added lessons, besides modifying other course details.

Editing a course

Once a course has been added by an educator and there are more updates to be incorporated, the educator will be able to edit the details of the course as its instructor. Editing a course includes the ability to update its lessons, as well. To implement this capability in the application, first, we will have to create a backend API that allows the update operation on a given course.

Then, this updated API needs to be accessed in frontend with the changed details of the course and its lessons. In the following sections, we will build this backend API and the `EditCourse` React component, which will allow instructors to change the course details and lessons.

Updating the course API

In the backend, we will need an API that allows an existing course to be updated if the user who is making the request is the authorized instructor of the given course. We will first declare the PUT route that accepts the update request from the client, as follows:

mern-classroom/server/routes/course.routes.js:

```
router.route('/api/courses/:courseId')
  .put(authCtrl.requireSignin, courseCtrl.isInstructor,
        courseCtrl.update)
```

A `PUT` request that is received at the `/api/courses/:courseId` route first checks if the signed-in user is the instructor of the course that is associated with the `courseId` provided in the URL. If the user is found to be authorized, the `update` controller is invoked. The `update` method in the course controller is defined as shown in the following code.

mern-classroom/server/controllers/course.controller.js:

```
const update = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        error: "Photo could not be uploaded"
      })
    }
    let course = req.course
    course = extend(course, fields)
    if(fields.lessons){
      course.lessons = JSON.parse(fields.lessons)
    }
    course.updated = Date.now()
    if(files.image){
      course.image.data = fs.readFileSync(files.image.path)
      course.image.contentType = files.image.type
    }
    try {
      await course.save()
      res.json(course)
    } catch (err) {
```

```
        return res.status(400).json({
          error: errorHandler.getErrorMessage(err)
        })
      })
    }
  }
```

As the request body may contain a file upload, we are using `formidable` here to parse the multipart data. The lessons array is an array of nested objects, and we need to specifically parse and assign the lessons array to the course before saving it. As we will see in the next section, the lessons array that is sent from the frontend will be stringified before sending, so in this controller, we need to additionally check whether the lessons field was received, and assign it separately after parsing it.

To use this API in the frontend, you will need to define a fetch method that takes the course ID, user auth credentials, and the updated course details, in order to make the fetch call to this update course API—as we have done for other API implementations.

We now have a course update API that can be used in the frontend to update details of a course. We will use this in the `EditCourse` component, which is discussed next.

The EditCourse component

In the frontend, we will add a view for editing a course and it will have two sections. The first part will let the user change the course details, including the name, category, description, and image; and the second part will allow the modification of the lessons for the course. The first part of this course is pictured in the following screenshot:

The screenshot shows a form for editing a course. At the top, there's a title field containing "Node.js Fundamentals" and a "SAVE" button. Below that is a "By" field with "Jane". Under "Category", it says "Software Development". To the right of the form is a large preview image of a Node.js logo featuring a green hexagon and the word "node.js". To the right of the image is a "Description" field containing the text: "Get familiar with the core concepts of Node.js before you start building Node.js applications for the enterprise." Below the description is a "CHANGE PHOTO" button with a camera icon. At the bottom of the screenshot, there's a section titled "Lessons - Edit and Rearrange" which says "5 lessons".

To implement this view, we will define a React component named `EditCourse`. This component will first load the course details by calling the `read` fetch method in the `useEffect` hook, as shown in the following code.

```
mern-classroom/client/course/EditCourse.js
```

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

```

    read({courseId: match.params.courseId}, signal).then((data) => {
      if (data.error) {
        setValues({...values, error: data.error})
      } else {
        setCourse(data)
      }
    })
  return function cleanup() {
    abortController.abort()
  }
}, [match.params.courseId])

```

After successfully receiving the course data in the response, it will be set to the `course` variable in the state by calling `setCourse`, and it will be used to populate the view. The first part of this view will render the course details similar to the Course view but using `TextFields` instead, with an option to upload a new image and a Save button to make the update call, as shown in the following code.

mern-classroom/client/course/EditCourse.js

```

<CardHeader title={<TextField label="Title" type="text" fullWidth
  value={course.name} onChange={handleChange('name')} />
  subheader={<div><Link to={"/user/" + course.instructor._id}>
    By {course.instructor.name}
  </Link>
  <TextField label="Category" type="text" fullWidth
    value={course.category}
    onChange={handleChange('category')} />
  </div>
  action={<Button variant="contained" color="secondary"
    onClick={updateCourse}>Save</Button>}
/>
<div className={classes.flex}>
  <CardMedia image={imageUrl} title={course.name}>
  <div className={classes.details}>
    <TextField multiline rows="5" label="Description" type="text"
      value={course.description}
      onChange={handleChange('description')} /><br/>
    <input accept="image/*"
      onChange={handleChange('image')} type="file" />
    <label htmlFor="icon-button-file">
      <Button variant="outlined" color="secondary" component="span">
        Change Photo
        <FileUpload/>
      </Button>
    </label> <span>{course.image ? course.image.name : ''}</span><br/>
  </div>
</div>

```

The changes to the input fields will be handled in order to capture the newly entered values with the `handleChange` method, which is defined as follows.

```
mern-classroom/client/course/EditCourse.js
```

```
const handleChange = name => event => {
  const value = name === 'image'
    ? event.target.files[0]
    : event.target.value
  setCourse({ ...course, [name]: value })
}
```

When the Save button is clicked, we will get all the course details and set it to `FormData`, which will be sent in the multipart format to the backend using the course update API. The `clickSubmit` function that is called on saving will be defined as follows:

```
mern-classroom/client/course/EditCourse.js
```

```
const clickSubmit = () => {
  let courseData = new FormData()
  course.name && courseData.append('name', course.name)
  course.description && courseData.append('description',
    course.description)
  course.image && courseData.append('image', course.image)
  course.category && courseData.append('category', course.category)
  courseData.append('lessons', JSON.stringify(course.lessons))
  update({
    courseId: match.params.courseId
  }, {
    t: jwt.token
  }, courseData).then((data) => {
    if (data && data.error) {
      console.log(data.error)
      setValues({...values, error: data.error})
    } else {
      setValues({...values, redirect: true})
    }
  })
}
```

The course lessons are also sent in this `FormData`, but as lessons are stored as an array of nested objects and `FormData` only accepts simple key-value pairs, we stringify the `lessons` value before assigning it.

In order to load `EditCourse` in the frontend of the application, we need to declare a frontend route for it. This component can only be viewed by a signed-in user who is also the instructor of the course. So, we will add a `PrivateRoute` in the `MainRouter` component, which will render this view only for authorized users at `/teach/course/edit/:courseId`.

mern-marketplace/client/MainRouter.js:

```
| <PrivateRoute path="/teach/course/edit/:courseId" component={EditCourse}/>
```

This link is added in the Course view in order to allow access to the `EditCourse` page.

We have looked at how to update and send the course details, along with all the lessons to the backend on saving, but we are left with the interface for editing the course lessons. In the following section, we will finish up the `EditCourse` component by looking into the implementation for updating course lessons.

Updating lessons

In order to allow instructors to update the lessons that they have added to a course, we will add the following section in the `EditCourse` component, which will allow the user to edit the lesson details, rearrange the order of the lessons, and delete a lesson:

Lessons - Edit and Rearrange

5 lessons

Lesson Number	Title	Content	Action
1	Node.js Module System	<p>In the Node.js module system, each file is treated as a separate module or a JavaScript library that can be used across the application code.</p> <p>Node.js has a set of built-in modules which you can use without any further installation. Other modules can be installed via the node package manager or npm.</p>	
2	Install Node.js	<p>To get started with Node.js, you first need to download the Node.js source code or a pre-built installer for your platform and install it.</p>	

The implementation of these lesson update features will mostly rely on array manipulation techniques. In the following sections, we will add the interface for an individual lesson in the list, and discuss how the edit, move, and delete functionalities are implemented.

Editing lesson details

Users will be able to edit the details of each field in a lesson in the `EditCourse` component. In the view, each item in the list of lessons will contain three `TextFields` for each of the fields in a lesson. These will be prepopulated with the existing values of the fields as shown in the following code.

```
mern-classroom/client/course/EditCourse.js
```

```
<ListItemIconText
  primary={}><TextField label="Title" type="text" fullWidth
    value={lesson.title}
    onChange={handleLessonChange('title', index)} />
  <br/>
  <TextField multiline rows="5" label="Content" type="text"
    fullWidth value={lesson.content}
    onChange={handleLessonChange('content', index)}/>
  <br/>
  <TextField label="Resource link" type="text" fullWidth
    value={lesson.resource_url}
    onChange={handleLessonChange('resource_url', index)} />
  <br/>
</>
</>
```

In order to handle the changes made to the values in each field, we will define a `handleLessonChange` method, which will take the field name and the corresponding lesson's index in the array.

The `handleLessonChange` method will be defined as follows:

```
mern-classroom/client/course/EditCourse.js
```

```
const handleLessonChange = (name, index) => event => {
  const lessons = course.lessons
  lessons[index][name] = event.target.value
  setCourse({ ...course, lessons: lessons })
}
```

The `lessons` array in the course is updated in the state, after setting the value in the specified field of the lesson at the provided index. This updated course with the modified lesson will get saved to the

database when the user clicks Save in the `EditCourse` view. Next, we will look at how we can allow the user to rearrange the order of the lessons.

Moving the lessons to rearrange the order

While updating lessons, the user will also be able to reorder each lesson on the list. There will be an up arrow button for each lesson, except for the very first lesson. This button will be added to each lesson item in the view as follows:

```
mern-classroom/client/course/EditCourse.js
```

```
{ index != 0 &&
  <IconButton color="primary" onClick={moveUp(index)}>
    <ArrowUp />
  </IconButton>
}
```

When the user clicks this button, the lesson in the current index will be moved up, and the lesson above it will be moved to its place in the array. The `moveUp` function will implement this behavior as follows:

```
mern-classroom/client/course/EditCourse.js
```

```
const moveUp = index => event => {
  const lessons = course.lessons
  const moveUp = lessons[index]
  lessons[index] = lessons[index-1]
  lessons[index-1] = moveUp
  setCourse({ ...course, lessons: lessons })
}
```

The rearranged lessons array is then updated in the state, and will be saved to the database when the user saves the changes in the `EditCourse` page. Next, we will implement the option to delete a lesson from the list.

Deleting a lesson

In the `EditCourse` page, each item rendered in the lessons list will have a delete option. The Delete button will be added in the view to each list item as follows:

```
mern-classroom/client/course/EditCourse.js
```

```
<ListItemSecondaryAction>
  <IconButton edge="end" aria-label="up" color="primary"
    onClick={deleteLesson(index)}>
    <DeleteIcon />
  </IconButton>
</ListItemSecondaryAction>
```

When the Delete button is clicked, we will take the index of the lesson that is being deleted and remove it from the `lessons` array. The `deleteLesson` function, which is called when the button is clicked, is defined as follows:

```
mern-classroom/client/course/EditCourse.js
```

```
const deleteLesson = index => event => {
  const lessons = course.lessons
  lessons.splice(index, 1)
  setCourse({...course, lessons:lessons})
}
```

In this function, we are splicing the array to remove the lesson from the given index, then adding the updated array in the course in the state. This new lesson array will be sent to the database with the course object when the user clicks the Save button in the `EditCourse` page.

This wraps up the three different ways an instructor can change the lessons for their course. With these implementations using array manipulation techniques integrated with the React component's features, the users can now edit the details, rearrange the order, and delete a lesson. In the next section, we will discuss the only

remaining feature for modifying a course, which is the ability to delete it from the database.

Deleting a course

In the MERN Classroom application, instructors will be able to permanently delete courses if the course has not been published already. In order to allow an instructor to delete a course, first, we will define a backend API for course deletion from the database, and then implement a React component that makes use of this API when the user interacts with the frontend to perform this deletion.

The delete course API

In order to implement a backend API that takes a request to delete a specified course from the database, we will first define a DELETE route as shown in the following code.

mern-classroom/server/routes/course.routes.js:

```
router.route('/api/courses/:courseId')
  .delete(authCtrl.requireSignin, courseCtrl.isInstructor,
          courseCtrl.remove)
```

This DELETE route takes the course ID as a URL parameter and checks if the current user is signed in and authorized to perform this delete, before proceeding to the `remove` controller method, which is defined in the following code.

mern-classroom/server/controllers/course.controller.js:

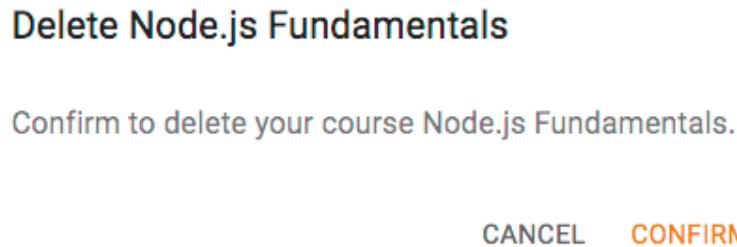
```
const remove = async (req, res) => {
  try {
    let course = req.course
    let deleteCourse = await course.remove()
    res.json(deleteCourse)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The `remove` method simply deletes the course document that corresponds to the provided ID from the Courses collection in the database. To access this backend API in the frontend, you will also need a fetch method with this route; similar to other API implementations. The fetch method will need to take the course ID and current user's auth credentials, then call the delete API with these values.

The fetch method will be used when the user performs the delete operation by clicking a button on the interface. In the next section, we will discuss a React component called `DeleteCourse`, which is where this interaction will take place.

The DeleteCourse component

The instructor for a course will see a delete option on the Course page when they are logged in and they are viewing an unpublished course. This delete option will be implemented in its own React component called `DeleteCourse`, and this component will be added to the `Course` component. The `DeleteCourse` component is basically a button, which, when clicked, opens a `Dialog` component asking the user to confirm the `delete` action, as shown in the following screenshot:



The implementation of the `DeleteCourse` component is similar to the `DeleteUser` component, as discussed in [Chapter 4, Adding a React Frontend to Complete MERN](#). Instead of a user ID, the `DeleteCourse` component will take the course ID and the `onRemove` function definition from the `Course` component as props, when it is added to `Course`, as shown in the following code:

```
mern-classroom/client/course/Course.js
```

```
| <DeleteCourse course={course} onRemove={removeCourse} />
```

With this implementation, course instructors will be able to remove a course from the platform.

In this section, we added the ability to add lessons to a course by extending the Course model and implementing a Lesson model. Then, we added the necessary backend APIs and user interface

updates to be able to add lessons, modify course details and lessons, and delete lessons and courses. The course module is now ready for us to implement the ability to publish a course and make it available on the application for enrollment. We will discuss this publishing feature in the next section.

Publishing courses

In the MERN Classroom, only courses that are published will be available to other users on the platform for enrollment. Once an instructor has created the course and updated it with lessons, they will have the option to publish it. Published courses will be listed on the home page, and all visitors will be able to view them. In the rest of this section, we will look into the implementation of allowing instructors to publish a course and listing these published courses in the frontend.

Implementing the publish option

Instructors for each course will be given the option to publish their course after they have added at least one lesson to the course. Publishing a course will also mean that the course can no longer be deleted, new lessons cannot be added, and existing lessons cannot be deleted. So, when the instructors choose to publish, they will be asked to confirm the action. In this section, we will look at how to use and extend the existing course module in order to integrate this publishing feature.

Publish button states

In the course view, when the instructor is logged in, they will see the PUBLISH button in three states, depending on whether the course can be published or not, and whether it is already published, as shown in the following screenshot:



The states of this button will primarily depend on whether the `published` attribute of the course document is set to `true` or `false`, and on the length of the `lessons` array. The button will be added to the `Course` component, as shown in the following code:

mern-classroom/client/course/Course.js

```
{ !course.published ?  
    (<> <Button color="secondary" variant="outlined"  
        onClick={clickPublish}>  
        { course.lessons.length == 0 ?  
            "Add atleast 1 lesson to publish"  
            : "Publish" }  
    </Button>  
    <DeleteCourse course={course} onRemove={removeCourse}/>  
</>) : (  
    <Button color="primary"  
        variant="outlined">Published</Button>  
)  
}
```

The delete option will only be rendered if the course is not already published. When the PUBLISH button is clicked, we will open a dialog asking the user for confirmation. The `clickPublish` function will be called when the button is clicked, and is defined as follows:

mern-classroom/client/course/Course.js

```
const clickPublish = () => {  
    if(course.lessons.length > 0) {
```

```
        setOpen(true)
    }
}
```

The `clickPublish` function will only open the dialog box if the length of the lessons array is more than zero; preventing the instructor from publishing a course without any lessons. Next, we will add the dialog box, which will let the instructor publish the course after confirmation.

Confirm to publish

When the instructor clicks on the PUBLISH button, they will see a dialog box informing them of the consequences of this action, and giving them the options to PUBLISH the course or CANCEL the action. The dialog box will look as follows:

Publish Course

Publishing your course will make it live to students for enrollment.
Make sure all lessons are added and ready for publishing.

CANCEL PUBLISH

To implement this dialog box, we will use the Material-UI Dialog component with the title and content text, and the PUBLISH and CANCEL buttons, as shown in the following code.

mern-classroom/client/course/Course.js

```
<Dialog open={open} onClose={handleClose} aria-labelledby="form-dialog-title">
  <DialogTitle id="form-dialog-title">Publish Course</DialogTitle>
  <DialogContent>
    <Typography variant="body1">
      Publishing your course will make it live to students
      for enrollment.
    </Typography>
    <Typography variant="body1">
      Make sure all lessons are added and ready for publishing.
    </Typography>
  </DialogContent>
  <DialogActions>
    <Button onClick={handleClose} color="primary" variant="contained">
      Cancel
    </Button>
    <Button onClick={publish} color="secondary" variant="contained">
      Publish
    </Button>
  </DialogActions>
</Dialog>
```

```
|   </DialogActions>
|</Dialog>
```

When the PUBLISH button on the dialog is clicked by the user as confirmation to publish the course, we will make an update API call to the backend, with the `published` attribute of the course set to `true`. The `publish` function to make this update will be defined as follows:

mern-classroom/client/course/Course.js

```
const publish = () => {
  let courseData = new FormData()
  courseData.append('published', true)
  update({
    courseId: match.params.courseId
  }, {
    t: jwt.token
  }, courseData).then((data) => {
    if (data && data.error) {
      setValues({...values, error: data.error})
    } else {
      setCourse({...course, published: true})
      setOpen(false)
    }
  })
}
```

In this function, we are using the same update API that has already been defined and used for saving modifications to other course details from the `EditCourse` view. Once the backend is successfully updated with the `published` value, it is also updated in the state of the `Course` component.

This `published` attribute in the course can be used to conditionally hide the options to add a new lesson, delete a course, and delete a lesson in both the `Course` and `EditCourse` components, in order to prevent the instructor from performing these actions after the course is already published. As courses are published by instructors, these courses will be listed in a view for all users on the platform, as discussed in the following section.

Listing published courses

All visitors to the MERN Classroom application will be able to access the published courses. In order to present these published courses, we will add the feature to retrieve all the published courses from the database, and display the courses in a list on the home page. In the following sections, we will implement this feature by first defining the backend API, which will take a request and return the list of published courses. Then, we will implement the frontend component that will fetch this API and render the courses.

The published courses API

In order to retrieve the list of published courses from the database, we will implement an API in the backend, by first declaring the route that will take a GET request at `'/api/courses/published'`, as shown in the following code:

mern-classroom/server/routes/course.routes.js:

```
| router.route('/api/courses/published')
|   .get(courseCtrl.listPublished)
```

A GET request to this route will invoke the `listPublished` controller method, which initiates a query to the Course collection for courses that have the `published` attribute's value as `true`. Then, the resulting courses are returned in the response. The `listPublished` controller method is defined as follows:

mern-classroom/server/controllers/course.controller.js:

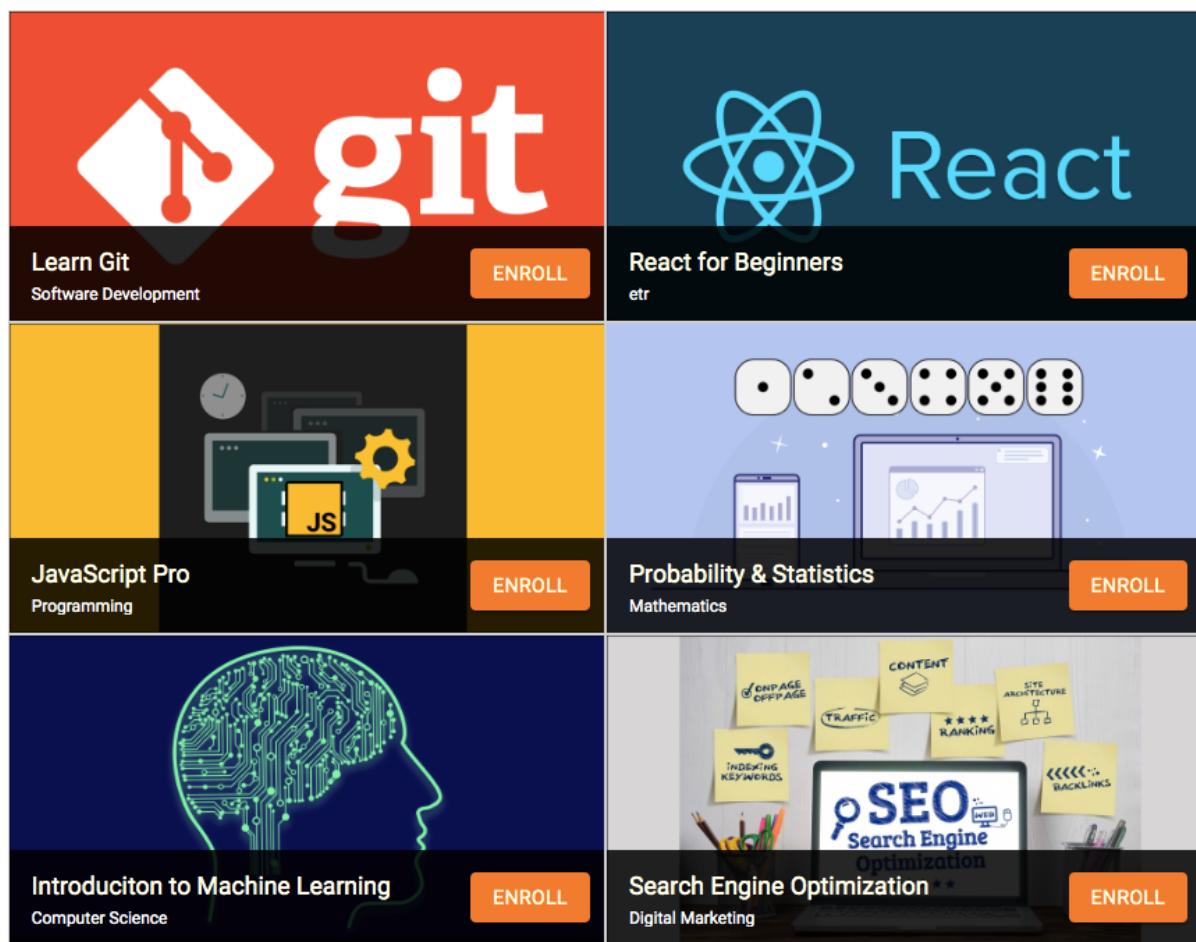
```
| const listPublished = (req, res) => {
|   Course.find({published: true}, (err, courses) => {
|     if (err) {
|       return res.status(400).json({
|         error: errorHandler.getErrorMessage(err)
|       })
|     }
|     res.json(courses)
|   }).populate('instructor', '_id name')
| }
```

To use this list API in the frontend, we also need to define a fetch method on the client-side, as we did for all the other API calls. Then, the fetch method will be used in the component, which will retrieve and display the published courses. In the next section, we will look into the implementation of rendering the retrieved course list in a React component.

The Courses component

For displaying the list of published courses, we will design a component that takes the array of courses as props from the parent component that it is added to. In the MERN Classroom application, we will render the published courses on the home page, as pictured in the next screenshot:

All Courses



In the `Home` component, we will retrieve the list of published courses from the backend in a `useEffect` hook, as shown in the following code:

```
mern-classroom/client/core/Home.js
```

```
useEffect(() => {
  const abortController = new AbortController()
  const signal = abortController.signal
  listPublished(signal).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      setCourses(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

Once the list of courses is received, it is set to the `courses` variable in the state. We will pass this `courses` array to the `Courses` component as props when it is added to the `Home` component, as follows:

```
mern-classroom/client/core/Home.js
```

```
| <Courses courses={courses} />
```

This `Courses` component will take these props and iterate through the array to render each course in a `GridList` component from Material-UI. The `Courses` component is defined as shown in the following code:

```
mern-classroom/client/course/Courses.js
```

```
export default function Courses(props) {
  return (
    <GridList cellHeight={220} cols={2}>
      {props.courses.map((course, i) => {
        return (
          <GridListTile key={i} style={{padding:0}}>
            <Link to={"/course/" + course._id}>
              <img src={'/api/courses/photo/' + course._id}
                  alt={course.name} />
            </Link>
            <GridListTileBar
              title={<Link to={"/course/" + course._id}>
                {course.name}</Link>}
              subtitle={<span>{course.category}</span>}
              actionIcon={auth.isAuthenticated() ?
                <Enroll courseId={course._id}/> :
                <Link to="/signin">
                  Sign in to Enroll</Link>
              }
            </GridListTileBar>
        )
      )}
    </GridList>
  )
}
```

```
        }
      />
    </GridListTile>
  ) )
</GridList>
)
}
Courses.propTypes = {
  courses: PropTypes.array.isRequired
}
```

Each course in the list will display its name, category, and image, and will be linked to the individual course page. The Enroll option, which will be implemented in its own component, will also be shown for each course, but only to the users who are signed in, and are browsing through the home page.

With courses now publishable by instructors and viewable by all visitors to the application, we can now start the implementation for enrollment on courses.

Enrolling on courses

All visitors to the MERN Classroom application will have the option to sign in and then enroll on any of the published courses. Enrolling on a course would give them access to the lesson details and would allow them to go through the lessons systematically to complete the course. In order to implement this feature, in this section, we will first define an Enrollment model to store enrollment details in the database. Then, we will add the backend API to create new enrollments when end users interact with the `Enroll` component that will be added to the frontend. Finally, we will implement the view that enables a student to see and interact with the content from the course on which they are enrolled.

Defining an Enrollment model

We will define an Enrollment schema and model in order to store the details of each enrollment in the application. It will have fields to store the reference to the course being enrolled in and the user who is enrolling as a student. It will also store an array corresponding to the lessons in the associated course, which will store the completion status of each of the lessons for this student. Additionally, we will store three timestamp values; the first value will signify when the student enrolled, the second value will indicate the last time that they completed a lesson or updated the enrollment, and finally, when they completed the course. This enrollment model will be defined in `server/models/enrollment.model.js`, and the code defining the enrollment fields are given in the following list with explanations:

- **Course reference:** The `course` field will store the reference to the course document with which this enrollment is associated:

```
| course: {  
|   type: mongoose.Schema.ObjectId,  
|   ref: 'Course'  
| }
```

- **Student reference:** The `student` field will store the reference to the user who created this enrollment by choosing to enroll on a course:

```
| student: {  
|   type: mongoose.Schema.ObjectId,  
|   ref: 'User'  
| }
```

- **Lesson status:** The `lessonStatus` field will store an array with references to each lesson that is stored in the associated course in the `lessons` array. For each object in this `lessonStatus` array, we will add a `complete` field that will store a Boolean value that indicates whether the corresponding lesson has been completed or not:

```
    lessonStatus: [ {
      lesson: {type: mongoose.Schema.ObjectId, ref: 'Lesson'},
      complete: Boolean
    } ]
```

- **Enrolled at:** The `enrolled` field will be a `Date` value indicating the time that the enrollment was created; in other words, when the student enrolled on the course:

```
    enrolled: {
      type: Date,
      default: Date.now
    }
```

- **Updated at:** The `updated` field will be another `Date` value, which will be updated every time a lesson is completed, indicating when was the last time that the user worked on the course lessons:

```
    updated: Date
```

- **Completed at:** The `completed` field will also be a `Date` type, which will only be set when all the lessons in the course have been completed:

```
    completed: Date
```

The fields in this schema definition will enable us to implement all the enrollment-related features in MERN Classroom. In the next section, we will implement the user's ability to enroll on a course, and store details of the enrollment using this Enrollment model.

The create Enrollment API

When a user chooses to enroll in a course, we will create a new enrollment and store it in the backend. To implement this feature, we need to define a create enrollment API on the server, by first declaring a route that accepts a `POST` request at

'/api/enrollment/new/:courseId', as shown in the following code:

mern-classroom/server/routes/enrollment.routes.js:

```
| router.route('/api/enrollment/new/:courseId')
|   .get(authCtrl.requireSignin, enrollmentCtrl.findEnrollment,
| enrollmentCtrl.create)
| router.param('courseId', courseCtrl.courseByID)
```

This route takes the course ID as a parameter in the URL. Hence, we also add the `courseByID` controller method from the course controllers in order to process this parameter and retrieve the corresponding course from the database. The user who initiates the request from the client- side is identified from the user auth credentials sent in the request. A `POST` request received at this route will first check whether the user is authenticated, and then check whether they are already enrolled on this course, before creating a new enrollment for this user in this course.

The `findEnrollment` controller method will query the `Enrollments` collection in the database in order to check whether there is already an enrollment with the given course ID and user ID. The `findEnrollment` method is defined as follows.

mern-classroom/server/controllers/enrollment.controller.js:

```
| const findEnrollment = async (req, res, next) => {
|   try {
|     let enrollments = await Enrollment.find({course:req.course._id,
|                                               student: req.auth._id})
|     if(enrollments.length == 0) {
|       next()
```

```

        }else{
          res.json(enrollments[0])
        }
      } catch (err) {
        return res.status(400).json({
          error: errorHandler.getErrorMessage(err)
        })
      }
    }
  }
}

```

If a matching result is returned from the query, then the resulting enrollment will be sent back in the response, otherwise, the `create` controller method will be invoked to create a new enrollment.

The `create` controller method generates a new enrollment object to be saved into the database from the course reference, user reference, and the `lessons` array in the given course. The `create` method is defined as shown in the following code.

mern-classroom/server/controllers/enrollment.controller.js:

```

const create = async (req, res) => {
  let newEnrollment = {
    course: req.course,
    student: req.auth,
  }
  newEnrollment.lessonStatus = req.course.lessons.map((lesson)=>{
    return {lesson: lesson, complete:false}
  })
  const enrollment = new Enrollment(newEnrollment)
  try {
    let result = await enrollment.save()
    return res.status(200).json(result)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

```

The `lessons` array in `course` is iterated over to generate the `lessonStatus` array of objects for the new enrollment document. Each object in the `lessonStatus` array has the `complete` value initialized to `false`. On successful saving of the new enrollment document based on these values, the new document is sent back in the response.

All the routes that are defined for enrollment APIs, such as this create API, are declared in the `enrollment.routes.js` file, and it will be similar to the other route files that have already been created in our application. As with the other routes, we need to load these new routes in the Express app by mounting the enrollment routes in `express.js`. The enrollment-related routes are mounted as follows.

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

```
| app.use('/', enrollmentRoutes)
```

To access the create API in the frontend, you will also need to define a fetch method similar to other fetch methods that have been defined in the application. Using this fetch method, the `Enroll` component that is discussed in the next section will be able to call this create enrollment API.

The Enroll component

The `Enroll` component will simply contain a button that initiates the enrollment call to the backend, and redirects the user if the server returns successfully with the new enrollment document's ID. This component takes the ID of the associated course as a prop from the parent component from where it is added. This prop will be used while making the create enrollment API call. The `Enroll` component is defined as shown in the following code.

mern-classroom/client/enrollment/Enroll.js:

```
export default function Enroll(props) {
  const [values, setValues] = useState({
    enrollmentId: '',
    error: '',
    redirect: false
  })
  if(values.redirect){
    return (<Redirect to={'/learn/' + values.enrollmentId}/>)
  }
  return (
    <Button variant="contained" color="secondary"
      onClick={clickEnroll}> Enroll </Button>
  )
}
```

When the ENROLL button is clicked, the create enrollment API will be fetched with the provided course ID to either retrieve an existing enrollment, or to create a new enrollment and receive it in the response. The `clickEnroll` function to be invoked when the button is clicked is defined as follows:

mern-classroom/client/enrollment/Enroll.js:

```
const clickEnroll = () => {
  const jwt = auth.isAuthenticated()
  create({
    courseId: props.courseId
  }, {
    t: jwt.token
  }).then((data) => {
    console.log(data)
```

```
    if (data && data.error) {
      setValues({...values, error: data.error})
    } else {
      setValues({...values, enrollmentId: data._id, redirect: true})
    }
  })
}
```

When the server sends back an enrollment successfully, the user will be redirected to the view that will display the details of the specific enrollment.

Since the `Enroll` component receives the course ID as a prop from the parent component, we also add `PropTypes` validation (as shown in the following code) for the component, as its functionality and implementation relies on this prop being passed.

`mern-classroom/client/enrollment/Enroll.js`:

```
Enroll.propTypes = {
  courseId: PropTypes.string.isRequired
}
```

When a server responds successfully on the API call, the user is redirected to the enrolled course view, where they can go through the lesson content. We will work on implementing this view in the next section.

The Enrolled Course view

For each course on which the user is enrolled, they will see a view that lists the details of the course, and each lesson in the course; with the option to complete each lesson. In the following sections we will implement this view, by first adding a backend API that returns a given enrollment's details, and then using this API in the frontend to build the enrolled course view.

The read enrollment API

The backend API which will return the enrollment details from the database will be defined as a GET route that accepts the request at `'/api/enrollment/:enrollmentId'`, and will be declared as follows:

```
mern-classroom/server/routes/enrollment.routes.js
```

```
router.route('/api/enrollment/:enrollmentId')
  .get(authCtrl.requireSignin, enrollmentCtrl.isStudent,
        enrollmentCtrl.read)
  router.param('enrollmentId', enrollmentCtrl.enrollmentByID)
```

A GET request at this route will first invoke the `enrollmentByID` method, since it contains the `enrollmentId` param in the URL declaration. The `enrollmentByID` method will query the `Enrollments` collection by the provided ID, and if a matching enrollment document is found, we ensure that the referenced course, the nested course instructor, and the referenced student details are also populated using the `populate` method from Mongoose. The `enrollmentByID` controller method is defined as shown in the following code:

```
mern-classroom/server/controllers/enrollment.controller.js:
```

```
const enrollmentByID = async (req, res, next, id) => {
  try {
    let enrollment = await Enrollment.findById(id)
      .populate({path: 'course', populate:{path: 'instructor'}})
      .populate('student', '_id name')
    if (!enrollment)
      return res.status('400').json({
        error: "Enrollment not found"
      })
    req.enrollment = enrollment
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve enrollment"
    })
  }
}
```

The resulting enrollment object is attached to the request object and passed on to the next controller method. Before returning this enrollment object in the response to the client, we will check whether the currently signed-in user is the student who is associated with this specific enrollment in the `isStudent` method, as defined in the following code.

mern-classroom/server/controllers/enrollment.controller.js:

```
const isStudent = (req, res, next) => {
  const isStudent = req.auth && req.auth._id ==
    req.enrollment.student._id
  if (!isStudent) {
    return res.status('403').json({
      error: "User is not enrolled"
    })
  }
  next()
}
```

The `isStudent` method checks whether the user who is identified by the auth credentials that were sent in the request matches the student who is referenced in the enrollment. If the two users don't match, a 403 status is returned with an error message, otherwise, the next controller method is invoked in order to return the enrollment object. The next controller method is the `read` method, and it is defined as follows:

mern-classroom/server/controllers/enrollment.controller.js:

```
const read = (req, res) => {
  return res.json(req.enrollment)
}
```

To use this read enrollment API in the frontend, you will also need to define a corresponding fetch method, as implemented for all other APIs in this application. Then, this fetch method will be used to retrieve the enrollment details to be rendered in a React component that the student will interact with. We will implement this `Enrollment` component in the next section.

The Enrollment component

The `Enrollment` component will load the details of the course and the lessons that were received from the read enrollment API. In this view, students will be able to go through each lesson in the course and mark each as complete. The lesson titles will be listed in a drawer, giving the student an overall idea of what the course contains, and how far they have progressed. Each item in the drawer will extend to reveal the details of the lesson, as pictured in the following screenshot:

The screenshot shows the 'MERN Classroom' application interface. On the left, there's a sidebar with a 'Course Overview' section containing a single item: 'Node.js Fundamentals'. Below it is a 'Lessons' section with five items numbered 1 to 5: 'Install Node.js', 'Node.js Module System', 'Asynchronous Node', 'HTTP Module', and 'File System'. The second item, 'Node.js Module System', is currently expanded, showing its details. The main content area displays the title 'Node.js Module System' and a description: 'In the Node.js module system, each file is treated as a separate module or a JavaScript library that can be used across the application code.' It also contains some sample code:

```
const calculate = require('./calculate.js')
const sum = calculate.sum(10, 35)
```

 and a note: 'Node.js has a set of built-in modules which you can use without any further installation. Other modules can be installed via the node package manager or npm.' At the bottom of the expanded lesson card is a 'RESOURCE LINK' button.

To implement this view, first, we need to make a fetch call to the read enrollment API in the `useEffect` hook in order to retrieve the details of the enrollment and set it to state, as shown in the following code.

mern-classroom/client/enrollment/Enrollment.js:

```
export default function Enrollment ({match}) {
  const [enrollment, setEnrollment] = useState({course:{instructor:[], lessonStatus: []}})
  const [values, setValues] = useState({
    redirect: false,
```

```

        error: '',
        drawer: -1
    })
const jwt = auth.isAuthenticated()
useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal
    read({enrollmentId: match.params.enrollmentId},
        {t: jwt.token}, signal).then((data) => {
        if (data.error) {
            setValues({...values, error: data.error})
        } else {
            setEnrollment(data)
        }
    })
    return function cleanup(){
        abortController.abort()
    }
}, [match.params.enrollmentId])
....
```

We will implement the drawer layout using Material-UI's `Drawer` component. In the drawer, we keep the first item as the Course Overview, which will give the user an overview of the course details, similar to the single course page. When the user enters this enrollment view, they will see the Course Overview first.

In the following code, after adding this first drawer item, we create a separate section for the lessons, where the `lessonStatus` array is iterated over to list the lesson titles in the drawer.

mern-classroom/client/enrollment/Enrollment.js:

```

<Drawer variant="permanent">
    <div className={classes.toolbar} />
    <List>
        <ListItem button onClick={selectDrawer(-1)}
            className={values.drawer == -1 ?
                classes.selectedDrawer : classes.unselected}>
            <ListItemIcon><Info /></ListItemIcon>
            <ListItemText primary={"Course Overview"} />
        </ListItem>
    </List>
    <Divider />
    <List>
        <ListSubheader component="div">
            Lessons
        </ListSubheader>
        {enrollment.lessonStatus.map((lesson, index) => (
            <ListItem button key={index} onClick={selectDrawer(index)}
                className={values.drawer == index ?
                    classes.selectedDrawer : classes.unselected}>
                <ListItemIcon>
                    <Avatar> {index+1} </Avatar>
```

```

        </ListItemAvatar>
        <ListItemText
            primary={enrollment.course.lessons[index].title} />
        <ListItemSecondaryAction> { lesson.complete ?
            <CheckCircle/> : <RadioButtonUncheckedIcon />}
        </ListItemSecondaryAction>
    </ListItem>
    ))
)
</List>
<Divider />
</Drawer>

```

Each of the items in the Lessons section of the drawer will also give the user a visual indication of whether the lesson has been completed, or is still incomplete. These check or uncheck icons will be rendered based on the Boolean value of the `complete` field in each item in the `lessonStatus` array.

To determine which drawer is currently selected, we will utilize the initialized `drawer` value to state with a -1. The -1 value will be associated with the Course Overview drawer item and view, whereas the index of each `lessonStatus` item will determine which lesson is displayed when selected from the drawer. When a drawer item is clicked, we will call the `selectDrawer` method, giving it either -1 or the index of the lesson clicked as its argument. The `selectDrawer` method is defined as follows:

mern-classroom/client/enrollment/Enrollment.js:

```

const selectDrawer = (index) => event => {
    setValues({...values, drawer:index})
}

```

This `selectDrawer` method sets the `drawer` value in the state according to the item clicked on the drawer. The actual content view will also render conditionally, depending on this `drawer` value, according to the following structure:

```

{ values.drawer == - 1 && (Overview of course) }
{ values.drawer != - 1 && (Individual lesson content based on the index value
represented in drawer) }

```

The course overview section can be designed and implemented according to the Course page. In order to render the individual lesson details, we can use a `Card` component as follows:

mern-classroom/client/enrollment/Enrollment.js:

```

{values.drawer != -1 && (<>
    <Typography variant="h5">{enrollment.course.name}</Typography>
    <Card> <CardHeader
        title={enrollment.course.lessons[values.drawer].title}
    />
    <CardContent>
        <Typography variant="body1">
            {enrollment.course.lessons[values.drawer].content}
        </Typography>
    </CardContent>
    <CardActions>
        <a href=
{enrollment.course.lessons[values.drawer].resource_url}>
            <Button variant="contained" color="primary">
                Resource Link</Button>
        </a>
    </CardActions>
</Card>
</>
) }

```

This will render the details of the lesson that has been selected, which are the title, content, and resource URL values. With this implementation, we now have a way to let users enroll on courses and view the details of their enrollment. This enrollment data is initially created from the course details, but will also store details that are specific to the student who enrolled, and their progress in the lessons and the course overall. In order to be able to record and track this progress, and then display the related statistical information to both students and instructors, we will update this implementation further in the following section in order to add these capabilities.

Tracking progress and enrollment stats

In a classroom application such as MERN Classroom, it can be valuable to let students visualize their progress in enrolled courses, and let instructors see how many students enrolled and completed their courses.

In this application, once a student is enrolled on a course, they will be able to go through each lesson in it, and mark it complete until all the lessons are done, and the whole course is complete. The application will leave visual cues to let a student know the state of their enrollments in courses. For instructors, once they publish a course, we will show the total number of students who enrolled on the course, and the total number of students who completed the course.

In the following sections, we will implement these capabilities, starting with letting users complete lessons and track their progress in a course, then listing their enrollments with indicators for which ones are complete and which are in progress, and finally, showing the enrollment stats for each published course.

Completing lessons

We will have to extend both the enrollment APIs and the enrollment view implementation to allow students first to complete lessons, and then the whole course. We will add a lesson complete API in the backend and use this API in the frontend to mark a lesson as complete when the user performs this action. In the following sections, we will add this API, then modify the `Enrollment` component to use this API, and visually indicate which lessons are complete.

Lessons completed API

We will add a `complete` API endpoint in the backend for enrollments, which will mark specified lessons as complete, and will also mark the enrolled course as completed when all the lessons are done. To implement this API, we will start by declaring a PUT route, as shown in the following code:

```
mern-classroom/server/routes/enrollment.routes.js

router.route('/api/enrollment/complete/:enrollmentId')
  .put(authCtrl.requireSignin,
    enrollmentCtrl.isStudent,
    enrollmentCtrl.complete)
```

When a PUT request is received at the `'/api/enrollment/complete/:enrollmentId'` URL, we will first make sure that the signed-in user is the student who is associated with this enrollment record, and then we will call the `complete` enrollment controller method. The `complete` method is defined as follows:

```
mern-classroom/server/controllers/enrollment.controller.js

const complete = async (req, res) => {
  let updatedData = {}
  updatedData['lessonStatus.$.complete'] = req.body.complete
  updatedData.updated = Date.now()
  if(req.body.courseCompleted)
    updatedData.completed = req.body.courseCompleted
  try {
    let enrollment = await
      Enrollment.updateOne({'lessonStatus._id':
        req.body.lessonStatusId},
      {'$set': updatedData})
    res.json(enrollment)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

In this `complete` method, we use the `updateOne` action from MongoDB to update the enrollment document, which contains the `lessonStatus` object with the corresponding `lessonStatusId` value provided in the request.

In the resulting enrollment document, we update the `complete` field of the specific object in the `lessonStatus` array, and the `updated` field of the enrollment document. If a `courseCompleted` value is sent in the request, we also update the `completed` field in the enrollment document. Once the enrollment document is updated successfully, it is sent back in the response.

To use this `complete` API endpoint in the frontend, you also need to define a corresponding fetch method like we did for other API implementations. This fetch method should make a PUT request to the complete enrollment route with related values sent in the request. As discussed in the next section, we will use this implemented API in the `Enrollment` component in order to allow students to complete lessons.

Completed lessons from the view

In the `Enrollment` component, in which we are rendering each lesson's details in the drawer view, we will give the student the option to mark the lesson as completed. This option will render conditionally, depending on whether the given lesson is already completed or not. This option will be added to the `action` property in `CardHeader`, as shown in the following code:

mern-classroom/client/enrollment/Enrollment.js:

```
| action={<Button
|   onClick={markComplete}
|   variant={enrollment.lessonStatus[values.drawer].complete ?
|     'contained' : 'outlined'} color="secondary">
|   {enrollment.lessonStatus[values.drawer].complete ?
|     "Completed" : "Mark as complete"}
| </Button>}
```

If the given `lessonStatus` object has the `complete` attribute set to `true`, then we render a filled-out button with the text `Completed`, otherwise an outlined button is rendered with the text `Mark as complete`. Clicking on this button makes a call to the `markComplete` function, which will make the API call to update the enrollment in the database. This `markComplete` function is defined as follows:

mern-classroom/client/enrollment/Enrollment.js:

```
| const markComplete = () => {
|   if(!enrollment.lessonStatus[values.drawer].complete){
|     const lessonStatus = enrollment.lessonStatus
|     lessonStatus[values.drawer].complete = true
|     let count = totalCompleted(lessonStatus)
|     let updatedData = {}
|     updatedData.lessonStatusId = lessonStatus[values.drawer]._id
|     updatedData.complete = true
|     if(count == lessonStatus.length){
|       updatedData.courseCompleted = Date.now()
|     }
|     complete({
|       enrollmentId: match.params.enrollmentId
|     }, {
|       t: jwt.token
|     }, updatedData).then((data) => {
|       if (data && data.error) {
|         setValues({...values, error: data.error})
|       } else {
|         setEnrollment({...enrollment, lessonStatus: lessonStatus})
|       }
|     })
|   }
| }
```

In this function, before making the API call to the backend, we prepare the values to be sent with the request in the `updatedData` object. We send the `lessonStatus` details, including the ID value and `complete` value set to `true` for the lesson that was completed by the user. We also calculate if the total number of completed lessons is equal to the total number of lessons, so that we can set and send the `courseCompleted` value in the request, as well.

The total number of completed lessons is calculated using the `totalCompleted` function, which is defined as follows:

mern-classroom/client/enrollment/Enrollment.js:

```
| const totalCompleted = (lessons) => {
|   let count = lessons.reduce((total, lessonStatus) => {
|     return total + (lessonStatus.complete ? 1 : 0)}, 0)
|   setTotalComplete(count)
```

```
| } return count
```

We use the array `reduce` function to find and tally the count for the completed lessons in the `lessonStatus` array. This count value is also stored in the state, so that it can be rendered in the view at the bottom of the drawer, as shown in the following screenshot:

The screenshot shows a mobile-style interface for a course overview. At the top, there's a header with the title "MERN Classroom" and a home icon. Below the header is a navigation bar with an info icon and the text "Course Overview". The main content area is titled "Lessons" and lists five items, each with a circular index number and a description followed by a completion status indicator (checkmark or outline). At the bottom of the list is a summary bar with the text "2 out of 5 completed".

Lesson Number	Description	Status
1	Install Node.js	✓
2	Node.js Module System	✓
3	Asynchronous Node	○
4	HTTP Module	○
5	File System	○

The student's lessons will have a check icon next to them, as an indication of which lessons are either complete or incomplete. We also give the student a number tally of how many were completed out of the total. The course is considered completed when all the lessons are done. This gives the student an idea of their progress in the course. Next, we will add a feature that will allow users to see the state of all the courses on which they are enrolled.

Listing all enrollments for a user

Once they are signed in to MERN Classroom, students will be able to view a list of all their enrollments on the home page. In order to implement this feature, we will first define a backend API, which returns the list of enrollments for a given user, and then use it in the frontend to render the list of enrollments to the user.

The list of enrollments API

The list of enrollments API will take a GET request and query the `Enrollments` collection in order to find enrollments that have a student reference that matches with the user who is currently signed in. To implement this API, we will first declare the GET route for `'/api/enrollment/enrolled'`, as shown in the following code:

```
mern-classroom/server/routes/enrollment.routes.js
```

```
| router.route('/api/enrollment/enrolled')
|   .get(authCtrl.requireSignin, enrollmentCtrl.listEnrolled)
```

A GET request to this route will invoke the `listEnrolled` controller method, which will query the database and return the results in the response to the client. The `listEnrolled` method is defined as follows:

```
mern-classroom/server/controllers/enrollment.controller.js
```

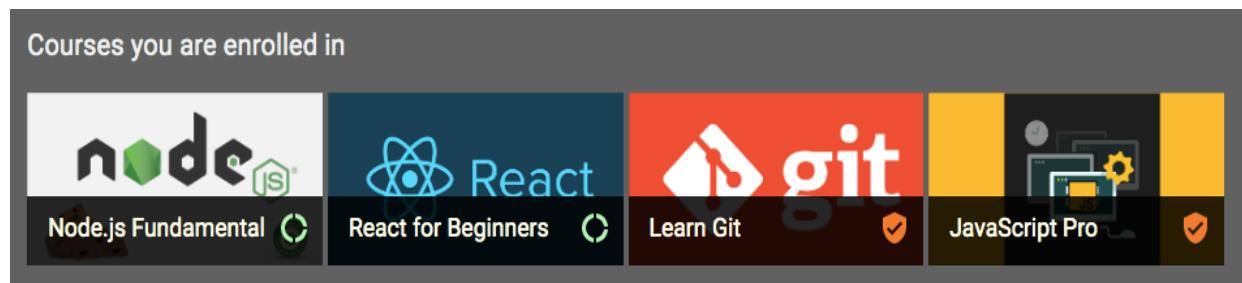
```
const listEnrolled = async (req, res) => {
  try {
    let enrollments = await Enrollment.find({student: req.auth._id})
      .sort({'completed': 1})
      .populate('course', '_id name
category')
    res.json(enrollments)
  } catch (err) {
    console.log(err)
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The query to the `Enrollments` collection finds all enrollments with the student reference that matches the user ID that was received in the auth credentials of the currently signed-in user. The resulting enrollments will be populated with the referenced course's name and category values, and the list will be sorted so that the completed enrollments are placed after the incomplete enrollments.

By defining a corresponding fetch method for this API on the client-side, we can call it in the React component that will render these enrollments. We will look at the implementation of this component in the next section.

The Enrollments component

The `Enrollments` component will be rendered on the Home page, and it will take the list of enrollments as props from the `Home` component. The received list of enrollments will be rendered in this component in order to show the user the courses on which they are enrolled. We will also indicate if an enrolled course in the list has been completed, or is in progress, using representative icons for each state, as shown in the following screenshot:



This view for listing enrollments will be very similar to the `Courses` component, which lists the published courses. In `Enrollments`, instead of courses, the enrollments received from the `Home` component will be iterated over to render each enrollment, as shown in the following code:

mern-classroom/client/enrollment/Enrollments.js:

```
{props.enrollments.map((course, i) => (
  <GridListTile key={i}>
    <Link to={"/learn/" + course._id}>
      <img src={`/api/courses/photo/' + course.course._id}
           alt= {course.course.name} />
    </Link>
    <GridListTileBar
      title={<Link to={"/learn/" + course._id}>{course.course.name}</Link>}
      actionIcon={<div> {course.completed ?
        (<CompletedIcon color="secondary"/>)
        : (<InProgressIcon/>)
      }
      </div>}
    />
```

```
|   </GridListTile>  
| }) }
```

Based on whether the individual enrollment already has a `complete` date value or not, we will render the icons conditionally. This will give the users an idea of which enrolled courses they have completed, and which they are yet to finish.

Now that we have implemented the features to allow students in this application to enroll on courses, complete lessons, and also track their progress, we can also provide enrollment stats about courses by extending on these implementations, as we will see next.

Enrollment stats

Once the instructor publishes a course, and other users in the application start enrolling and completing lessons in the course, we will show the total number of enrollments and course completions as simple enrollment statistics for the course. To implement this feature, in the following sections we will first implement an API that returns the enrollment stats, and then show these stats in the view.

The enrollment stats API

In order to implement a backend API that will query the `Enrollments` collection in the database to calculate the stats for a specific course, we first need to declare a GET route at `'/api/enrollment/stats/:courseId'`, as shown in the following code.

```
mern-classroom/server/routes/enrollment.routes.js
```

```
| router.route('/api/enrollment/stats/:courseId')
|   .get(enrollmentCtrl.enrollmentStats)
```

A GET request at this URL will return a `stats` object containing the total enrollments and total completions for the course, as identified by the `courseId` provided in the URL parameter. This implementation is defined in the `enrollmentStats` controller method, as shown in the following code.

```
mern-classroom/server/controllers/enrollment.controller.js
```

```
| const enrollmentStats = async (req, res) => {
|   try {
|     let stats = {}
|     stats.totalEnrolled = await Enrollment.find({course:req.course._id})
|                               .countDocuments()
|     stats.totalCompleted = await Enrollment.find({course:req.course._id})
|                               .exists('completed', true)
|                               .countDocuments()
|     res.json(stats)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

In this `enrollmentStats` method, we run two queries against the `Enrollments` collection using the course ID that is provided in the request. In the first query, we simply find all the enrollments for the given course, and count these results using MongoDB's `countDocuments()`. In the second query, we find all the enrollments for the given course, and

also check whether the `completed` field exists in these enrollments. Then we finally get the count of these results. These numbers are sent back in the response to the client.

Similar to other API implementations, you will also need to define a corresponding fetch method on the client that will make the GET request to this route. Using this fetch method, we will retrieve and display these stats for each published course, as discussed in the next section.

Displaying enrollment stats for a published course

The enrollment stats can be retrieved from the backend and rendered in the Course view, as shown in the following image:



To retrieve these enrollment stats, we will add a second `useEffect` hook in the `Course` component in order to make a fetch call to the enrollment stats API, as shown in the following code:

`mern-classroom/client/course/Course.js`

```
useEffect(() => {
  const abortController = new AbortController()
  const signal = abortController.signal
  enrollmentStats({courseId: match.params.courseId},
    {t:jwt.token}, signal).then((data) => {
    if (data.error) {
      setValues({...values, error: data.error})
    } else {
      setStats(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [match.params.courseId])
```

This will receive the enrollment stats for the given course and set it to the `stats` variable in state, and we can render it in the view, as shown in the following code:

`mern-classroom/client/course/Course.js`

```
{course.published &&
  (<div> <span> <PeopleIcon /> {stats.totalEnrolled} enrolled </span>
```

```
|           <span> <CompletedIcon/> {stats.totalCompleted} completed  
|     </span>  
</div>  
}
```

With this feature added to the Course component, to any visitor who is browsing through courses in the MERN Classroom application, a published course in the application will look as shown in the following image:

The screenshot shows a course card for "Node.js Fundamentals". At the top, it says "24 enrolled" and "5 completed". Below that, it says "By Jane" and "Software Development". The main image features the Node.js logo with a green hexagon and a small illustration of a person carrying boxes. To the right, there's a description: "Get familiar with the core concepts of Node.js before you start building Node.js applications for the enterprise." An orange "ENROLL" button is at the bottom right. A circular progress bar at the bottom indicates 5 completed lessons.

Lessons

5 lessons

1

Install Node.js

This screenshot of the Course page with the course details, enroll option, and enrollment stats, manages to capture all the features that we have implemented in this chapter in order to make this view possible. A user who signed up to the classroom application became

an educator to create and publish this course with lessons. Then, other users enrolled in the course and completed the course lessons to generate the enrollment stats. We simply extended the MERN skeleton application to add more models, APIs, and React frontend components, which retrieved and then rendered the data received in order to build a complete classroom application.

Summary

In this chapter, we developed a simple online classroom application called MERN Classroom, by extending the skeleton application. We incorporated functionality that allowed users to have multiple roles, including educator and student; to add and publish courses with lessons as an instructor; to enroll on courses and complete lessons as a student; and to keep track of course completion progress and enrollment statistics.

While implementing these features, we practiced how to extend the full-stack component slices that make up the frontend–backend-synced application. We added new features by simply implementing data schemas and models, adding new backend APIs, and integrating these with new React components in the frontend to complete the full-stack slice. By building this application up gradually from smaller units of implementation to complex and combined features, you should now have a better grasp of how to combine the different parts of a MERN-based full-stack application.

In order to learn how to integrate even more complex features, and find solutions to the tricky problems that you may face when developing advanced real-world applications with this stack, we will start building a MERN-based, feature-rich online marketplace application in the next chapter.

Exercising MERN Skills with an Online Marketplace

With more business being conducted over the internet than ever before, the ability to buy and sell in an online marketplace setting has become a core requirement for many web platforms. In this and the next two chapters, we will utilize the MERN stack technologies to develop an online marketplace application complete with features that enable users to buy and sell.

We will build out everything from simple to advanced features for this application, starting in this chapter with a reiteration of the full-stack development lessons learned in previous chapters to set up a base for the marketplace platform. We will be extending the MERN skeleton application with support for seller accounts and shops with products, to incrementally integrate marketplace functionalities such as product search and suggestions. By the end of this chapter, you will have a better grasp of how to extend, integrate, and combine the different aspects of full-stack implementations to add complex features to your applications.

In this chapter, we will start building the online marketplace by covering the following topics:

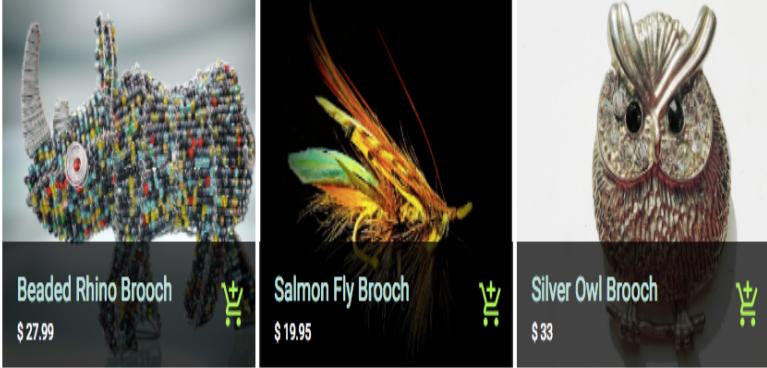
- Introducing the MERN Marketplace app
- Users with seller accounts
- Adding shops to the marketplace
- Adding products to shops
- Searching for products by name and category

Introducing the MERN Marketplace app

The MERN Marketplace application will allow users to become sellers, who can manage multiple shops and add the products they want to sell in each shop. Users who visit MERN Marketplace will be able to search for and browse products they want to buy and add products to their shopping cart to place an order. The resulting marketplace application will look as pictured in the following screenshot:

Select category  Search products

Jewelry  brooch 



Beaded Rhino Brooch \$27.99  

Salmon Fly Brooch \$19.95  

Silver Owl Brooch \$33  

Explore by category

Collectibles 

Tools

Shoes

Sports



Latest Products



Converse All Star Superheroes
Happy Feet
Added on Mon Jan 08 2018
\$140  



Joker Figurine
Toyish
Added on Mon Jan 08 2018
\$51.45  



Darth Vader Figurine
Toyish
Added on Mon Jan 08 2018
\$19.99  



Speed Skipping Rope
Athletico
Added on Mon Jan 08 2018
\$7.55  

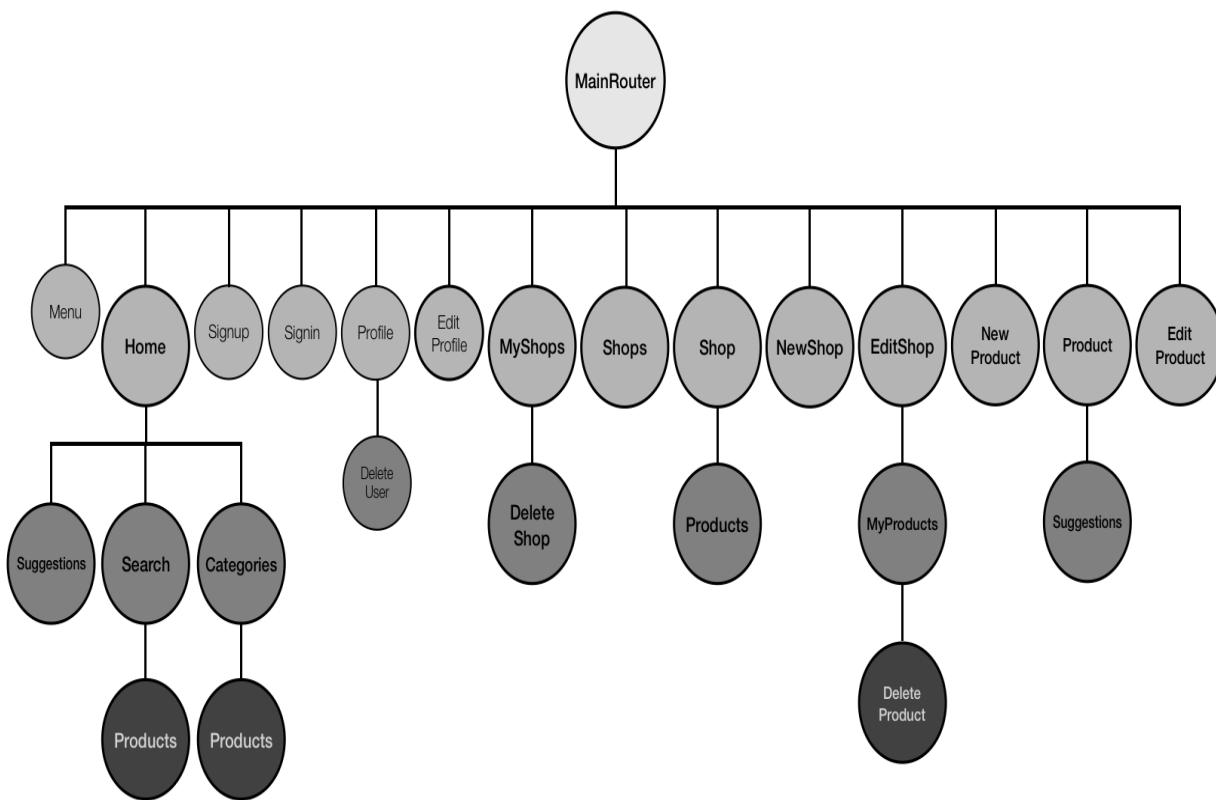
The code for the complete MERN Marketplace application is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter07%20and%2008/mern-marketplace>. The implementations discussed in this and the next chapter can be accessed in the shop-cart-order-pay branch of the repository. You can clone this code and run the application as you go through the code explanations for the rest of this chapter.



In this chapter, we will extend the MERN skeleton to build a simple version of the online marketplace, starting with the following features:

- Users with seller accounts
- Shop management
- Product management
- Product search by name and category

The views needed for these features related to seller accounts, shops, and products will be developed by extending and modifying the existing React components in the MERN skeleton application. The component tree pictured next shows all the custom React components that make up the MERN Marketplace frontend developed in this chapter:

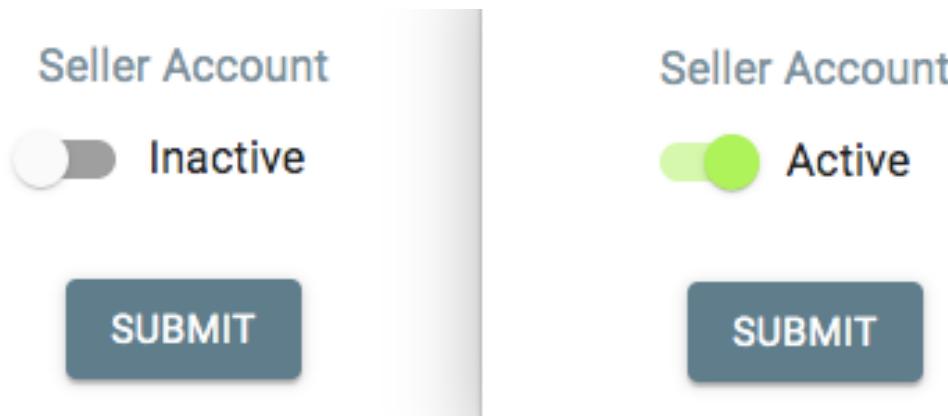


We will add new React components to implement views for managing shops and products as well as browsing and searching for products. We will also modify existing components such as the

EditProfile, Menu, and Home components to develop the skeleton code into a marketplace application as we build out the different features in the rest of the chapter. These marketplace features will depend on the user's ability to update their accounts into seller accounts. In the next section, we will begin building the MERN Marketplace application by updating the existing user implementation to enable seller account features.

Allowing users to be sellers

Any user with an account on the MERN Marketplace application will have the option to update their accounts to seller accounts by making changes to their profiles. We will add this option to convert to a seller account in the *"Edit Profile"* page, as shown in the following screenshot:



A user with an active seller account will be allowed to create and manage their own shops, where they can manage products. Regular users will not have access to a seller dashboard, whereas users with active seller accounts will see a link to their dashboard on the menu as MY SHOPS. The following screenshot shows how the menu looks to a regular user in contrast to a user with an active seller account:



To add this seller account feature, we need to update the user model, the Edit Profile view and add a MY SHOPS link to the menu that will only be visible to sellers, as discussed in the following sections.

Updating the user model

We need to store additional detail about each user to determine whether a user is an active seller or not. We will update the user model that we developed in [Chapter 3, *Building a Backend with MongoDB, Express, and Node*](#), to add a `seller` value that will be set to `false` by default to represent regular users and can additionally be set to `true` to represent users who are also sellers. We will update the existing user schema to add this `seller` field with the following code:

mern-marketplace/server/models/user.model.js:

```
  seller: {  
    type: Boolean,  
    default: false  
  }
```

This `seller` value for each user must be sent to the client with the user details received on successful sign-in, so the view can be rendered accordingly to show information relevant to the seller. We will update the response sent back in the `signin` controller method to add this detail, as highlighted in the following code:

mern-marketplace/server/controllers/auth.controller.js:

```
...  
return res.json({  
  token,  
  user: {  
    _id: user._id,  
    name: user.name,  
    email: user.email,  
    seller: user.seller  
  }  
})  
...
```

Using this `seller` field value, we can render the frontend based on authorizations permitted only to seller accounts. Before rendering

views based on seller authorizations, we first need to implement the option to activate seller account features in the `EditProfile` view, as discussed in the next section.

Updating the Edit Profile view

A signed-in user will see a toggle in the Edit Profile view, allowing them to either activate or deactivate the seller feature. We will update the `EditProfile` component to add a `Material-UI Switch` component in `FormControlLabel`, as shown in the following code:

mern-marketplace/client/user/EditProfile.js:

```
<Typography variant="subtitle1" className={classes.subheading}>
  Seller Account
</Typography>
<FormControlLabel
  control=<Switch
    checked={values.seller}
    onChange={handleCheck}
  />
  label={values.seller? 'Active' : 'Inactive'}
/>
```

Any changes to the switch will be set to the value of the `seller` in state by calling the `handleCheck` method. The `handleCheck` method is implemented as shown here:

mern-marketplace/client/user/EditProfile.js:

```
const handleCheck = (event, checked) => {
  setValues({...values, 'seller': checked})
}
```

When the form to edit profile details is submitted, the `seller` value is also added to details sent in the update to the server, as highlighted in the following code:

mern-marketplace/client/user/EditProfile.js:

```
const clickSubmit = () => {
  const jwt = auth.isAuthenticated()
  const user = {
    name: values.name || undefined,
    email: values.email || undefined,
    password: values.password || undefined,
```

```

    seller: values.seller || undefined
  }
  update({
    userId: match.params.userId
  }, {
    t: jwt.token
  }, user).then((data) => {
  if (data && data.error) {
    setValues({...values, error: data.error})
  } else {
    auth.updateUser(data, ()=>{
      setValues({...values, userId: data._id, redirectToProfile:
true})
    })
  }
})
}

```

On successful update, the user details stored in `sessionStorage` for auth purposes should also be updated. The `auth.updateUser` method is called to do this `sessionStorage` update. The implementation for the `auth.updateUser` method was discussed in *Updating the Edit Profile view* section of [Chapter 6, Building a Web-Based Classroom Application](#).

Once the updated `seller` value is available in the frontend, we can use it to render the interface accordingly. In the next section, we will see how to render the menu differently based on whether the user viewing the application has an active seller account.

Updating the menu

In the frontend of the marketplace application, we can render different options based on whether the user currently browsing the application has an active seller account. In this section, we will add the code to conditionally display a link to *MY SHOPS* on the navigation bar, which will only be visible to the signed-in users who have active seller accounts.

We will update the `Menu` component within the previous code so that it only renders when a user is signed in, as follows:

mern-marketplace/client/core/Menu.js:

```
| {auth.isAuthenticated().user.seller &&
|   (<Link to="/seller/shops">
|     <Button color = {isPartActive(history, "/seller/")}> My Shops </Button>
|   </Link>)
| }
```

This *MY SHOPS* link on the navigation bar will take users with active seller accounts to the seller dashboard view where they can manage the shops they own on the marketplace.

With these updates to the user implementation, it is now possible for users on the marketplace to update their regular accounts to seller accounts, and we can begin incorporating features that will allow these sellers to add shops to the marketplace. We will see how to do this in the following section.

Adding shops to the marketplace

Sellers on MERN Marketplace can create shops and add products to each shop. To store the shop data and enable shop management, we will implement a Mongoose Schema for shops, backend APIs to access and modify the shop data, and frontend views for both the shop owner and buyers browsing through the marketplace.

In the following sections, we will build out the shop module in the application by first defining the shop model for storing shop data in the database, then implementing the backend APIs and frontend views for the shop-related features including creating new shops, listing all shops, listing shops by owner, displaying a single shop, editing shops, and deleting shops from the application.

Defining a Shop model

We will implement a Mongoose model to define a Shop model for storing the details of each shop. This model will be defined in `server/models/shop.model.js`, and the implementation will be similar to other Mongoose model implementations covered in previous chapters, like the Course model defined in [Chapter 6, Building a Web-Based Classroom Application](#). The Shop schema in this model will have simple fields to store shop details, along with a logo image, and a reference to the user who owns the shop. The code blocks defining the shop fields are given in the following list with explanations:

- **Shop name and description:** The `name` and `description` fields will be string types, with `name` as a required field:

```
name: {
  type: String,
  trim: true,
  required: 'Name is required'
},
description: {
  type: String,
  trim: true
},
```

- **Shop logo image:** The `image` field will store the logo image file uploaded by the user as data in the MongoDB database:

```
image: {
  data: Buffer,
  contentType: String
},
```

- **Shop owner:** The `owner` field will reference the user who creates the shop:

```
owner: {
  type: mongoose.Schema.ObjectId,
  ref: 'User'
}
```

- ***Created at* and *updated at* times:** The `created` and `updated` fields will be `Date` types, with `created` generated when a new shop is added, and `updated` changed when any shop details are modified:

```
    updated: Date,  
    created: {  
        type: Date,  
        default: Date.now  
    },
```

The fields added in this schema definition will enable us to implement the shop-related features in MERN Marketplace. In the next section, we will start developing these features by implementing the full-stack slice that will allow sellers to create new shops.

Creating a new shop

In MERN Marketplace, a user who is signed in and has an active seller account will be able to create new shops. To implement this feature, in the following sections we will add a create shop API in the backend, along with a way to fetch this API in the frontend, and a create new shop form view that takes user input for shop fields.

The create shop API

For the implementation of the create shop API that will allow creating new shops in the database, we will first add a `POST` route, as shown in the following code:

```
mern-marketplace/server/routes/shop.routes.js:
```

```
| router.route('/api/shops/by/:userId')
|   .post(authCtrl.requireSignin, authCtrl.hasAuthorization,
|         userCtrl.isSeller, shopCtrl.create)
```

A `POST` request to this route at `/api/shops/by/:userId` will first ensure the requesting user is signed in and is also the authorized owner, in other words, it is the same user associated with the `:userId` specified in the route param.

To process the `:userId` param and retrieve the associated user from the database, we will utilize the `userByID` method in the user controller. We will add the following to the `Shop` routes in `shop.routes.js`, so the user is available in the `request` object as `profile`:

```
mern-marketplace/server/routes/shop.routes.js:
```

```
| router.param('userId', userCtrl.userByID)
```

The `shop.routes.js` file containing the shop routes will be very similar to the `user.routes` file. To load these new shop routes in the Express app, we need to mount the shop routes in `express.js` as shown in the following code, as we did for the auth and user routes:

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

```
| app.use('/', shopRoutes)
```

The request to the create shop route will also verify that the current user is a seller before creating a new shop with the shop data

passed in the request. We will update the user controller to add the `isSeller` method, which will ensure that the current user is actually a seller. The `isSeller` method is defined as follows:

mern-marketplace/server/controllers/user.controller.js:

```
const isSeller = (req, res, next) => {
  const isSeller = req.profile && req.profile.seller
  if (!isSeller) {
    return res.status('403').json({
      error: "User is not a seller"
    })
  }
  next()
}
```

The `create` method in the shop controller, which is invoked after a seller is verified, uses the `formidable` node module to parse the multipart request that may contain an image file uploaded by the user for the shop logo. If there is a file, `formidable` will store it temporarily in the filesystem, and we will read it using the `fs` module to retrieve the filetype and data to store it in the `image` field in the shop document. The `create` controller method will look as shown in the following code block:

mern-marketplace/server/controllers/shop.controller.js:

```
const create = (req, res, next) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Image could not be uploaded"
      })
    }
    let shop = new Shop(fields)
    shop.owner= req.profile
    if(files.image){
      shop.image.data = fs.readFileSync(files.image.path)
      shop.image.contentType = files.image.type
    }
    shop.save((err, result) => {
      if (err) {
        return res.status(400).json({
          error: errorHandler.getErrorMessage(err)
        })
      }
    })
  })
}
```

```
    res.status(200).json(result)
  })
}
```

The logo image file for the shop is uploaded by the user and stored in MongoDB as data. Then, in order to be shown in the views, it is retrieved from the database as an image file at a separate `GET` API.

The `GET` API is set up as an Express route at `/api/shops/logo/:shopId`, which gets the image data from MongoDB and sends it as a file in the response. The implementation steps for file upload, storage, and retrieval are outlined in detail in the *Upload profile photo* section of [Chapter 5, Starting with a Simple Social Media Application](#).

This create shop API endpoint can now be used in the frontend to make a `POST` request. Next, we will add a `fetch` method on the client side to make this request from the application's client interface.

Fetching the create API in the view

In the frontend, to make a request to this create API, we will set up a `fetch` method on the client side to make a `POST` request to the API route and pass it the multipart form data containing details of the new shop. This `fetch` method will be defined as follows:

`mern-marketplace/client/shop/api-shop.js`:

```
const create = (params, credentials, shop) => {
  return fetch('/api/shops/by/' + params.userId, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Authorization': 'Bearer ' + credentials.t
    },
    body: shop
  })
  .then((response) => {
    return response.json()
  }).catch((err) => console.log(err))
}
```

We will use this method in the create new shop form view, implemented in the next section, to send the user-entered shop details to the backend.

The NewShop component

Sellers in the marketplace application will interact with a form view to enter details of a new shop and create the new shop. We will render this form in the `NewShop` component, which will allow a seller to create a shop by entering a name and description, and uploading a logo image file from their local filesystem, as pictured in the following screenshot:

The screenshot shows a user interface for creating a new shop. At the top, the title "New Shop" is displayed. Below it is a green button labeled "UPLOAD LOGO" with an upward arrow icon. A horizontal line separates this from the input fields. The first field is labeled "Name" and contains a single character, "A". The second field is labeled "Description" and is empty. At the bottom, there are two buttons: a dark blue "SUBMIT" button on the left and a light gray "CANCEL" button on the right.

We will implement this form in a React component named `NewShop`. For the view, we will first add the file upload elements using a Material-UI button and an HTML5 file input element, as shown in the following code:

mern-marketplace/client/shop/NewShop.js:

```
<input accept="image/*" onChange={handleChange('image')}  
      id="icon-button-file"  
      style={{display:'none'}} type="file" />  
<label htmlFor="icon-button-file">  
  <Button variant="contained" color="secondary" component="span">  
    Upload Logo <FileUpload/>  
  </Button>  
</label>  
<span>{values.image ? values.image.name : ''}</span>
```

Then, we add the name and description form fields with the `TextField` components, as shown next:

mern-marketplace/client/shop/NewShop.js:

```
<TextField
  id="name"
  label="Name"
  value={values.name}
  onChange={handleChange('name')} /> <br/>
<TextField
  id="multiline-flexible"
  label="Description"
  multiline rows="2"
  value={values.description}
  onChange={handleChange('description')} />
```

These form field changes will be tracked with the `handleChange` method when a user interacts with the input fields to enter values. The `handleChange` function will be defined as shown in the following code:

mern-marketplace/client/shop/NewShop.js:

```
const handleChange = name => event => {
  const value = name === 'image'
    ? event.target.files[0]
    : event.target.value
  setValues({ ...values, [name]: value })
}
```

The `handleChange` method updates the state with the new values, including the name of the image file, should one be uploaded by the user.

Finally, you can complete this form view by adding a submit button that when clicked, should send the form data to the server. We will define a `clickSubmit` method, as shown next, which will be called when the submit button is clicked by the user:

mern-marketplace/client/shop/NewShop.js:

```
const clickSubmit = () => {
  const jwt = auth.isAuthenticated()
  let shopData = new FormData()
  values.name && shopData.append('name', values.name)
```

```

    values.description && shopData.append('description',
values.description)
    values.image && shopData.append('image', values.image)
    create({
      userId: jwt.user._id
    }, {
      t: jwt.token
    }, shopData).then((data) => {
      if (data.error) {
        setValues({...values, error: data.error})
      } else {
        setValues({...values, error: '', redirect: true})
      }
    })
}

```

This `clickSubmit` function will take the input values and populate `shopData`, which is a `FormData` object that ensures the data is stored in the correct format needed for the `multipart/form-data` encoding type.

Then the `create` fetch method is called to create the new shop in the backend with this form data. On successful shop creation, the user is redirected back to the `MyShops` view with the following code:

mern-marketplace/client/shop/NewShop.js:

```

if (values.redirect) {
  return (<Redirect to={'/seller/shops'} />)
}

```

The `NewShop` component can only be viewed by a signed-in user who is also a seller. So we will add a `PrivateRoute` in the `MainRouter` component, as shown in the following code block, that will render this form only for authenticated users at `/seller/shop/new`:

mern-marketplace/client/MainRouter.js:

```
<PrivateRoute path="/seller/shop/new" component={NewShop} />
```

This link can be added to any of the view components that may be accessed by the seller, for example in a view where a seller manages their shops in the marketplace. Now that it is possible to add new shops in the marketplace, in the next section we will discuss the implementations to fetch and list these shops from the database in the backend to the application views in the frontend.

Listing shops

In MERN Marketplace, regular users will be able to browse through a list of all the shops on the platform, and each shop owner will manage a list of their own shops. In the following sections, we will implement the full-stack slices for retrieving and displaying two different lists of shops – a list of all the shops, and the list of shops owned by a specific user.

Listing all shops

Any user browsing through the marketplace will be able to see a list of all the shops on the marketplace. In order to implement this feature, we will have to query the `shops` collection to retrieve all the shops in the database and display it in a view to the end user. We achieve this by adding a full-stack slice with the following:

- A backend API to retrieve the list of shops
- A `fetch` method in the frontend to make a request to the API
- A React component to display the list of shops

The shops list API

In the backend, we will define an API to retrieve all the shops from the database, so the shops in the marketplace can be listed in the frontend. This API will accept a request from the client to query the `shops` collection and return the resulting shop documents in the response. First, we will add a route to retrieve all the shops stored in the database when the server receives a `GET` request at `'/api/shops'`. This route is declared as shown in the following code:

```
mern-marketplace/server/routes/shop.routes.js
```

```
| router.route('/api/shops')
|   .get(shopCtrl.list)
```

A `GET` request received at this route will invoke the `list` controller method, which will query the `shops` collection in the database to return all the shops. The `list` method is defined as follows:

```
mern-marketplace/server/controllers/shop.controller.js:
```

```
const list = async (req, res) => {
  try {
    let shops = await Shop.find()
    res.json(shops)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This method will return all the shops in the database in response to the requesting client. Next, we will see how to make a request to this shop list API from the client side.

Fetch all shops for the view

In order to use the shop list API in the frontend, we will define a `fetch` method that can be used by React components to load this list of shops. The `list` method on the client side will use `fetch` to make a `GET` request to the API, as shown in the following code:

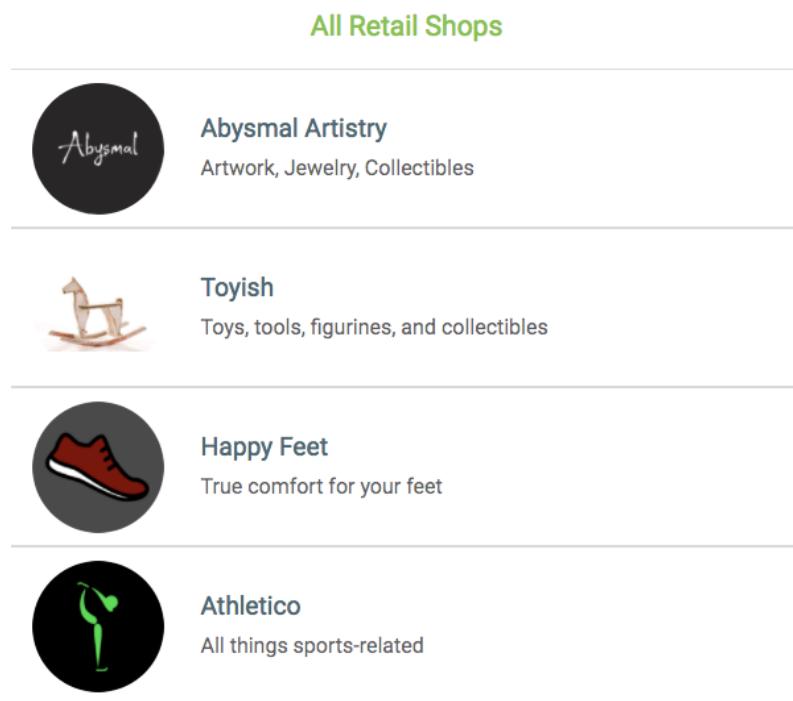
`mern-marketplace/client/shop/api-shop.js`:

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

As we will see in the next section, this `list` method can be used in the React component to display the list of shops.

The Shops component

In the `Shops` component, we will render the list of shops in a Material-UI `List`, after fetching the data from the server and setting the data in a state to be displayed as shown in the following screenshot:



To implement this component, we first need to fetch and render the list of shops. We will make the fetch API call in the `useEffect` hook, and set the received `shops` array in the state, as shown here:

mern-marketplace/client/shop/Shops.js:

```
export default function Shops() {
  const [shops, setShops] = useState([])

  useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal
    list(signal).then((data) => {
      if (!data.error) {
        setShops(data)
      }
    })
  })
}
```

```

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

```

In the `shops` component view, this retrieved `shops` array is iterated over using `map`, with each shop's data rendered in the view in a Material-UI `ListItem`, and each `ListItem` is also linked to the individual shop's view, as shown in the following code:

`mern-marketplace/client/shop/Shops.js`:

```

{shops.map((shop, i) => {
    return <Link to={"/shops/" + shop._id} key={i}>
        <Divider/>
        <ListItem button>
            <ListItemIcon>
                <Avatar src={'/api/shops/logo/' + shop._id + "?" + new
Date().getTime()} />
            </ListItemIcon>
            <div className={classes.details}>
                <Typography type="headline"
                    component="h2" color="primary">
                    {shop.name}
                </Typography>
                <Typography type="subheading" component="h4">
                    {shop.description}
                </Typography>
            </div>
        </ListItem>
        <Divider/>
    </Link>
})}

```

The `shops` component will be accessed by the end user at `/shops/all`, which is set up with React Router and declared in `MainRouter.js` as follows:

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

```

| <Route path="/shops/all" component={Shops}/>

```

Adding this link to any view in the application will redirect the user to a view displaying all the shops in the marketplace. Next, we will similarly implement the feature to list the shops owned by a specific user.

Listing shops by owner

Authorized sellers on the marketplace will see a list of the shops they created, which they can manage by editing or deleting any shop on the list. In order to implement this feature, we will have to query the shops' collection to retrieve all the shops with the same owner and display it only to the authorized owner of the shops. We achieve this by adding a full-stack slice with the following:

- A backend API that ensures the requesting user is authorized and retrieves the relevant list of shops
- A `fetch` method in the frontend to make a request to this API
- A React component to display the list of shops to the authorized user

The shops by owner API

We will implement an API in the backend to return the list of shops of a specific owner, so it can be rendered in the frontend for the end user. We will start by adding a route in the backend to retrieve all the shops created by a given user when the server receives a `GET` request at `/api/shops/by/:userId`. This route is declared as shown in the following code:

mern-marketplace/server/routes/shop.routes.js:

```
| router.route('/api/shops/by/:userId')
|   .get(authCtrl.requireSignin, authCtrl.hasAuthorization,
|     shopCtrl.listByOwner)
```

A `GET` request to this route will first ensure the requesting user is signed in and is also the authorized owner, before invoking the `listByOwner` controller method in `shop.controller.js`. This method will query the `Shop` collection in the database to get the matching shops. This `listByOwner` method is defined as follows:

mern-marketplace/server/controllers/shop.controller.js:

```
| const listByOwner = async (req, res) => {
|   try {
|     let shops = await Shop.find({owner:
|       req.profile._id}).populate('owner', '_id name')
|     res.json(shops)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

In the query to the `Shop` collection, we find all the shops where the `owner` field matches the user-specified with the `userId` param, then populate the referenced user's ID and name in the `owner` field, and return the resulting shops in an array in the response to the

client. Next, we will see how to make a request to this API from the client side.

Fetch all shops owned by a user for the view

In the frontend, to fetch the shops for a specific user using this list by owner API, we will add a `fetch` method that takes the signed-in user's credentials to make a `GET` request to the API route with the specific user ID passed in the URL. This `fetch` method is defined as shown in the following code:

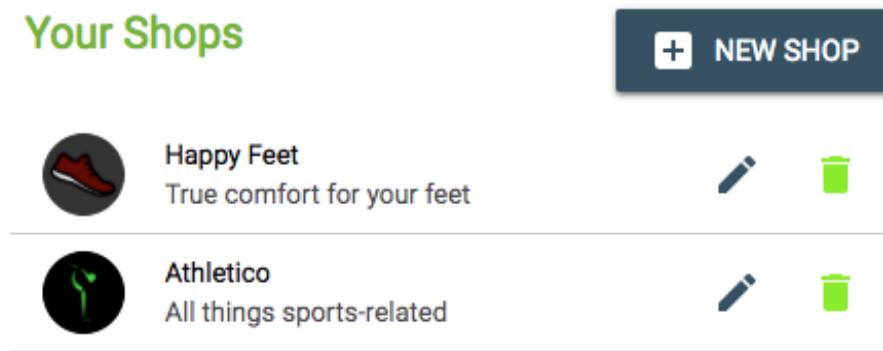
mern-marketplace/client/shop/api-shop.js:

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

The shops returned in the response from the server using this method can be rendered in a React component to display the shops to the authorized user, as discussed in the next section.

The MyShops component

The `MyShops` component is similar to the `shops` component. It fetches the list of shops owned by the current user, and renders each shop in a `ListItem`, as pictured in the following screenshot:



Additionally, each shop has an `edit` and a `delete` option, unlike the list of items in `shops`. The implementation for the `MyShops` component is the same as `shops`, except for these edit and delete buttons, which are added as follows:

mern-marketplace/client/shop/MyShops.js:

```
<ListItemSecondaryAction>
  <Link to={"/seller/shop/edit/" + shop._id}>
    <IconButton aria-label="Edit" color="primary">
      <Edit/>
    </IconButton>
  </Link>
  <DeleteShop shop={shop} onRemove={removeShop}/>
</ListItemSecondaryAction>
```

The `Edit` button links to an "*Edit Shop*" view, whereas the `DeleteShop` component, which is discussed later in the chapter, handles the delete action. The `DeleteShop` component updates the list by calling the `removeShop` method passed from `MyShops`. This `removeShop` method allows us

to update the state with the modified list of shops for the current user and is defined in the `MyShops` component, as shown here:

mern-marketplace/client/shop/MyShops.js:

```
| const removeShop = (shop) => {
|   const updatedShops = [...shops]
|   const index = updatedShops.indexOf(shop)
|   updatedShops.splice(index, 1)
|   setShops(updatedShops)
| }
```

The `MyShops` component can only be viewed by a signed-in user who is also a seller. So we will add a `PrivateRoute` in the `MainRouter` component, which will render this component only for authenticated users at `/seller/shops`, as shown in the following code:

mern-marketplace/client/MainRouter.js:

```
| <PrivateRoute path="/seller/shops" component={MyShops} />
```

In the marketplace application, we add this link to the navigation menu to redirect a signed-in seller to the view where they can manage the shops they own by editing or deleting a shop. Before adding the ability to edit or delete shops, next we will look into how to retrieve a single shop from the backend and display it to the end user.

Displaying a shop

Any users visiting MERN Marketplace will be able to browse through each individual shop. In the following sections, we will implement the individual shop view by adding a read shop API to the backend, a way to call this API from the frontend, and the React component that will display the shop details in the view.

The read a shop API

In order to implement the read shop API in the backend, we will start by adding a `GET` route that queries the `Shop` collection with an ID and returns the shop in the response. The route is declared along with a route parameter handler, as shown in the following code:

`mern-marketplace/server/routes/shop.routes.js`:

```
router.route('/api/shop/:shopId')
  .get(shopCtrl.read)
router.param('shopId', shopCtrl.shopByID)
```

The `:shopId` param in the route URL will invoke the `shopByID` controller method, which is similar to the `userByID` controller method. It retrieves the shop from the database and attaches it to the request object to be used in the `next` method. The `shopByID` method is defined as follows:

`mern-marketplace/server/controllers/shop.controller.js`:

```
const shopByID = async (req, res, next, id) => {
  try {
    let shop = await Shop.findById(id).populate('owner', '_id
name').exec()
    if (!shop)
      return res.status('400').json({
        error: "Shop not found"
      })
    req.shop = shop
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve shop"
    })
  }
}
```

The shop object queried from the database will also contain the name and ID details of the owner, as we specified in the `populate()` method. The `read` controller method then returns this shop object in

response to the client. The `read` controller method is defined as shown in the following code:

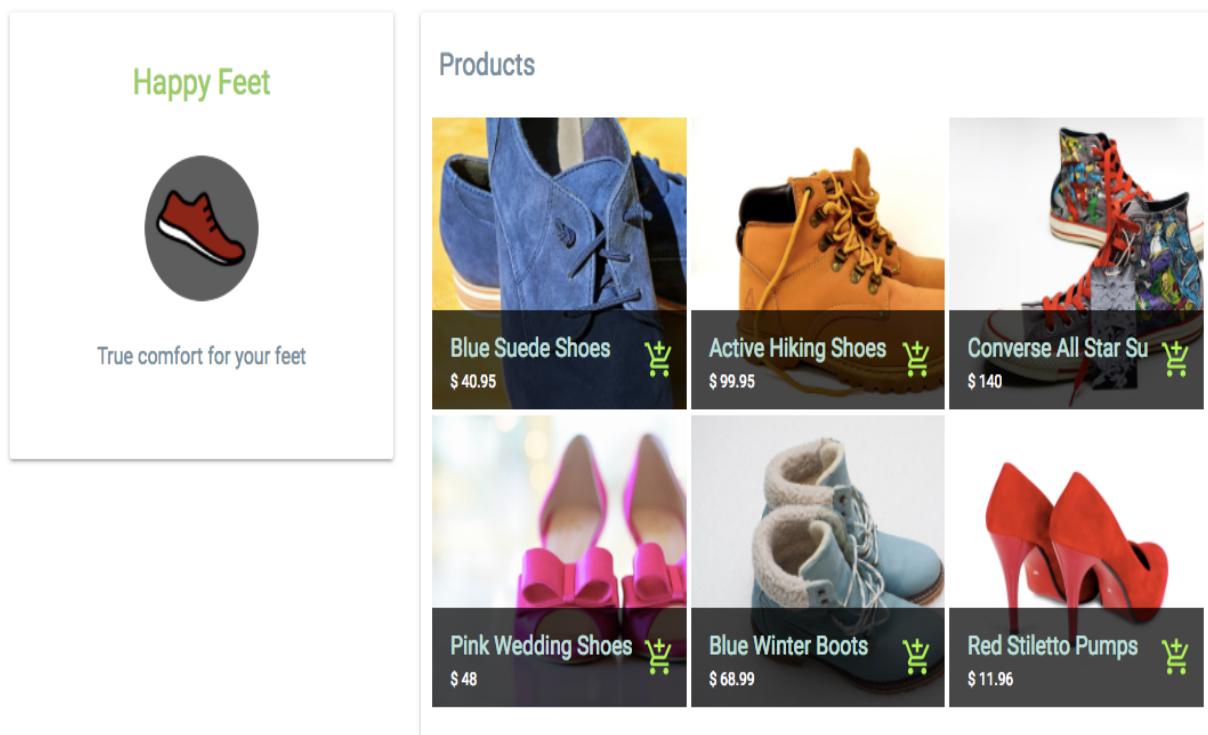
`mern-marketplace/server/controllers/shop.controller.js`:

```
| const read = (req, res) => {  
|   req.shop.image = undefined  
|   return res.json(req.shop)  
| }
```

We are removing the `image` field before sending the response since images will be retrieved as files in separate routes. With this API ready in the backend, you can now add the implementation to call it in the frontend by adding a `fetch` method in `api-shop.js`, similar to other `fetch` methods already added for other API implementations. We will use the `fetch` method to call the `read` shop API in the React component that will render the shop details, as discussed in the next section.

The Shop component

The `Shop` component will render the shop details and also a list of products in the specified shop using a product list component, which will be discussed in the *Products* section. The completed single `Shop` view will look as pictured in the following screenshot:



To implement this `Shop` component, we will first retrieve the shop details with a fetch call to the read API in a `useEffect` hook, and set the received values to state, as shown in the following code:

mern-marketplace/client/shop/Shop.js:

```
export default function Shop({match}) {
  const [shop, setShop] = useState('')
  const [error, setError] = useState('')

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

```

    const signal = abortController.signal

    read({
      shopId: match.params.shopId
    }, signal).then((data) => {
      if (data.error) {
        setError(data.error)
      } else {
        setShop(data)
      }
    })
    return function cleanup() {
      abortController.abort()
    }
  }, [match.params.shopId])
...
}

```

This `useEffect` hook will only run when the `shopId` changes in the route params.

The retrieved shop data is set to state and rendered in the view to display the shop's name, logo, and description with the following code:

mern-marketplace/client/shop/Shop.js:

```

<CardContent>
  <Typography type="headline" component="h2">
    {shop.name}
  </Typography><br/>
  <Avatar src={logoUrl}/><br/>
  <Typography type="subheading" component="h2">
    {shop.description}
  </Typography><br/>
</CardContent>

```

The `logoUrl` points to the route from where the logo image can be retrieved from the database (if the image exists), and it's defined as follows:

mern-marketplace/client/shop/Shop.js:

```

const logoUrl = shop._id
  ? `/api/shops/logo/${shop._id}?${new Date().getTime()}`
  : '/api/shops/defaultphoto'

```

The `Shop` component will be accessed in the browser at the `/shops/:shopId` route, which is defined in `MainRouter` as follows:

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

```
| <Route path="/shops/:shopId" component={Shop}/>
```

This route can be used in any component to link to a specific shop, and this link will take the user to the corresponding `Shop` view with the shop details loaded. In the next section, we will add the ability to allow the shop owners to edit these shop details.

Editing a shop

Authorized sellers in the application will be able to update the shops they have already added to the marketplace. To implement this capability, we will have to create a backend API that allows the update operation on a given shop after ensuring that the requesting user is authenticated and authorized. Then this updated API needs to be called from the frontend with the changed details of the shop. In the following sections, we will build this backend API and the React component to allow sellers to make changes to their shops.

The edit shop API

In the backend, we will need an API that allows updating an existing shop in the database if the user making the request is the authorized seller of the given shop. We will first declare the PUT route that accepts the update request from the client as follows:

mern-marketplace/server/routes/shop.routes.js:

```
| router.route('/api/shops/:shopId')
|   .put(authCtrl.requireSignin, shopCtrl.isOwner, shopCtrl.update)
```

A PUT request received at the `/api/shops/:shopId` route first checks if the signed-in user is the owner of the shop associated with the `shopId` provided in the URL using the `isOwner` controller method, which is defined as follows:

mern-marketplace/server/controllers/shop.controller.js:

```
| const isOwner = (req, res, next) => {
|   const isOwner = req.shop && req.auth
|                           && req.shop.owner._id == req.auth._id
|   if(!isOwner){
|     return res.status('403').json({
|       error: "User is not authorized"
|     })
|   }
|   next()
| }
```

In this method, if the user is found to be authorized, the `update` controller is invoked with a call to `next()`.

The `update` controller method will use the `formidable` and `fs` modules as in the `create` controller method discussed earlier, to parse the form data and update the existing shop in the database. The `update` method in the shop controllers is defined as shown in the following code:

mern-marketplace/server/controllers/shop.controller.js:

```

const update = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Photo could not be uploaded"
      })
    }
    let shop = req.shop
    shop = extend(shop, fields)
    shop.updated = Date.now()
    if(files.image) {
      shop.image.data = fs.readFileSync(files.image.path)
      shop.image.contentType = files.image.type
    }
    try {
      let result = await shop.save()
      res.json(result)
    } catch (err) {
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}

```

To use this update API in the frontend, you will need to define a `fetch` method that takes the shop ID, user auth credentials, and the updated shop details to make the fetch call to this update shop API, as we have done for other API implementations including the create shop API in the *Creating a new shop* section.

We now have a shop update API that can be used in the frontend to update the details of a shop. We will use this in the `EditShop` component, which is discussed next.

The EditShop component

The `EditShop` component will show a form similar to the create new shop form, pre-populated with the existing shop details. This component will also show a list of the products in this shop, to be discussed in the *Products* section. The completed Edit Shop view is pictured in the following screenshot:

Edit Shop



CHANGE LOGO

Name
Athletico

Description
All things sports-related

Owner: Thomas Brogan

Products

	Blue Boxing Gloves	<input type="button" value="edit"/> <input type="button" value="trash"/>
	Speed Skipping Rope	<input type="button" value="edit"/> <input type="button" value="trash"/>
	OTG 2kg Dumbbell	<input type="button" value="edit"/> <input type="button" value="trash"/>
	Hand Grip Strengthener	<input type="button" value="edit"/> <input type="button" value="trash"/>

The form part of this view for editing shop details is similar to the form in the `NewShop` component, with the same form fields and a `formData` object that holds the multipart form data to be sent with the `update` `fetch`

method. In contrast to the `NewShop` component, in this component, we will need to utilize the read shop API to fetch the given shop's details in an `useEffect` hook and pre-populate the form fields. You can combine the implementations discussed for the `NewShop` component and `Shop` component to complete the `EditShop` component.

The `EditShop` component will only be accessible by authorized shop owners. So we will add a `PrivateRoute` in the `MainRouter` component as shown next, which will render this component only for authenticated users at `/seller/shop/edit/:shopId`:

mern-marketplace/client/MainRouter.js:

```
| <PrivateRoute path="/seller/shop/edit/:shopId" component={EditShop} />
```

This link is added with an edit icon for each shop in the `MyShops` component, allowing a seller to access the edit page for each of their shops. In the `MyShops` view, sellers are also able to delete their shops, as implemented in the next section.

Deleting a shop

As a part of managing the shops they own, authorized sellers will have the option to delete any of their own shops. In order to allow a seller to remove a shop from the marketplace, in the following sections, first we will define a backend API for shop deletion from the database, and then implement a React component that makes use of this API when the user interacts with the frontend to perform this deletion.

The delete shop API

In order to delete a shop from the database, we will implement a delete shop API in the backend, which will accept a DELETE request from the client at `/api/shops/:shopId`. We will add the `DELETE` route for this API as shown in the following code, which will allow an authorized seller to delete one of their own shops:

mern-marketplace/server/routes/shop.routes.js:

```
| router.route('/api/shops/:shopId')
|   .delete(authCtrl.requireSignin, shopCtrl.isOwner, shopCtrl.remove)
```

When a `DELETE` request is received at this route, if the `isOwner` method confirms that the signed-in user is the owner of the shop, then the `remove` controller method deletes the shop specified by the `shopId` in the param. The `remove` method is defined as follows:

mern-marketplace/server/controllers/shop.controller.js:

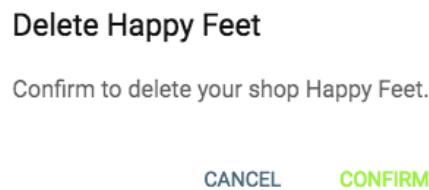
```
| const remove = async (req, res) => {
|   try {
|     let shop = req.shop
|     let deletedShop = shop.remove()
|     res.json(deletedShop)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

This `remove` method simply deletes the shop document that corresponds to the provided ID from the `shops` collection in the database. To access this backend API in the frontend, you will also need a `fetch` method with this route, similar to other API implementations. The `fetch` method will need to take the shop ID and current user's auth credentials then call the delete shop API with these values.

The `fetch` method will be used when the user performs the delete operation by clicking a button in the frontend interface. In the next section, we will discuss a React component called `DeleteShop`, where this delete shop action will be performed by the user.

The DeleteShop component

The `DeleteShop` component is added to the `MyShops` component for each shop in the list. It takes the `shop` object and a `onRemove` method as props from `MyShops`. This component is basically a button that, when clicked, opens a `Dialog` component asking the user to confirm the delete action, as shown in the following screenshot:



The implementation of the `DeleteShop` component is similar to the `DeleteUser` component discussed in [Chapter 4, Adding a React Frontend to Complete MERN](#). Instead of a user ID, the `DeleteShop` component will take the `shop` object and `onRemove` function definition from the `MyShops` component as props when it is added to `MyShops`, as shown in the following code:

`mern-marketplace/client/shop/MyShops.js`:

```
| <DeleteShop shop={shop} onRemove={removeShop} />
```

With this implementation, authorized sellers will be able to remove a shop that they own from the marketplace.

We have implemented the shop module for the marketplace by first defining the Shop model for storing shop data, and then integrating the backend APIs and frontend views to be able to perform CRUD operations on shops from the application. These shop features, with the ability to create new shops, display a shop, edit and delete shops, will allow both buyers and sellers to interact with the shops in the marketplace. The shops will also have products, discussed next,

which the owners will manage and the buyers will be able to browse through, with an option to add products to their cart.

Adding products to shops

Products are the most crucial aspect of a marketplace application. In the MERN Marketplace, sellers can manage products in their shops, and visitors can search for and browse products. While we will implement the features to allow authorized sellers to add, modify, and delete products from their shops, we will also incorporate features to list products in ways that are meaningful to the end user. In the application, we will retrieve and display products by a specific shop, products related to a given product, and the latest products added to the marketplace. In the following sections, we will build out the product module incorporating these features by first defining a product model for storing product data in the database, and then implementing the backend APIs and frontend views for the product-related features including adding new products to a shop, rendering different lists of products, displaying a single product, editing products, and deleting products.

Defining a Product model

Products will be stored in a product collection in the database. To implement this, we will add a Mongoose model to define a Product model for storing the details of each product. This model will be defined in `server/models/product.model.js`, and the implementation will be similar to other Mongoose model implementations covered in previous chapters, like the Course model defined in [Chapter 6, Building a Web-Based Classroom Application](#).

For MERN Marketplace, we will keep the product schema simple with support for fields such as `product name, description, image, category, quantity, price, created at, updated at`, and a reference to the shop. The code defining the product fields in the product schema are given in the following list, along with explanations:

- **Product name and description:** The `name` and `description` fields will be `String` types, with `name` as a `required` field:

```
name: {
  type: String,
  trim: true,
  required: 'Name is required'
},
description: {
  type: String,
  trim: true
},
```

- **Product image:** The `image` field will store an image file to be uploaded by the user as data in the MongoDB database:

```
image: {
  data: Buffer,
  contentType: String
},
```

- **Product category:** The `category` value will allow grouping products of the same type together:

```
    category: {
      type: String
    },
```

- **Product quantity:** The `quantity` field will represent the amount available for selling in the shop:

```
    quantity: {
      type: Number,
      required: "Quantity is required"
    },
```

- **Product price:** The `price` field will hold the unit price this product will cost the buyer:

```
    price: {
      type: Number,
      required: "Price is required"
    },
```

- **Product shop:** The `shop` field will reference the shop to which the product was added:

```
    shop: {
      type: mongoose.Schema.ObjectId,
      ref: 'Shop'
    }
```

- **Created and updated at times:** The `created` and `updated` fields will be `Date` types, with `created` generated when a new product is added, and the `updated` time changed when the product's details are modified:

```
    updated: Date,
    created: {
      type: Date,
      default: Date.now
    },
```

The fields in this schema definition will enable us to implement the product-related features in MERN Marketplace. To begin the implementation of these features, in the next section, we will implement the full-stack slice that will allow sellers to add new products to their existing shops in the marketplace.

Creating a new product

Sellers in MERN Marketplace will be able to add new products to the shops they own on the platform. To implement this feature, in the following sections we will add a create product API in the backend, along with a way to fetch this API in the frontend, and a create new product form view that takes user input for product fields.

The create product API

We will add a backend API that will let authorized shop owners save new products to the database with a `POST` request from the client side. In order to implement this create product API in the backend, we will first add a route at `/api/products/by/:shopId`, which accepts a `POST` request containing the product data. Sending a request to this route will create a new product associated with the shop identified by the `:shopId` param. This create product API route is declared as shown in the following code:

```
mern-marketplace/server/routes/product.routes.js:
```

```
| router.route('/api/products/by/:shopId')
|   .post(authCtrl.requireSignin, shopCtrl.isOwner, productCtrl.create)
| router.param('shopId', shopCtrl.shopByID)
```

The `product.routes.js` file containing this route declaration will be very similar to the `shop.routes.js` file, and to load these new routes in the Express app, we need to mount the product routes in `express.js`, as shown next:

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

```
| app.use('/', productRoutes)
```

The code to handle a request to the create product API route will first check that the current user is the owner of the shop to which the new product will be added before creating the new product in the database. This API utilizes the `shopByID` and `isOwner` methods from the shop controller to process the `:shopId` param and to verify that the current user is the shop owner, before invoking the `create` controller method. The `create` method is defined as follows:

```
mern-marketplace/server/controllers/product.controller.js:
```

```

const create = (req, res, next) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        message: "Image could not be uploaded"
      })
    }
    let product = new Product(fields)
    product.shop= req.shop
    if(files.image){
      product.image.data = fs.readFileSync(files.image.path)
      product.image.contentType = files.image.type
    }
    try {
      let result = await product.save()
      res.json(result)
    } catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}

```

This `create` method, in the product controller, uses the `formidable` node module to parse the multipart request that may contain an image file uploaded by the user along with the product fields. The parsed data is then saved to the Products collection as a new product.

In the frontend, to use this create product API, you will also need to set up a `fetch` method in `client/product/api-product.js` to make a `POST` request to the create API by passing the multipart form data from the view. This `fetch` method can then be utilized in the React component, which takes the product details from the user and sends the request to create a new product. The implementation of this form-based React component to create new products is discussed in the next section.

The NewProduct component

An authorized seller who already has a shop created in the marketplace will see a form view for adding new products to the shop. We will implement this form view in a React component named `NewProduct`. The `NewProduct` component will be similar to the `NewShop` component. It will contain a form that allows a seller to create a product by entering a name, description, category, quantity, and price, and to upload a product image file from their local filesystem, as pictured in the following screenshot:

New Product

UPLOAD PHOTO 

Name

Description

Category

Quantity

Price

SUBMIT

CANCEL

The `NewProduct` component can be implemented almost exactly the same as the `NewShop` component, with the exception of retrieving the shop ID from the frontend route URL that will render the `NewProduct`

component. This component will load at a route that is associated with a specific shop, so only signed-in users who are sellers can add a product to a shop they own. To define this route, we add a `PrivateRoute` in the `MainRouter` component as shown next, which will render this form only for authorized users at the URL `/seller/:shopId/products/new`:

mern-marketplace/client/MainRouter.js:

```
| <PrivateRoute path="/seller/:shopId/products/new" component={NewProduct}/>
```

Adding this link for a specific shop to any view in the frontend will render the `NewProduct` component for the signed-in user. In this view, the users will be able to fill out the new product details in the form and then save the product to the database in the backend, only if they are the authorized owner of the given shop. Next, we will look into the implementations for retrieving and displaying these products on different lists.

Listing products

In MERN Marketplace, products will be presented to users in multiple ways. The two main distinctions will be in the way products are listed for sellers and the way they are listed for buyers. In the following sections, we will see how to list products in a shop for both sellers and buyers, then also discuss how to list product suggestions for buyers, featuring products that are related to a specific product, along with the latest products added to the marketplace.

Listing by shop

Visitors to the marketplace will browse products in each shop, and sellers will manage a list of products in each of their shops. Both these features will share the same backend API that will retrieve all the products for a specific shop but will be rendered differently for the two types of users. In the following sections, first, we will implement the backend API for fetching the products in a specific shop. Then, we will use the API in two different React components to render the list of products to the seller of the shop in one component, and to the buyers in another component.

The products by shop API

In order to implement the backend API to retrieve products from a specific shop in the database, we will set up a `GET` route at `/api/products/by/:shopId`, as shown in the following code:

mern-marketplace/server/routes/product.routes.js:

```
| router.route('/api/products/by/:shopId')
|   .get(productCtrl.listByShop)
```

The `listByShop` controller method executed in response to this request will query the Product collection to return the products matching the given shop's reference. The `listByShop` method is defined as shown in the following code:

mern-marketplace/server/controllers/product.controller.js:

```
| const listByShop = async (req, res) => {
|   try {
|     let products = await Product.find({shop: req.shop._id})
|       .populate('shop', '_id name').select('-image')
|     res.json(products)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

Each product in the resulting products array will contain the name and ID details of the associated shop, and we will omit the `image` field since images can be retrieved via separate API routes.

In the frontend, to fetch the products in a specific shop using this API to list by shop, we will also need to add a `fetch` method in `api-product.js`, similar to our other API implementations. Then, the `fetch` method can be called in any React component to render the products, for

example, to display products in a shop to all buyers, as discussed in the next section.

Products component for buyers

We will build a `Products` component, mainly for displaying the products to visitors who may buy the products. We can reuse this component across the application to render different product lists relevant to the buyer. It will receive the product list as props from a parent component that displays a list of products. A rendered Products view may look as shown in the following screenshot:



In the marketplace application, the list of products in a shop will be displayed to the user in an individual `Shop` view. So this `Products` component is added to the `Shop` component and given the list of relevant products as props, as shown next:

mern-marketplace/client/shop/Shop.js:

```
| <Products products={products} searched={false}/></Card>
```

The `searched` prop relays whether this list is a result of a product search, so appropriate messages can be rendered.

In the `Shop` component, we need to add a call to the `listByShop` fetch method in a `useEffect` hook to retrieve the relevant products and set it to state, as shown in the following code:

mern-marketplace/client/shop/Shop.js:

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

  listByShop({
    shopId: match.params.shopId
  }, signal).then((data)=>{
    if (data.error) {
      setError(data.error)
    } else {
      setProducts(data)
    }
  })

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

In the `Products` component, if the product list sent in the props contains products, the list is iterated over and the relevant details of each product are rendered in a Material-UI `GridListTile`, with a link to the individual product view and an `AddToCart` component (the implementation for which is discussed in [Chapter 8, Extending the Marketplace for Orders and Payments](#)). The code to render the list of products is added as follows:

mern-marketplace/client/product/Products.js:

```
{props.products.length > 0 ?
  (<div>
    <GridList cellHeight={200} cols={3}>
      {props.products.map((product, i) => (
        <GridListTile key={i}>
          <Link to={"/product/" + product._id}>
            <img src={'/api/product/image/' + product._id}
                 alt={product.name} />
          </Link>
          <GridListTileBar
            title={<Link to={"/product/" + product._id}>
              {product.name}</Link>}
            subtitle={<span>$ {product.price}</span>}
            actionIcon={(
              <AddToCart item={product}/>
            )}
          />
        </GridListTile>))
  )}
```

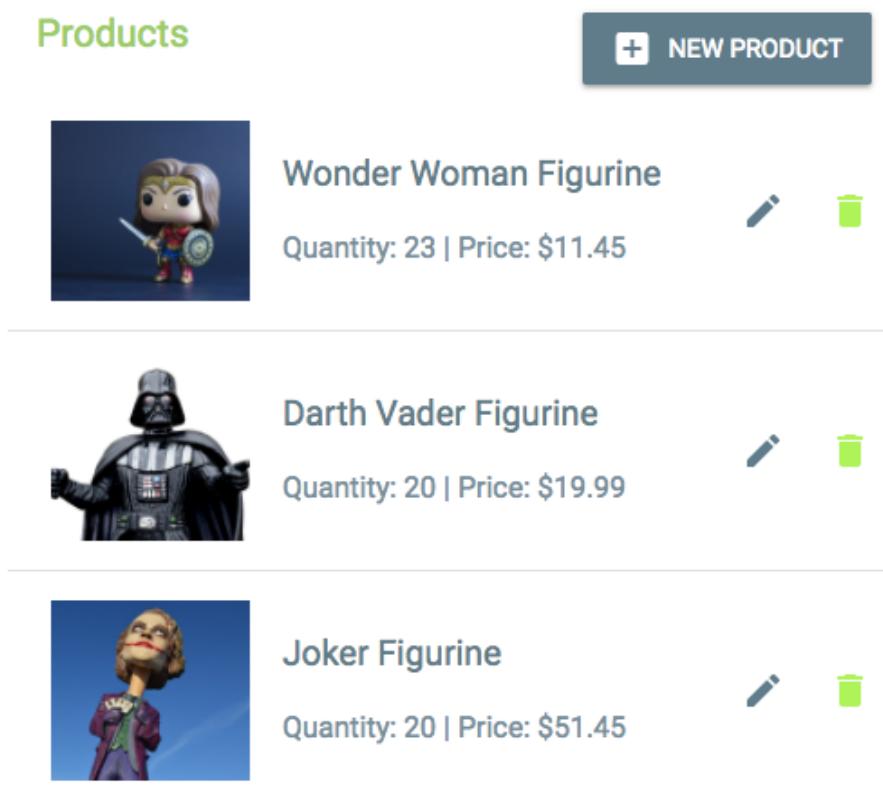
```
        }
      </GridList>
    </div>) : props.searched && (<Typography component="h4">
      No products found! :(</Typography>) }
```

If the `products` array sent in the props is found to be empty, and this was a result of a search action by the user, we render an appropriate message to inform the user that no products were found.

This `Products` component can be used to render different lists of products for buyers, including products in a shop, products by category, and products in search results. In the next section, we will implement a `MyProducts` component that will render a list of products only for shop owners, giving them a different set of interaction options.

MyProducts component for shop owners

In contrast to the `Products` component, the `MyProducts` component in `client/product/MyProducts.js` is only for displaying products to sellers so they can manage the products in each shop they own and will be displayed to the end user as pictured in the following screenshot:



The `MyProducts` component is added to the `EditShop` view as shown in the following code, so sellers can manage a shop and its contents in one place. It is provided with the shop's ID in a prop so that relevant products can be fetched:

`mern-marketplace/client/shop/EditShop.js`:

```
| <MyProducts shopId={match.params.shopId}/>
```

In `MyProducts`, the relevant products are first loaded in a state with an `useEffect` hook using the `listByShop` fetch method, as shown in the following code:

mern-marketplace/client/product/MyProducts.js:

```
export default function MyProducts (props) {
  const [products, setProducts] = useState([])
  useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal
    listByShop({
      shopId: props.shopId
    }, signal).then((data)=>{
      if (data.error) {
        console.log(data.error)
      } else {
        setProducts(data)
      }
    })
    return function cleanup(){
      abortController.abort()
    }
  }, [])
...
}
```

This list of products is then iterated over with each product rendered in the `ListItem` components along with edit and delete options, similar to the `MyShops` list view. The edit button links to the Edit Product view. The `DeleteProduct` component handles the delete action, and reloads the list by calling an `onRemove` method passed from `MyProducts` to update the state with the updated list of products for the current shop.

The `removeProduct` method, defined in `MyProducts`, is provided as the `onRemove` prop to the `DeleteProduct` component. The `removeProduct` method is defined as follows:

mern-marketplace/client/product/MyProducts.js:

```
const removeProduct = (product) => {
  const updatedProducts = [...products]
```

```
|   const index = updatedProducts.indexOf(product)
|   updatedProducts.splice(index, 1)
|   setProducts(updatedProducts)
| }
```

Then it is passed as a prop to the `DeleteProduct` component when it is added to `MyProducts` as shown next:

mern-marketplace/client/product/MyProducts.js:

```
| <DeleteProduct
|   product={product}
|   shopId={props.shopId}
|   onRemove={removeProduct}>
```

Implementing a separate `MyProducts` component this way gives the shop owner the ability to see the list of products in their shop with the option to edit and delete each. In the next section, we will complete the implementation for retrieving different types of product lists from the backend and rendering them as product suggestions for buyers in the frontend.

Listing product suggestions

Visitors to MERN Marketplace will see product suggestions, such as the latest products added to the marketplace and products related to the product they are currently viewing. In the following sections, we will first look at the implementation of the backend APIs for retrieving the latest products and a list of products related to a given product, and then implement a React component called Suggestions to render these lists of products.

Latest products

On the home page of the MERN Marketplace, we will display five of the latest products added to the marketplace. To fetch the latest products, we will set up a backend API that will receive a `GET` request at `/api/products/latest`, as shown in the following code:

mern-marketplace/server/routes/product.routes.js:

```
| router.route('/api/products/latest')
|   .get(productCtrl.listLatest)
```

A `GET` request received at this route will invoke the `listLatest` controller method. This method will find all products, sort the list of products in the database with the `created` date field from newest to oldest, and return the first five from the sorted list in the response. This `listLatest` controller method is defined as follows:

mern-marketplace/server/controllers/product.controller.js:

```
| const listLatest = async (req, res) => {
|   try {
|     let products = await Product.find({}).sort('-created')
|       .limit(5).populate('shop', '_id name').exec()
|     res.json(products)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

To use this API in the frontend, you will also need to set up a corresponding `fetch` method in `api-product.js` for this latest products API, similar to other API implementations. This retrieved list will then be rendered in the `Suggestions` component to be added to the home page. Next, we will discuss a similar API for retrieving a list of related products.

Related products

In each individual product view, we will show five related products as suggestions. To retrieve these related products, we will set up a backend API that accepts a request at `/api/products/related`, as shown in the following code.

mern-marketplace/server/routes/product.routes.js:

```
| router.route('/api/products/related/:productId')
|   .get(productCtrl.listRelated)
| router.param('productId', productCtrl.productByID)
```

The `:productId` param in the route URL route will call the `productByID` controller method, which is similar to the `shopByID` controller method, and retrieves the product from the database and attaches it to the request object to be used in the `next` method. The `productByID` controller method is defined as follows:

mern-marketplace/server/controllers/product.controller.js:

```
| const productByID = async (req, res, next, id) => {
|   try {
|     let product = await Product.findById(id)
|       .populate('shop', '_id name').exec()
|     if (!product)
|       return res.status('400').json({
|         error: "Product not found"
|       })
|     req.product = product
|     next()
|   } catch (err) {
|     return res.status('400').json({
|       error: "Could not retrieve product"
|     })
|   }
| }
```

Once the product is retrieved, the `listRelated` controller method is invoked. This method queries the `Product` collection in the database to find other products with the same category as the given product,

excluding the given product, and returns the first five products in the resulting list. This `listRelated` controller method is defined as follows:

mern-marketplace/server/controllers/product.controller.js:

```
const listRelated = async (req, res) => {
  try{
    let products = await Product.find({ "_id": { "$ne": req.product },
      "category": req.product.category})
      .limit(5).populate('shop', '_id name').exec()
    res.json(products)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

In order to utilize this related products API in the frontend, we will set up a corresponding `fetch` method in `api-product.js`. The `fetch` method will be called in the `Product` component with the product ID to populate the `Suggestions` component rendered in the product view. We will look at the implementation of this `Suggestions` component in the next section.

The Suggestions component

The `Suggestions` component will be rendered on the home page and on an individual product page to show the latest products and related products, respectively. Once rendered, the `Suggestions` component may look as shown in the following screenshot:

Latest Products



Swiss Army Pocket Knife
Toyish
Added on Wed Jan 10 2018
\$ 25.99  



Red Stiletto Pumps
Happy Feet
Added on Tue Jan 09 2018
\$ 11.96  



Blue Winter Boots
Happy Feet
Added on Tue Jan 09 2018
\$ 68.99  

This component will receive the relevant list of products from the parent component as props, along with a title for the list:

```
| <Suggestions products={suggestions} title={suggestionTitle}/>
```

In the `Suggestions` component, the received list is iterated over and individual products are rendered with relevant details, a link to the individual product page, and an `AddToCart` component, as shown in the following code.

mern-marketplace/client/product/Suggestions.js:

```
<Typography type="title"> {props.title} </Typography>
{props.products.map((item, i) => {
  return <span key={i}>
    <Card>
      <CardMedia image={'/api/product/image/' + item._id}
                 title={item.name}/>
      <CardContent>
        <Link to={'/product/' + item._id}>
          <Typography type="title" component="h3">
            {item.name}</Typography>
        </Link>
        <Link to={'/shops/' + item.shop._id}>
          <Typography type="subheading">
            <Icon>shopping_basket</Icon> {item.shop.name}
          </Typography>
        </Link>
        <Typography component="p">
          Added on {(new Date(item.created)).toDateString()}
        </Typography>
      </CardContent>
      <Typography type="subheading" component="h3">$ {item.price}</Typography>
      <Link to={'/product/' + item._id}>
        <IconButton color="secondary" dense="dense">
          <ViewIcon className={classes.iconButton}/>
        </IconButton>
      </Link>
      <AddToCart item={item}/>
    </Card>
  </span>))}
```

This `Suggestions` component can be reused to render any list of products to buyers, and in this section, we have discussed how to retrieve and display two different lists of products. Each product in the lists is linked to a view that will render details of the individual product. In the next section, we will look at the implementation of reading and displaying a single product to the end user.

Displaying a product

Visitors to the MERN Marketplace will be able to view more details of each product in a separate view. In the following sections, we will implement a backend API to retrieve a single product from the database and then use it in the frontend to render the single product in a React component.

Read a product API

In the backend, we will add an API with a `GET` route that queries the Products collection with an ID and returns the product in the response. The route will be declared as shown in the following code:

```
mern-marketplace/server/routes/product.routes.js:
```

```
| router.route('/api/products/:productId')
|   .get(productCtrl.read)
```

The `:productId` param in the URL invokes the `productByID` controller method, which retrieves the product from the database and appends it to the request object. The product in the request object is used by the `read` controller method to respond to the `GET` request.

The read controller method is defined as follows:

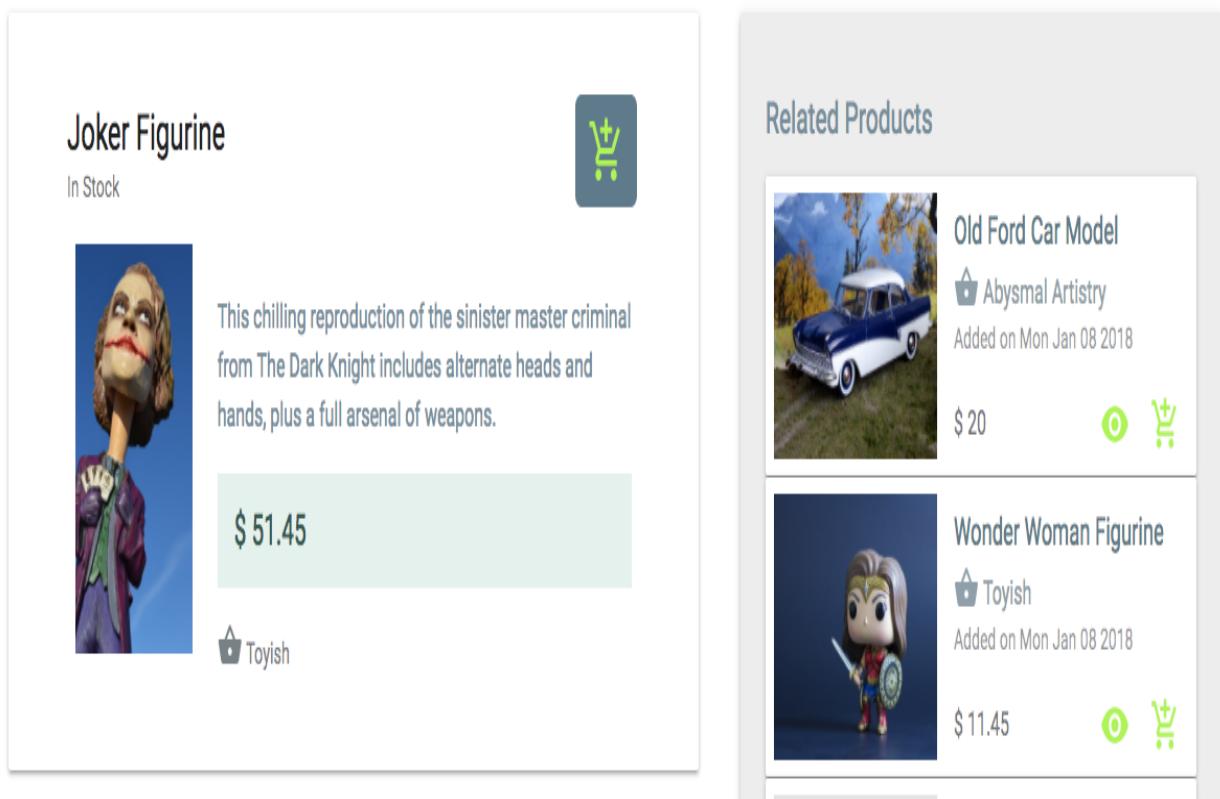
```
mern-marketplace/server/controllers/product.controller.js:
```

```
| const read = (req, res) => {
|   req.product.image = undefined
|   return res.json(req.product)
| }
```

To use this read product API in the frontend, we will need to add a `fetch` method in `client/product/api-product.js`, similar to other API implementations. Then this `fetch` method can be used in the React component, which will render the individual product details, as discussed in the next section.

Product component

We will add a React component named `Product` to render the individual product details, along with an add to cart option. In this single product view, we will also show a list of related products, as pictured in the following screenshot:



The `Product` component can be accessed in the browser at the `/product/:productId` route, which is defined in `MainRouter` as follows:

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

```
| <Route path="/product/:productId" component={Product} />
```

The product details and the related product list data will be fetched by calling the relevant APIs with `useEffect` hooks using the `productId`

specified in the route param, as shown in the following code:

mern-marketplace/client/product/Product.js:

```
export default function Product ({match}) {
  const [product, setProduct] = useState({shop:{}})
  const [suggestions, setSuggestions] = useState([])
  const [error, setError] = useState('')
  useEffect() => {
    const abortController = new AbortController()
    const signal = abortController.signal

    read({productId: match.params.productId, signal}).then((data) => {
      if (data.error) {
        setError(data.error)
      } else {
        setProduct(data)
      }
    })
    return function cleanup() {
      abortController.abort()
    }
  }, [match.params.productId])

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

    listRelated({
      productId: match.params.productId, signal).then((data) => {
      if (data.error) {
        setError(data.error)
      } else {
        setSuggestions(data)
      }
    })
    return function cleanup() {
      abortController.abort()
    }
  }, [match.params.productId])
```

In the first `useEffect` hook, we call the `read` API to retrieve the specified product and set it to state. In the second hook, we call the `listRelated` API to get the list of related products and set it to the state to be passed as a prop to a `Suggestions` component added in the product view.

The product details part of the component displays relevant information about the product and an `AddToCart` component in a Material-UI `Card` component, as shown in the following code:

```
mern-marketplace/client/product/Product.js:
```

```
|<Card>
|  <CardHeader
|    action={<AddToCart cartStyle={classes.addCart}
|      item={product}/>}
|    title={product.name}
|    subheader={product.quantity > 0? 'In Stock': 'Out of
|      Stock'}
|  />
|  <CardMedia image={imageUrl} title={product.name}/>
|  <Typography component="p" variant="subtitle1">
|    {product.description}<br/>
|    $ {product.price}
|    <Link to={`/shops/${product.shop._id}`}>
|      <Icon>shopping_basket</Icon> {product.shop.name}
|    </Link>
|  </Typography>
|</Card>
```

The `Suggestions` component is added in the `Product` view with the related list data passed as a prop, as shown next:

```
mern-marketplace/client/product/Product.js:
```

```
|<Suggestions products={suggestions} title='Related Products' />
```

With this view complete, visitors to the marketplace application will be able to find out more about a specific product as well as explore other similar products. In the next section, we will discuss how to add the ability for shop owners to edit and delete the products they added to the marketplace.

Editing and deleting a product

Implementations to edit and delete products in the application are similar to editing and deleting shops, as covered in the previous sections, *Editing a shop* and *Deleting a shop*. These functionalities will require the corresponding APIs in the backend, fetch methods in the frontend, and React component views with forms and actions. In the following sections, we will highlight the frontend view, route, and backend API endpoints for editing and deleting a product from the marketplace.

Edit

The edit functionality is very similar to the create product functionality we implemented earlier. The `EditProduct` form component, which can be implemented to render a form that allows product detail modification, will also only be accessible by verified sellers at

`/seller/:shopId/:productId/edit.`

To restrict access to this view, we can add a `PrivateRoute` in `MainRouter` to declare the route to the `EditProduct` view as follows:

mern-marketplace/client/MainRouter.js:

```
| <PrivateRoute path="/seller/:shopId/:productId/edit" component={EditProduct}/>
```

The `EditProduct` component contains the same form as `NewProduct`, but with populated values of the product retrieved using the read product API. On form submit, it uses a `fetch` method to send multipart form data with a PUT request to the edit product API in the backend at `/api/products/by/:shopId`. This backend route declaration for the edit product API will be as follows:

mern-marketplace/server/routes/product.routes.js:

```
| router.route('/api/product/:shopId/:productId')
|   .put(authCtrl.requireSignin, shopCtrl.isOwner, productCtrl.update)
```

The `update` controller method is invoked when an authorized user sends a PUT request to this API. It is similar to the product `create` method and the shop `update` method. It handles the multipart form data using `formidable` and extends the product details to save the updates to the database.

This implementation of an edit product form view integrated with an update API in the backend will allow shop owners to modify the

details of products in their shops. Next, we will look at the highlights for integrating product deletion functionality to the application.

Delete

In order to implement the delete product functionality, we can implement a `DeleteProduct` component similar to the `DeleteShop` component, and add it to the `MyProducts` component for each product in the list. It can take the `product` object, `shopID`, and an `onRemove` method as a prop from `MyProducts`, as discussed in the *MyProducts component for shop owners* section.

The component will function the same as `DeleteShop`, opening a dialog for confirmation on button-click and then, when the delete intent is confirmed by the user, calling the `fetch` method for delete, which makes the DELETE request to the server at `/api/product/:shopId/:productId`. This backend API for deleting a product from the database will be declared as follows with the other product routes:

mern-marketplace/server/routes/product.routes.js:

```
| router.route('/api/product/:shopId/:productId')
|   .delete(authCtrl.requireSignin, shopCtrl.isOwner,
|           productCtrl.remove)
```

The `remove` controller method will be invoked if an authorized user makes the DELETE request to this API, and it will delete the specified product from the database, like the `remove` controller method for shops.

We started the implementation of the product-related features for the marketplace in this section by first defining a schema for storing product details and then discussing the full-stack slices for creating, listing, reading, updating, and deleting products in the application. In the next section, we will look into how to allow users in the marketplace to search for products in varied ways, so they can easily find the products they are looking for.

Searching for products with name and category

In MERN Marketplace, visitors will be able to search for specific products by name and also in a specific category. In the following sections, we will discuss how this search functionality can be added by first looking at backend APIs that will retrieve the distinct categories from the Products collection, and perform the search query against the products stored. Then, we will discuss different cases for utilizing these APIs, such as a view to perform the search action and a view for displaying products by categories.

The categories API

To allow users to select a specific category to search in, we will first set up an API that retrieves all the distinct categories present in the Products collection in the database. A `GET` request to `/api/products/categories` will return an array of unique categories, and this route is declared as shown here:

```
mern-marketplace/server/routes/product.routes.js:
```

```
| router.route('/api/products/categories')
|   .get(productCtrl.listCategories)
```

The `listCategories` controller method queries the Products collection with a `distinct` call against the `category` field, as shown in the following code:

```
mern-marketplace/server/controllers/product.controller.js:
```

```
| const listCategories = async (req, res) => {
|   try {
|     let products = await Product.distinct('category', {})
|     res.json(products)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

This categories API can be used in the frontend with a corresponding `fetch` method to retrieve the array of distinct categories and displayed in the view. This can be paired with a search API to allow users to search for products by its name in a specific category. In the next section, we will discuss this search API.

The search products API

We can define a search products API that will take a `GET` request at `/api/products?search=value&category=value`, with query parameters in the URL to query the Products collection with the provided search text and category values. The route for this search API will be defined as follows:

mern-marketplace/server/routes/product.routes.js:

```
| router.route('/api/products')
|   .get(productCtrl.list)
```

The `list` controller method will first process the query parameters in the request, then find products in the given category, if any, with names that partially match with the provided search text. This `list` method is defined as shown in the following code:

mern-marketplace/server/controllers/product.controller.js:

```
const list = async (req, res) => {
  const query = {}
  if(req.query.search)
    query.name = {'$regex': req.query.search, '$options': "i"}
  if(req.query.category && req.query.category != 'All')
    query.category = req.query.category
  try {
    let products = await Product.find(query)
      .populate('shop', '_id name')
      .select('-image').exec()
    res.json(products)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The resulting products returned against the provided query parameters in the request are populated with shop details and downsized by removing the image field value, before being sent

back in the response. To use this API in the frontend to perform a product search, we will need a `fetch` method that can construct the query parameters in the request URL, as discussed in the next section.

Fetch search results for the view

To utilize this search API in the frontend, we will set up a method that constructs the URL with query parameters and calls a fetch to make a request to the search product API. This `fetch` method will be defined as follows.

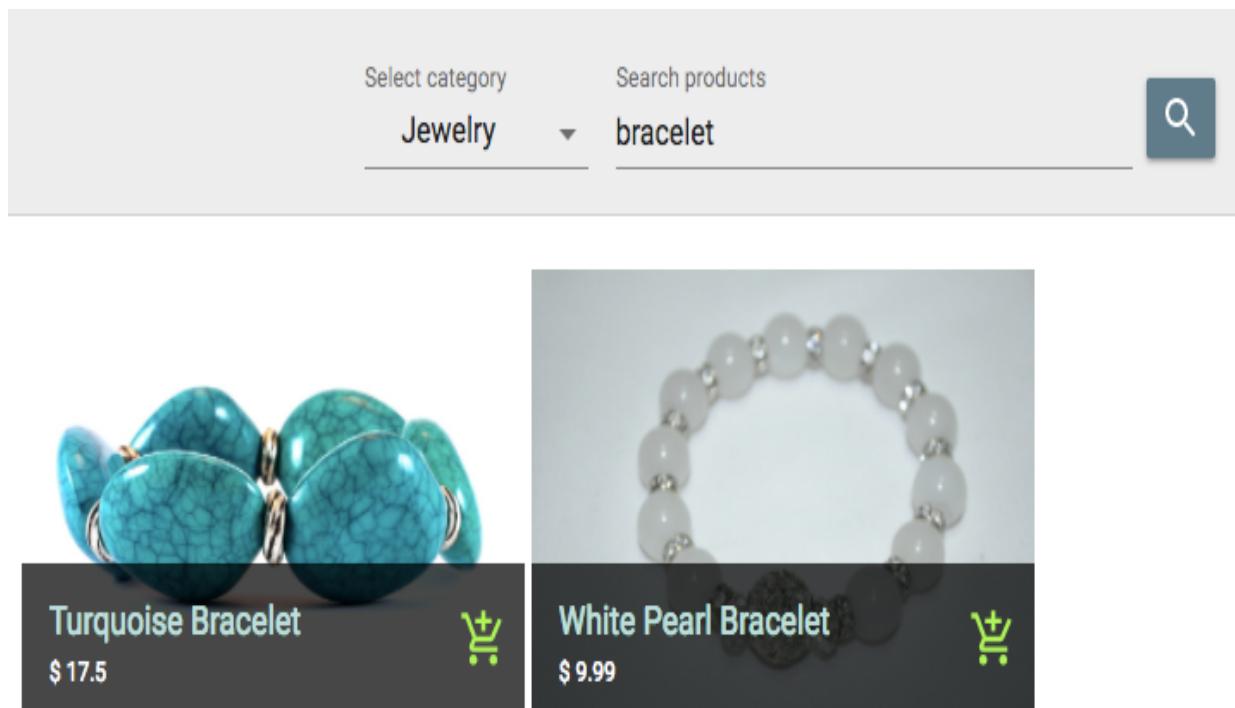
mern-marketplace/client/product/api-product.js:

```
import queryString from 'query-string'
const list = (params) => {
  const query = queryString.stringify(params)
  return fetch('/api/products?' + query, {
    method: 'GET',
  }).then(response => {
    return response.json()
  }).catch((err) => console.log(err))
}
```

In order to construct the query parameters in the correct format, we will use the `query-string` node module, which will help stringify the `params` object into a query string that can be attached to the request route URL. The keys and values in this `params` object will be defined by the React component where we call this `list` method. Next, we will look at the `Search` component, which will utilize this method to enable the end user to search for products in the marketplace.

The Search component

The first use case for applying the categories API and search API together to perform a search action is in the `Search` component. This component, once implemented and functional, will render as shown in the following screenshot:



This `Search` component provides the user with a simple form containing a search input text field and a dropdown of the category options received from a parent component that will retrieve the list using the distinct categories API. The code to render this search form view will be as follows:

mern-marketplace/client/product/Search.js:

```
<TextField id="select-category" select label="Select category" value={category}
    onChange={handleChange('category')}
    selectProps={{ MenuProps: { className: classes.menu, } }}>
```

```

<MenuItem value="All"> All </MenuItem>
{props.categories.map(option => (
  <MenuItem key={option} value={option}> {option} </MenuItem>
))
</TextField>
<TextField id="search" label="Search products" type="search" onKeyDown={enterKey}
  onChange={handleChange('search')}
/>
<Button raised onClick={search}> Search </Button>

```

Once the user enters a search text and hits *Enter*, we will make a call to the `search` method. To detect that the *Enter* key was pressed, we use the `onKeyDown` attribute on the `TextField` and define the `enterKey` handler method as follows:

mern-marketplace/client/product/Search.js:

```

const enterKey = (event) => {
  if(event.keyCode == 13) {
    event.preventDefault()
    search()
  }
}

```

The `search` method makes a call to the search API using the `list` `fetch` method, providing it with the necessary search query parameters and values. This `search` method is defined as shown in the following code:

mern-marketplace/client/product/Search.js:

```

const search = () => {
  if(values.search) {
    list({
      search: values.search || undefined, category: values.category
    }).then((data) => {
      if (data.error) {
        console.log(data.error)
      } else {
        setValues({...values, results: data, searched:true})
      }
    })
  }
}

```

In this method, the query parameters provided to the `list` method are the search text value, if any, and the selected category value. Then the results array received from the backend is set to the values in

state and passed as a prop to the `Products` component, as shown next, to render the matching products underneath the search form:

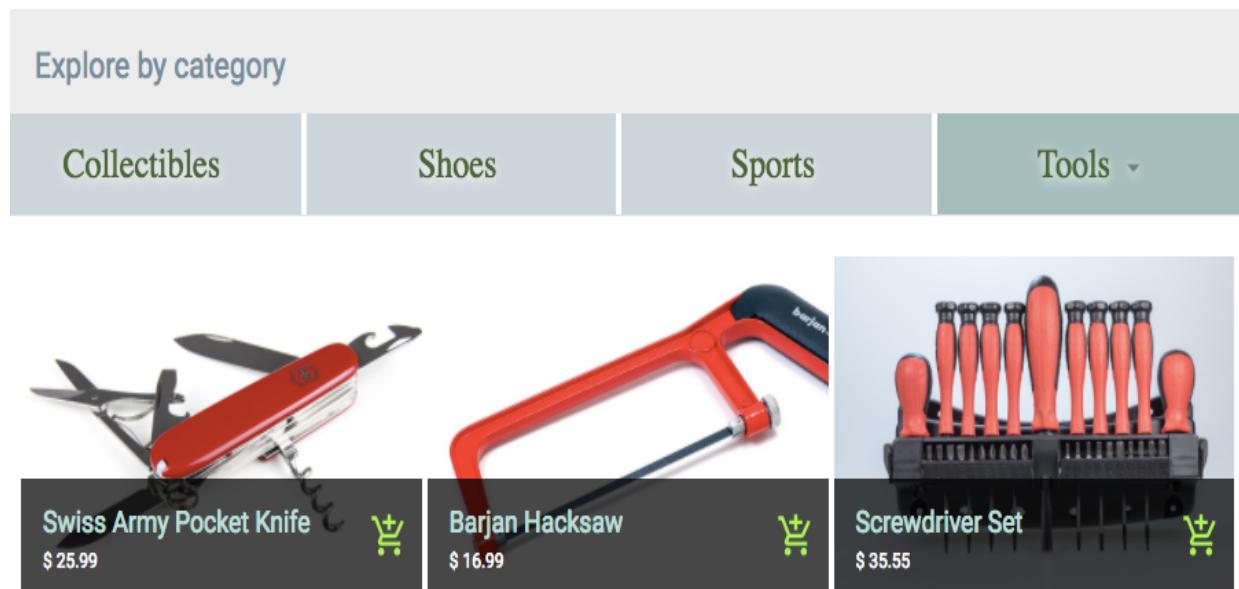
`mern-marketplace/client/product/Search.js`:

```
| <Products products={results} searched={searched} />
```

This search view gives visitors a useful tool to look for the specific product they want among many that may be stored in the database for the complete marketplace. In the next section, we will look at another simple use case for utilizing the categories and search APIs in the frontend.

The Categories component

The `Categories` component is the second use case for the distinct categories and search APIs. For this component, we first fetch the list of categories in a parent component and send it as props to display the categories to the user, as shown in the following screenshot:



When the user selects a category in the displayed list, a call is made to the Search API with just a category value, and the backend returns all the products in the selected category. The returned products are then rendered in a `Products` component. This can be a simple way to combine these APIs and display meaningful products to buyers browsing through the marketplace.

In this first version of the MERN Marketplace, users can become sellers to create shops and add products, and visitors can browse shops and search for products, while the application also suggests products to the visitors.

Summary

In this chapter, we started building an online marketplace application using the MERN stack. The MERN skeleton was extended to allow users to have active seller accounts, so they can create shops and add products to each shop with the intention to sell to other users. We also explored how to utilize the stack to implement features such as product browsing, searching, and suggestions for regular users who are interesting in buying.

While going through the implementations in this chapter, we explored how to lay down the foundations with full-stack implementations to be able to combine and extend interesting features such as search and suggestions. You can apply these same approaches while building out other full-stack applications that may require these features.

Even with these features incorporated, a marketplace application is still incomplete without a shopping cart for checkout, order management, and payment processing. In the next chapter, we will grow our marketplace application to add these advanced features and learn more about how the MERN stack can be used to implement these core aspects of an e-commerce application.

Extending the Marketplace for Orders and Payments

Processing payments from customers when they place orders and allowing sellers to manage these orders are key aspects of e-commerce applications. In this chapter, we'll extend the online marketplace we built in the previous chapter by implementing capabilities for buyers to add products to a shopping cart, a checkout, and place orders, and for sellers to manage these orders and have payments processed from the marketplace application. Once you've gone through this chapter and added these features, besides extending the marketplace application with advanced features, you will be able to utilize browser storage, process payments using Stripe, and integrate other technologies into this stack.

In this chapter, we will extend the online marketplace by covering the following topics:

- Introducing a cart, payments, and orders in the MERN Marketplace
- Implementing a shopping cart
- Using Stripe for payments
- Integrating the checkout process
- Creating a new order
- Listing orders for each shop
- Viewing single-order details

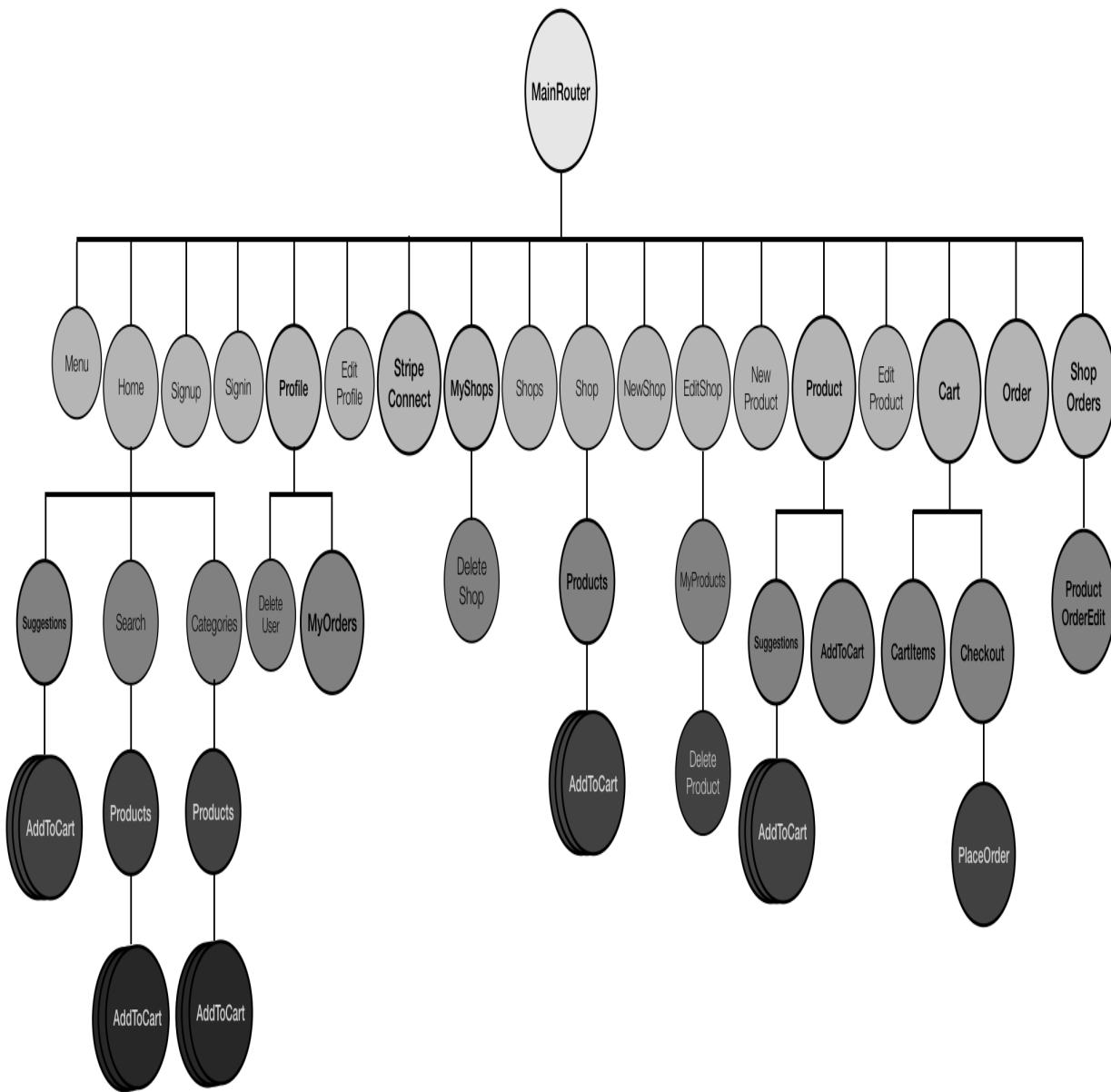
Introducing cart, payments, and orders in the MERN Marketplace

The MERN Marketplace application we developed in [Chapter 7, Exercising MERN Skills with an Online Marketplace](#), has very simple features and is missing core e-commerce functionality. In this chapter, we will extend this marketplace application so that it includes a shopping cart feature for the buyer, Stripe integration for processing credit card payments, and a basic order-management flow for the seller. The implementations that follow are kept simple to serve as starting points for developing more complex versions of these features for your own applications.



The code for the complete MERN Marketplace application is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter07%20and%2008/mern-marketplace>. You can clone this code and run the application as you go through the code explanations in the rest of this chapter. To get the code for Stripe payments working, you will need to create your own Stripe account and update the `config/config.js` file with your testing values for the Stripe API key, secret key, and Stripe Connect client ID.

The following component tree diagram shows the custom components that make up the MERN Marketplace frontend, including the components for the shopping cart, payments, and order-related features that will be implemented in the rest of this chapter:



The features that will be discussed in this chapter modify some of the existing components, such as `Profile`, `MyShops`, `Products`, and `Suggestions`, and also add new components, such as `AddToCart`, `MyOrders`, `Cart`, and `ShopOrders`. In the next section, we will begin extending the online marketplace with the implementation of the shopping cart.

Implementing a shopping cart

Visitors to the MERN Marketplace can add products they wish to buy to a shopping cart by clicking the add to cart button on each product. A cart icon on the menu will indicate the number of products that have already been added to their cart as the user continues to browse through the marketplace. They can also update the cart's contents and begin the checkout process by opening the cart view. But to complete the checkout process and place an order, users will be required to sign in.

The shopping cart is mainly a frontend feature, so the cart details will be stored locally on the client side until the user places the order at checkout. To implement the shopping cart features, we will set up helper methods in `client/cart/cart-helper.js` that will help manipulate the cart details from relevant React components.

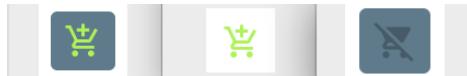
In the following sections, we will look at how to add products to the cart, update the menu to indicate the status of the cart, and implement the cart view where users can see and modify all the items that have already been added to their cart before checking out.

Adding to the cart

While browsing the products in the marketplace, users will see the option on each product to add it to their cart. This option will be implemented with a React component named `AddToCart`. This `AddToCart` component in `client/Cart/AddToCart.js` takes a `product` object and a CSS style object as props from the parent component it is added to. For example, in the MERN Marketplace, it is added to a Product view as follows:

```
| <AddToCart cartStyle={classes.addCart} item={product}/>
```

The `AddToCart` component, when rendered, displays a cart icon button depending on whether the passed item is in stock or not, as shown in the following screenshot:



For example, if the item quantity is more than 0, `AddCartIcon` is displayed; otherwise, `DisabledCartIcon` is rendered. The appearance of the icon depends on the CSS style object that's passed in the props. The code to render these variations of the `AddToCart` button is as follows.

`mern-marketplace/client/cart/AddToCart.js`:

```
{ props.item.quantity >= 0 ?  
    <IconButton color="secondary" dense="dense" onClick={addToCart}>  
        <AddCartIcon className={props.cartStyle || classes.iconButton}/>  
    </IconButton> :  
    <IconButton disabled={true} color="secondary" dense="dense">  
        <DisabledCartIcon className={props.cartStyle ||  
        classes.disabledIconButton}/>  
    </IconButton>  
}
```

The `AddCartIcon` button calls an `addToCart` method when it is clicked. The `addToCart` method is defined as follows.

mern-marketplace/client/cart/AddToCart.js:

```
const addToCart = () => {
  cart.addItem(props.item, () => {
    setRedirect({redirect:true})
  })
}
```

The `addToCart` method invokes the `addItem` helper method defined in `cart-helper.js`. This `addItem` method takes the `product` item and a state-updating `callback` function as parameters and stores the updated cart details in `localStorage` and executes the callback that was passed, as shown in the following code.

mern-marketplace/client/cart/cart-helper.js:

```
addItem(item, cb) {
  let cart = []
  if (typeof window !== "undefined") {
    if (localStorage.getItem('cart')) {
      cart = JSON.parse(localStorage.getItem('cart'))
    }
    cart.push({
      product: item,
      quantity: 1,
      shop: item.shop._id
    })
    localStorage.setItem('cart', JSON.stringify(cart))
    cb()
  }
}
```

The cart data stored in `localStorage` contains an array of cart item objects, each containing product details, the quantity of the product that was added to the cart (which is set to `1` by default), and the ID of the shop the product belongs to. As products get added to the cart and stored in `localStorage`, we will also display the updated item count on the navigation menu, as discussed in the next section.

Cart icon in the menu

In the menu, we will add a link to the cart view, as well as a badge that displays the length of the cart array stored in `localStorage` in order to visually inform the user of how many items are currently in their cart. The rendered link and badge will look as follows:



The link for the cart will be similar to the other links in the menu, with the exception of the Material-UI `Badge` component, which displays the cart length. It will be added as follows:

mern-marketplace/client/core/Menu.js:

```
<Link to="/cart">
  <Button color={isActive(history, "/cart")}>
    Cart
    <Badge invisible={false} color="secondary"
      badgeContent= {cart.itemTotal()}>
      <CartIcon />
    </Badge>
  </Button>
</Link>
```

The cart length is returned by the `itemTotal` helper method in `cart-helper.js`, which reads the `cart` array stored in `localStorage` and returns the length of the array. The `itemTotal` method is defined as follows.

mern-marketplace/client/cart/cart-helper.js:

```
itemTotal() {
  if (typeof window !== "undefined") {
    if (localStorage.getItem('cart')) {
      return JSON.parse(localStorage.getItem('cart')).length
    }
  }
  return 0
}
```

Clicking on this cart link, with the item total displayed on the menu, will take the user to the cart view and reveal details of the items that have already been added to the cart. In the next section, we will discuss the implementation of this cart view.

The cart view

The cart view will contain the cart items and checkout details. But initially, only the cart details will be displayed until the user is ready to check out. The code to render this cart view will be added as follows.

mern-marketplace/client/cart/Cart.js:

```
| <Grid container spacing={24}>
|   <Grid item xs={6} sm={6}>
|     <CartItems checkout={checkout}
|       setCheckout={showCheckout}/>
|   </Grid>
| {checkout &&
|   <Grid item xs={6} sm={6}>
|     <Checkout/>
|   </Grid>}
| </Grid>
```

The `CartItems` component, which displays the items in the cart, is passed a `checkout` Boolean value and a state update method for this `checkout` value so that the `Checkout` component and its options can be rendered conditionally based on user interaction.

The `showCheckout` method to update the `checkout` value is defined as follows.

mern-marketplace/client/cart/Cart.js:

```
| const showCheckout = val => {
|   setCheckout(val)
| }
```

The `Cart` component will be accessed at the `/cart` route, so we need to add a `Route` to the `MainRouter` component as follows.

mern-marketplace/client/MainRouter.js:

```
| <Route path="/cart" component={Cart}/>
```

This is the link we use on the Menu to redirect the user to the cart view, which contains cart details. In the next section, we will look at the implementation of the `CartItems` component, which will render details of each item in the cart and allow modifications.

The CartItems component

The `CartItems` component will allow the user to view and update the items currently in their cart. It will also give them the option to start the checkout process if they are signed in, as shown in the following screenshot:

Shopping Cart

	Joker Figurine \$ 51.45 Shop: Toyish Quantity: 1	\$51.45
	Wonder Woman Figurine \$ 11.45 Shop: Toyish Quantity: 1	\$11.45
	Speed Skipping Rope \$ 7.55 Shop: Athletico Quantity: 2	\$15.1

Total: \$78 [SIGN IN TO CHECKOUT](#) [CONTINUE SHOPPING](#)

If the cart contains items, the `CartItems` component iterates over the items and renders the products in the cart. If no items have been added, the cart view just displays a message stating that the cart is empty. The code for this implementation is as follows.

mern-marketplace/client/cart/CartItems.js:

```
| {cartItems.length > 0 ? <span>
|   (cartItems.map((item, i) => {
```

```
    ...
    ... Display product details
    ... Edit quantity
    ... Remove product option
    ...
  })
}
... Show total price and Checkout options ...
</span> :
<Typography variant="subtitle1" component="h3" color="primary">
  No items added to your cart.
</Typography>
}
```

For each product item, we show the details of the product and an editable quantity text field, along with a remove item option. Finally, we show the total price of the items in the cart and the option to start the checkout operation. In the following sections, we will look into the implementations of these cart item display and modification options.

Retrieving cart details

Before the cart item details can be displayed, we need to retrieve the cart details stored in `localStorage`. For this purpose, we implement the `getCart` helper method in `cart-helper.js`, which retrieves and returns the cart details from `localStorage`, as shown in the following code.

mern-marketplace/client/cart/cart-helper.js:

```
getCart() {
  if (typeof window !== "undefined") {
    if (localStorage.getItem('cart')) {
      return JSON.parse(localStorage.getItem('cart'))
    }
  }
  return []
}
```

In the `CartItems` component, we will retrieve the cart items using the `getCart` helper method and set it to the state of the initial value of `cartItems`, as shown in the following code.

mern-marketplace/client/cart/CartItems.js:

```
| const [cartItems, setCartItems] = useState(cart.getCart())
```

Then, this `cartItems` array that was retrieved from `localStorage` is iterated over using the `map` function to render the details of each item, as shown in the following code.

mern-marketplace/client/cart/CartItems.js:

```
<span key={i}>
  <Card>
    <CardMedia image={'/api/product/image/' + item.product._id}
              title={item.product.name}/>
    <CardContent>
      <Link to={'/product/' + item.product._id}>
        <Typography type="title" component="h3"
                    color="primary">
          {item.product.name}</Typography>
      </Link>
    </CardContent>
  </Card>
</span>
```

```
<Typography type="subheading" component="h3"
color="primary">
  $ {item.product.price}
</Typography>
<span>${item.product.price * item.quantity}</span>
<span>Shop: {item.product.shop.name}</span>
</CardContent>
<div>
  ... Editable quantity ...
  ... Remove item option ...
</div>
</Card>
<Divider/>
</span>
```

For each rendered cart item, we will also give the user the option to change the quantity, as discussed in the next section.

Modifying quantity

Each cart item displayed in the cart view will contain an editable `TextField` that will allow the user to update the quantity for each product they are buying, with a minimum allowed value of `1`, as shown in the following code.

mern-marketplace/client/cart/CartItems.js:

```
Quantity: <TextField  
          value={item.quantity}  
          onChange={handleChange(i)}  
          type="number"  
          inputProps={{ min:1 }}  
          InputLabelProps={{  
            shrink: true,  
          }}  
        />
```

When the user updates this value, the `handleChange` method is called to enforce the minimum value validation, update the `cartItems` in the state, and update the cart in `localStorage` using a helper method. The `handleChange` method is defined as follows.

mern-marketplace/client/cart/CartItems.js:

```
const handleChange = index => event => {  
  let updatedCartItems = cartItems  
  if(event.target.value == 0){  
    updatedCartItems[index].quantity = 1  
  }else{  
    updatedCartItems[index].quantity = event.target.value  
  }  
  setCartItems([...updatedCartItems])  
  cart.updateCart(index, event.target.value)  
}
```

The `updateCart` helper method takes the index of the product being updated in the cart array and the new quantity value as parameters and updates the details stored in `localStorage`. This `updateCart` helper method is defined as follows.

mern-marketplace/client/cart/cart-helper.js:

```
updateCart(itemIndex, quantity) {
  let cart = []
  if (typeof window !== "undefined") {
    if (localStorage.getItem('cart')) {
      cart = JSON.parse(localStorage.getItem('cart'))
    }
    cart[itemIndex].quantity = quantity
    localStorage.setItem('cart', JSON.stringify(cart))
  }
}
```

Besides updating the item quantity in the cart, users will also have the option to remove the item from the cart, as discussed in the next section.

Removing items

Each item in the cart will have a remove option next to it. This remove item option is a button that, when clicked, passes the array index of the item to the `removeItem` method so that it can be removed from the array. This button is rendered with the following code.

mern-marketplace/client/cart/CartItems.js:

```
| <Button color="primary" onClick={removeItem(i)}>x Remove</Button>
```

The `removeItem` click handler method uses the `removeItem` helper method to remove the item from the cart in `localStorage`, then updates the `cartItems` in the state. This method also checks whether the cart has been emptied so that checkout can be hidden by using the `setCheckout` function passed as a prop from the `Cart` component. The `removeItem` click handler method is defined as follows.

mern-marketplace/client/cart/CartItems.js:

```
| const removeItem = index => event =>{
|   let updatedCartItems = cart.removeItem(index)
|   if(updatedCartItems.length == 0){
|     props.setCheckout(false)
|   }
|   setCartItems(updatedCartItems)
| }
```

The `removeItem` helper method in `cart-helper.js` takes the index of the product to be removed from the array, splices it out, and updates `localStorage` before returning the updated `cart` array. This `removeItem` helper method is defined as follows.

mern-marketplace/client/cart/cart-helper.js:

```
| removeItem(itemIndex) {
|   let cart = []
|   if (typeof window !== "undefined") {
|     if (localStorage.getItem('cart')) {
```

```
    cart = JSON.parse(localStorage.getItem('cart'))
  }
  cart.splice(itemIndex, 1)
  localStorage.setItem('cart', JSON.stringify(cart))
}
return cart
}
```

As users modify the items in their cart by either changing the quantity or removing an item, they will also see the updated total price of all the items currently in the cart, as discussed in the next section.

Showing the total price

At the bottom of the `CartItems` component, we will display the total price of the items in the cart. It will be rendered with the following code.

mern-marketplace/client/cart/CartItems.js:

```
| <span className={classes.total}>Total: ${getTotal()}</span>
```

The `getTotal` method will calculate the total price while taking the unit price and quantity of each item in the `cartItems` array into consideration. This method is defined as follows.

mern-marketplace/client/cart/CartItems.js:

```
const getTotal = () => {
  return cartItems.reduce((a, b) => {
    return a + (b.quantity*b.product.price)
  }, 0)
}
```

With this, the users will have an overview of what they are buying and how much it will cost before they are ready to check out and place the order. In the next section, we will look at how to render the checkout option conditionally, depending on the state of the cart and whether the user is signed in.

Option to check out

The user will see the option to perform the checkout depending on whether they are signed in and whether the checkout has already been opened, as implemented in the following code.

mern-marketplace/client/cart/CartItems.js:

```
{!props.checkout && (auth.isAuthenticated() ?  
    <Button onClick={openCheckout}>  
        Checkout  
    </Button> :  
    <Link to="/signin">  
        <Button>Sign in to checkout</Button>  
    </Link>)  
}
```

When the checkout button is clicked, the `openCheckout` method will use the `setCheckout` method passed as a prop to set the checkout value to `true` in the `Cart` component. The `openCheckout` method is defined as follows.

mern-marketplace/client/cart/CartItems.js:

```
const openCheckout = () => {  
    props.setCheckout(true)  
}
```

Once the checkout value is set to `true` in the `Cart` view, the `Checkout` component will be rendered to allow the user to enter the checkout details and place an order.

This will complete the buying process for a user, who is now able to add items to their shopping cart and modify each item until they are ready to checkout. But before getting into the implementation of the checkout functionality, which will involve gathering and processing payment information, in the next section, we will discuss how to use Stripe in our application to add the intended payment-related features.

Using Stripe for payments

Payment processing is required across implementations of the checkout, order creation, and order management processes. It also involves making updates to both the buyer's and seller's user data. Before we delve into the implementations of the checkout and order features, we will briefly discuss payment processing options and considerations using Stripe and learn how to integrate it in the MERN Marketplace.

Stripe provides an extensive set of tools that are necessary to integrate payments in any web application. These tools can be selected and used in different ways, depending on the specific type of application and the payment use case being implemented.

In the case of the MERN Marketplace setup, the application itself will have a platform on Stripe and will expect sellers to have connected Stripe accounts on the platform so that the application can charge users who enter their credit card details at checkout on behalf of the sellers. In the MERN Marketplace, a user can add products from different shops to their shopping cart so that charges on their cards will only be created by the application for the specific product that was ordered when it is processed by the seller. Additionally, sellers will have complete control over the charges that are created on their behalf from their own Stripe dashboards. We will demonstrate how to use the tools provided by Stripe to get this payment setup working.

Stripe provides a complete set of documentation and guidelines for each tool and also exposes testing data for accounts and platforms that are set up on Stripe. For the purpose of implementing payments in the MERN Marketplace, we will be using testing keys and leave it up to you to extend the implementation for live payments.

In the following sections, we will discuss how to connect a Stripe account for each seller, collect credit card details from the user with Stripe Card Elements, use Stripe Customer to record the user's payment information securely, and create a charge with Stripe for processing a payment.

Stripe-connected account for each seller

To create charges on behalf of sellers, the application will let a user, who is a seller, connect their Stripe account to their MERN Marketplace user account. In the following sections, we will implement this functionality by updating the user model so that it can store Stripe credentials, add the view components to allow users to connect to Stripe, and add a backend API to complete Stripe OAuth before updating the database with the retrieved credentials from Stripe.

Updating the user model

When a seller connects their Stripe account to the marketplace, we will need to store their Stripe credentials with their other user details so that they can be used later for payment processing when they sell products. To store the Stripe OAuth credentials after a user's Stripe account is successfully connected, we will update the user model that we developed in [Chapter 3](#), *Building a Backend with MongoDB, Express, and Node*, with the following field.

mern-marketplace/server/models/user.model.js:

```
| stripe_seller: {}
```

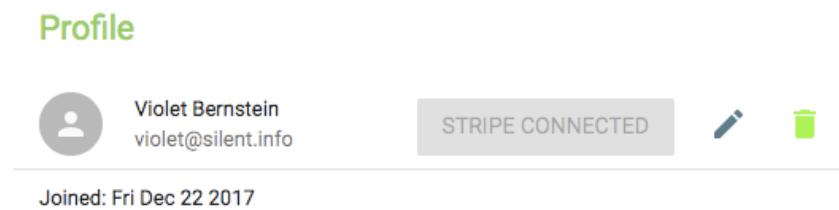
This `stripe_seller` field will store the seller's Stripe account credentials that were received from Stripe on authentication. This will be used when a charge needs to be processed via Stripe for a product they sold from their shop. Next, we will look at the frontend component that will allow the user to connect to Stripe from our application.

Button to connect with Stripe

In the user profile page of a seller, if the user has not connected their Stripe account yet, we will show a button that will take the user to Stripe to authenticate and connect their Stripe account. The Connect with Stripe button will be rendered in the Profile view as follows:



If the user has successfully connected their Stripe account already, we will show a disabled STRIPE CONNECTED button instead, as shown in the following screenshot:



The code that's added to the `Profile` component will check whether the user is a seller before rendering the Stripe-related button. Then, a second check will confirm whether Stripe credentials already exist in the `stripe_seller` field for the given user. If Stripe credentials already exist for the user, then the disabled `STRIPE CONNECTED` button is shown; otherwise, a link to connect to Stripe using their OAuth link is displayed instead, as implemented in the following code.

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

```
| {user.seller && (user.stripe_seller ?  
|   <Button variant="contained" disabled className=
```

```

{classes.stripe_connected}>
    Stripe connected
</Button>
: (<a href={"https://connect.stripe.com/oauth/authorize?
response_type=code&client_id="
+config.stripe_connect_test_client_id+"&scope=read_write"}
    className={classes.stripe_connect}>
    <img src={stripeButton}/>
    </a>
)
}

```

The OAuth link takes the platform's client ID, which we will set in a `config` variable, and other option values as query parameters. This link takes the user to Stripe and allows the user to connect an existing Stripe account or create a new one. Once Stripe's auth process has completed, it returns to our application using a redirect URL set in the platform's Connect settings in the dashboard on Stripe. Stripe attaches either an auth code or an error message as query parameters to the redirect URL.

The MERN Marketplace redirect URI is set to `/seller/stripe/connect`, which will render the `StripeConnect` component. We will declare this route as follows.

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

```
| <Route path="/seller/stripe/connect" component={StripeConnect}/>
```

When Stripe redirects the user to this URL, we will render the `stripeConnect` component so that it handles Stripe's response to authentication, as discussed in the next section.

The StripeConnect component

The `StripeConnect` component will basically complete the remaining auth process steps with Stripe and render the relevant messages based on whether the Stripe connection was successful, as shown in the following screenshot:



When the `StripeConnect` component loads, we will use a `useEffect` hook to parse the query parameters attached to the URL from the Stripe redirect, as shown in the following code.

`mern-marketplace/client/user/StripeConnect.js`:

```
useEffect(() => {
  const abortController = new AbortController()
  const signal = abortController.signal
  const jwt = auth.isAuthenticated()
  const parsed = queryString.parse(props.location.search)
  if(parsed.error){
    setValues({...values, error: true})
  }
  if(parsed.code){
    setValues({...values, connecting: true, error: false})
    //post call to stripe, get credentials and update user data
    stripeUpdate({
      userId: jwt.user._id
    }, {
      t: jwt.token
    }, parsed.code, signal).then((data) => {
      if (data.error) {
        setValues({...values, error: true, connected: false,
                  connecting: false})
      } else {
        setValues({...values, connected: true,
                  connecting: false, error: false})
      }
    })
  }
  return function cleanup() {
```

```
|     abortController.abort()  
| }  
, [])
```

For parsing, we use the same `query-string` node module that we used previously to implement a product search. Then, if the `URL query` parameter contains an `auth_code` and not an `error`, we make an API call in order to complete the Stripe OAuth from our server with the `stripeUpdate` `fetch` method.

The `stripeUpdate` `fetch` method is defined in `api-user.js` and passes the auth code retrieved from Stripe to an API we will set up in our server at `'/api/stripe_auth/:userId'`. This `stripeUpdate` `fetch` method is defined as follows.

mern-marketplace/client/user/api-user.js:

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

This fetch method is calling a backend API that we have to add on our server to complete the OAuth process and save the retrieved credentials to the database. We will implement this API in the next section.

The stripe auth update API

Once the Stripe account has been connected, to complete the OAuth process, we need to make a POST API call to Stripe OAuth from our server. We need to send the previously retrieved auth code to Stripe OAuth with the POST API call and receive the credentials to be stored in the seller's user account for processing charges. We will achieve this Stripe auth update by implementing an update API in the backend. This Stripe auth update API will receive a PUT request at `/api/stripe_auth/:userId` and initiate the POST API call to retrieve the credentials from Stripe.

The route for this Stripe auth update API will be declared on the server in user routes, as follows.

mern-marketplace/server/routes/user.routes.js:

```
| router.route('/api/stripe_auth/:userId')
|   .put(authCtrl.requireSignin, authCtrl.hasAuthorization,
|       userCtrl.stripe_auth, userCtrl.update)
```

A request to this route uses the `stripe_auth` controller method to retrieve the credentials from Stripe and passes it to the existing user update method so that it can be stored in the database.

To make a POST request to the Stripe API from our server, we will use the `request` node module, which needs to be installed with the following command from the command line:

```
| yarn add request
```

The `stripe_auth` controller method in the user controller will be defined as follows.

mern-marketplace/server/controllers/user.controller.js:

```
| const stripe_auth = (req, res, next) => {
|   request({
|     url: "https://connect.stripe.com/oauth/token",
|     method: "POST",
|     json: true,
|     body: { client_secret:config.stripe_test_secret_key,
|             code:req.body.stripe,
|             grant_type:'authorization_code'}
|   }, (error, response, body) => {
|     if(body.error){
|       return res.status('400').json({
|         error: body.error_description
|       })
|     }
|     req.body.stripe_seller = body
|     next()
|   })
| }
```

The POST API call to Stripe takes the platform's secret key and the retrieved auth code to complete the authorization. Then, it returns the credentials for the connected account in `body`, which is then appended to the request body so that the user's details can be updated in the `next()` call to the `update` controller method.

These auth credentials retrieved from Stripe can be used in our application to create charges on customer credit cards on behalf of the seller when they sell products from their shops. In the next section, we will learn how to collect the customer credit card details during checkout using Stripe.

Stripe Card Elements for checkout

During checkout, to collect credit card details from the user, we will use Stripe's `Card Elements` to add the credit card field to the checkout form. To integrate `Card Elements` with our React interface, we will utilize the `react-stripe-elements` node module, which can be installed by running the following command from the command line:

```
| yarn add react-stripe-elements
```

We will also need to inject the `Stripe.js` code into `template.js` to access Stripe in the frontend code, as shown here.

mern-marketplace/template.js:

```
| <script id="stripe-js" src="https://js.stripe.com/v3/"></script>
```

For the MERN Marketplace, Stripe will be required in the Cart view, where the `Checkout` component needs it to render `Card Elements` and process card detail input. We will wrap the `Checkout` component we added to `Cart.js` with the `StripeProvider` component from `react-stripe-elements` so that the `Elements` component in `Checkout` has access to the Stripe instance.

mern-marketplace/client/cart/Cart.js:

```
| <StripeProvider apiKey={config.stripe_test_api_key}>
|   <Checkout/>
| </StripeProvider>
```

Then, within the `Checkout` component, we will use Stripe's `Elements` component. Using Stripe's `Card Elements` will enable the application to collect the user's credit card details and use the Stripe instance to tokenize card information rather than handle it on our own servers.

The implementation details for this part of collecting the card details and generating the card token during the checkout process will be discussed in the *Integrating the checkout process* and *Creating a new order* sections. In the next section, we will discuss how to use Stripe to securely record the card details that will be received from a user with `Card Elements`.

Stripe Customer for recording card details

When an order is being placed at the end of the checkout process, the generated card token will be used to create or update a Stripe Customer (<https://stripe.com/docs/api#customers>) representing our user. This is a good way to store credit card information (<https://stripe.com/docs/saving-cards>) with Stripe for further use, such as for creating charges against specific products in the cart when a seller processes the ordered product from their shop. This eliminates the complications of having to store user credit card details securely on your own server. To integrate Stripe Customer with our application, in the following sections, we will update the user model so that it stores Stripe Customer details and update the user controller methods so that we can create or update Stripe Customer information using the Stripe node module in the backend.

Updating the user model

To use Stripe Customer to securely store the credit card information of each user and process payments as needed in the application, we need to store details of the Stripe Customer associated with each user. To keep track of the corresponding Stripe Customer information for a user in our database, we will update the user model with the following field:

```
| stripe_customer: {},
```

This field will store a Stripe Customer object that will allow us to create recurring charges and track multiple charges associated with the same user in our platform. To be able to create or update a Stripe Customer, we need to utilize Stripe's Customer API. In the next section, we will update the user controller so that we can integrate and use this Customer API from Stripe.

Updating the user controller

We will create a new, or update an existing, Stripe Customer when the user places an order after entering their credit card details. To implement this, we will update the user controllers with a `stripeCustomer` method that will be called before the order is created when our server receives a request to the create order API (as discussed in the *Creating a new order* section).

In the `stripeCustomer` controller method, we will need to use the `stripe` node module, which can be installed with the following command:

```
| yarn add stripe
```

After installing the `stripe` module, it needs to be imported into the user controller file. Then, the `stripe` instance needs to be initialized with the application's Stripe secret key.

mern-marketplace/server/controllers/user.controller.js:

```
| import stripe from 'stripe'  
| const myStripe = stripe(config.stripe_test_secret_key)
```

The `stripeCustomer` controller method will check whether the current user already has a corresponding Stripe Customer stored in the database, and then use the card token received from the frontend to either create a new Stripe Customer or update the existing one, as discussed in the following sections.

Creating a new Stripe Customer

If the current user does not have a corresponding Stripe Customer – in other words, a value is not stored for the `stripe_customer` field – we will use the create a customer API (https://stripe.com/docs/api#create_customer) from Stripe, as follows.

mern-marketplace/server/controllers/user.controller.js:

```
myStripe.customers.create({
    email: req.profile.email,
    source: req.body.token
}).then((customer) => {
    User.update({ '_id': req.profile._id },
        { '$set': { 'stripe_customer': customer.id } },
        (err, order) => {
            if (err) {
                return res.status(400).send({
                    error: errorHandler.getErrorMessage(err)
                })
            }
            req.body.order.payment_id = customer.id
            next()
        }
    )
})
```

If the Stripe Customer is successfully created, we will update the current user's data by storing the Stripe Customer ID reference in the `stripe_customer` field. We will also add this Customer ID to the order being placed so that it is simpler to create a charge related to the order. Once a Stripe Customer has been created, we can update the Stripe Customer the next time a user enters credit card details for a new order, as discussed in the next section.

Updating an existing Stripe Customer

For an existing Stripe Customer – in other words, where the current user already has a value stored for the `stripe_customer` field – we will use the Stripe API to update a Stripe Customer, as follows.

mern-marketplace/server/controllers/user.controller.js:

```
myStripe.customers.update(req.profile.stripe_customer, {
    source: req.body.token
},
  (err, customer) => {
  if(err){
    return res.status(400).send({
      error: "Could not update charge details"
    })
  }
  req.body.order.payment_id = customer.id
  next()
})
```

Once the Stripe Customer has been successfully updated, we will add the Customer ID to the order being created in the `next()` call. Though not covered here, the Stripe Customer feature can be used to allow users to store and update their credit card information from the application. With the user's payment information securely stored and accessible, we can look into how to use this information to process a payment when an ordered product is processed by the seller.

Creating a charge for each product that's processed

When a seller updates an order by processing the product that was ordered in their shop, the application will create a charge on behalf of the seller on the customer's credit card for the cost of the product ordered.

To implement this, we will update the `user.controller.js` file with a `createCharge` controller method that will use Stripe's create a charge API and needs the seller's Stripe account ID, along with the buyer's Stripe Customer ID. The `createCharge` controller method will be defined as follows.

mern-marketplace/server/controllers/user.controller.js:

```
const createCharge = (req, res, next) => {
  if(!req.profile.stripe_seller){
    return res.status('400').json({
      error: "Please connect your Stripe account"
    })
  }
  myStripe.tokens.create({
    customer: req.order.payment_id,
  }, {
    stripeAccount: req.profile.stripe_seller.stripe_user_id,
  }).then((token) => {
    myStripe.charges.create({
      amount: req.body.amount * 100, //amount in cents
      currency: "usd",
      source: token.id,
    }, {
      stripeAccount: req.profile.stripe_seller.stripe_user_id,
    }).then((charge) => {
      next()
    })
  })
}
```

If the seller has not connected their Stripe account yet, the `createCharge` method will return a 400 error response to indicate that a connected

Stripe account is required.

To be able to charge the Stripe Customer on behalf of the seller's Stripe account, we need to generate a Stripe token with the Customer ID and the seller's Stripe account ID and then use that token to create a charge.

The `createCharge` controller method will be called when the server receives a request to update an order with a product status change to **Processing** (the API implementation for this order update request will be discussed in the *Listing orders by shop* section).

This covers all the Stripe-related concepts that are relevant to the implementation of payment processing for the specific use cases of the MERN Marketplace. Now, we will continue with our implementations in order to allow a user to complete the checkout process and place their order from their shopping cart.

Integrating the checkout process

Users who are signed in and have items added to their cart will be able to start the checkout process. We will add a Checkout form to collect customer details, delivery address information, and credit card information, as shown in the following screenshot:

The screenshot shows a 'Checkout' form with the following fields:

- Name:** Jamie Woolcraft
- Email:** jamie@wool.info
- Delivery Address:**
 - Street Address: 1234 Panchita Ave
 - City: Milpitas
 - State: CA
 - Zip Code: 95035
 - Country: USA
- Card details:** A placeholder card number (4242 4242 4242 4242) and expiration date (05 / 23).
- PLACE ORDER** button.

This checkout view will consist of two parts, with the first part for collecting buyer details including name, email, and delivery address,

and the second part for entering credit card details and placing the order. In the following sections, we will complete the implementation of the checkout process by initializing the checkout form details and adding the fields for collecting buyer details. Then, we will collect the buyer's credit card details to allow them to place the order and finish the checkout process.

Initializing checkout details

In this section, we will create the checkout view, which contains the form fields and the place order option in a `Checkout` component. In this component, we will initialize the `checkoutDetails` object in the state before collecting the details from the form. We will prepopulate the customer details based on the current user's details and add the current cart items to `checkoutDetails`, as shown in the following code.

mern-marketplace/client/cart/Checkout.js:

```
const user = auth.isAuthenticated().user
const [values, setValues] = useState({
  checkoutDetails: {
    products: cart.getCart(),
    customer_name: user.name,
    customer_email: user.email,
    delivery_address: { street: '', city: '', state: '',
      zipcode: '', country: ''},
  },
  error: ''
})
```

These customer information values, which are initialized in `checkoutDetails`, will be updated when the user interacts with the form fields. In the following sections, we will add the form fields and the change-handling functions for the customer information and delivery address details to be collected in this checkout view.

Customer information

In the checkout form, we will have fields for collecting the customer's name and email address. To add these text fields to the `Checkout` component, we will use the following code.

mern-marketplace/client/cart/Checkout.js:

```
<TextField id="name" label="Name" value={values.checkoutDetails.customer_name} onChange={handleCustomerChange('customer_name')}/>
<TextField id="email" type="email" label="Email" value={values.checkoutDetails.customer_email} onChange={handleCustomerChange('customer_email')}/><br/>
```

When the user updates the values in these two fields, the `handleCustomerChange` method will update the relevant details in the state. The `handleCustomerChange` method is defined as follows.

mern-marketplace/client/cart/Checkout.js:

```
const handleCustomerChange = name => event => {
  let checkoutDetails = values.checkoutDetails
  checkoutDetails[name] = event.target.value || undefined
  setValues({...values, checkoutDetails: checkoutDetails})
}
```

This will allow the user to update the name and email of the customer that this order is associated with. Next, we will look at the implementation for collecting the delivery address details for this order.

Delivery address

To collect the delivery address from the user, we will add fields to collect address details such as the street address, city, state, zip code, and country name to the checkout form. We will use the following code to add the text fields to allow a user to enter these address details.

mern-marketplace/client/cart/Checkout.js:

```
<TextField id="street" label="Street Address" value={values.checkoutDetails.delivery_address.street} onChange={handleAddressChange('street')}/>
<TextField id="city" label="City" value={values.checkoutDetails.delivery_address.city} onChange={handleAddressChange('city')}/>
<TextField id="state" label="State" value={values.checkoutDetails.delivery_address.state} onChange={handleAddressChange('state')}/>
<TextField id="zipcode" label="Zip Code" value={values.checkoutDetails.delivery_address.zipcode} onChange={handleAddressChange('zipcode')}/>
<TextField id="country" label="Country" value={values.checkoutDetails.delivery_address.country} onChange={handleAddressChange('country')}/>
```

When the user updates these address fields, the `handleAddressChange` method will update the relevant details in the state, as follows.

mern-marketplace/client/cart/Checkout.js:

```
const handleAddressChange = name => event => {
  let checkoutDetails = values.checkoutDetails
  checkoutDetails.delivery_address[name] =
    event.target.value || undefined
  setValues({...values, checkoutDetails: checkoutDetails})
}
```

With these text fields and handle change functions in place, the `checkoutDetails` object in the state will contain the customer information and delivery address that was entered by the user. In the next section, we will collect payment information from the buyer and use it

with the other checkout details to complete the checkout process and place the order.

Placing an order

The remaining steps of the checkout process will involve collecting the user's credit card details securely, thus allowing the user to place the order, emptying the cart from storage, and redirecting the user to a view with the order details. We will implement these steps by building a `PlaceOrder` component that consists of the remaining elements in the checkout view, which are the credit card field and the place order button. In the following sections, as we develop this component, we will use Stripe Card Elements to collect credit card details, add a place order button for the user to complete the checkout process, utilize a cart helper method to empty the cart, and redirect the user to an order view.

Using Stripe Card Elements

In order to use Stripe's `CardElement` component from `react-stripe-elements` to add the credit card field to the `PlaceOrder` component, we need to wrap the `PlaceOrder` component using the `injectStripe` **higher-order component (HOC)** from Stripe.

This is because the `CardElement` component needs to be part of a payment form component that is built with `injectStripe` and also wrapped with the `Elements` component. So, when we create a component called `PlaceOrder`, we will wrap it with `injectStripe` before exporting it, as shown in the following code.

mern-marketplace/client/cart/PlaceOrder.js:

```
const PlaceOrder = (props) => { ... }
PlaceOrder.propTypes = {
  checkoutDetails: PropTypes.object.isRequired
}
export default injectStripe(PlaceOrder)
```

Then, we will add this `PlaceOrder` component in the Checkout form, pass it the `checkoutDetails` object as a prop, and wrap it with the `Elements` component from `react-stripe-elements`, as shown here.

mern-marketplace/client/cart/Checkout.js:

```
<Elements> <PlaceOrder checkoutDetails={values.checkoutDetails} />
</Elements>
```

The `injectStripe` HOC provides the `props.stripe` property that manages the `Elements` group. This will allow us to call `props.stripe.createToken` within `PlaceOrder` to submit card details to Stripe and get back the card token. Next, we will learn how to use the Stripe `CardElement` component to collect credit card details from within the `PlaceOrder` component.

The CardElement component

Stripe's `CardElement` is self-contained, so we can just add it to the `PlaceOrder` component, then incorporate styles as desired, and the card detail input will be taken care of. We will add the `CardElement` component to `PlaceOrder` as follows.

mern-marketplace/client/cart/PlaceOrder.js:

```
<CardElement className={classes.StripeElement}
    {...{style: {
      base: {
        color: '#424770',
        letterSpacing: '0.025em',
        '::placeholder': {
          color: '#aab7c4',
        },
      },
      invalid: {
        color: '#9e2146',
      },
    }}/>
```

This will render the credit card details field in the checkout form view. In the next section, we will learn how to securely validate and store the credit card details that are entered in this field when the user clicks on a button to place an order and complete the checkout process.

Adding a button to place an order

The final element in the checkout view is the Place Order button, which will complete the checkout process if all the details are entered correctly. We will add this button to the `PlaceOrder` component after `CardElement`, as shown in the following code.

mern-marketplace/client/cart/PlaceOrder.js:

```
| <Button color="secondary" variant="raised" onClick={placeOrder}>Place  
| Order</Button>
```

Clicking on the Place Order button will call the `placeOrder` method, which will attempt to tokenize the card details using `stripe.createToken`. If this is unsuccessful, the user will be informed of the error, but if this is successful, then the checkout details and generated card token will be sent to our server's create order API (covered in the next section). The `placeOrder` method is defined as follows.

mern-marketplace/client/cart/PlaceOrder.js:

```
const placeOrder = ()=>{  
    props.stripe.createToken().then(payload => {  
        if(payload.error){  
            setValues({...values, error: payload.error.message})  
        }else{  
            const jwt = auth.isAuthenticated()  
            create({userId:jwt.user._id}, {  
                t: jwt.token  
            }, props.checkoutDetails, payload.token.id).then((data) => {  
                if (data.error) {  
                    setValues({...values, error: data.error})  
                } else {  
                    cart.emptyCart(()=> {  
                        setValues({...values, 'orderId':data._id,'redirect': true})  
                    })  
                }  
            })  
        }  
    }
```

```
| })
```

The `create` fetch method that we invoked here to make a POST request to the create order API in the backend is defined in `client/order/api-order.js`. It takes the checkout details, the card token, and user credentials as parameters and sends them to the API, as seen in previous API implementations. When the new order is successfully created, we will also empty the cart in `localStorage`, as discussed in the next section.

Empty cart

If the request to the create order API is successful, we will empty the cart in `localStorage` so that the user can add new items to the cart and place a new order if desired. To empty the cart in browser storage, we will use the `emptyCart` helper method in `cart-helper.js`, which is defined as follows.

mern-marketplace/client/cart/cart-helper.js:

```
emptyCart(cb) {
  if(typeof window !== "undefined") {
    localStorage.removeItem('cart')
    cb()
  }
}
```

The `emptyCart` method removes the cart object from `localStorage` and updates the state of the view by executing the callback passed to it from the `placeOrder` method, where it is invoked. With the checkout process completed, we can now redirect the user out of the cart and checkout view, as discussed in the next section.

Redirecting to the order view

With the order placed and the cart emptied, we can redirect the user to the order view, which will show them the details of the order that was just placed. To implement this redirect, we can use the Redirect component from React Router, as shown in the following code.

mern-marketplace/client/cart/PlaceOrder.js:

```
| if (values.redirect) {  
|     return (<Redirect to={'/order/' + values.orderId}/>)  
| }
```

This redirection also works as an indication to the user that the checkout process has been completed. A completed checkout process will also result in a new order being created in the application's backend. In the next section, we will look into the implementation of creating and storing these new orders in the database.

Creating a new order

When a user places an order, the details of the order that were confirmed at checkout will be used to create a new order record in the database, update or create a Stripe Customer for the user, and decrease the stock quantities of products ordered. In the following sections, we will add an order model to define the details of the orders to be stored in the database and discuss the implementation of the backend API that will be called from the frontend to create the new order record.

Defining an Order model

To store the orders in the backend, we will define a Schema for the order model that will record order details including the customer details, payment information, and an array of the products ordered. The structure of each product in this array of products ordered will be defined in a separate subschema called `CartItemSchema`. In the following sections, we will define these schemas so that we can store orders and cart items in the database.

The Order schema

The Order schema defined in `server/models/course.model.js` will contain fields for storing the customer's name and email, along with their user account reference, delivery address information, payment reference, created and updated-at timestamps, and an array of products ordered. The pieces of code for defining the order fields are as follows:

- **Customer name and email:** To record the details of the customer who the order is meant for, we will add the `customer_name` and `customer_email` fields to the `Order` schema:

```
customer_name: { type: String, trim: true, required: 'Name is required' },
customer_email: { type: String, trim: true,
  match: [/^[\w\.-]+@[^\w\.-]+\.\w+$/, 'Please fill a valid email address'],
  required: 'Email is required' }
```

- **User who placed the order:** To reference the signed-in user who placed the order, we will add an `ordered_by` field:

```
ordered_by: {type: mongoose.Schema.ObjectId, ref: 'User'}
```

- **Delivery address:** The delivery address information for the order will be stored in the delivery address subdocument with the `street`, `city`, `state`, `zipcode`, and `country` fields:

```
delivery_address: {
  street: {type: String, required: 'Street is required'},
  city: {type: String, required: 'City is required'},
  state: {type: String},
  zipcode: {type: String, required: 'Zip Code is required'},
  country: {type: String, required: 'Country is required'}
},
```

- **Payment reference:** The payment information will be relevant when the order is updated and a charge needs to be created after an ordered product has been processed by the seller. We will record the Stripe Customer ID that's relevant to the credit

card details in a `payment_id` field as a reference to the payment information for this order:

```
|   payment_id: {},
```

- **Products ordered:** The main content of the order will be the list of products ordered, along with details such as the quantity of each. We will record this list in a field called `products` in the `Order` schema. The structure of each product will be defined separately in `CartItemSchema`.

mern-marketplace/server/models/order.model.js:

```
|   products: [CartItemSchema],
```

The fields in this schema definition will enable us to store the necessary details for each order. `CartItemSchema`, which is used to record the details of each product that was ordered, will be discussed in the next section.

The CartItem schema

The `CartItem` schema will represent each product that was ordered when an order was placed. It will contain a reference to the product, the quantity of the product that was ordered by the user, a reference to the shop the product belongs to, and its status, as shown in the following code.

mern-marketplace/server/models/order.model.js:

```
const CartItemSchema = new mongoose.Schema({
  product: {type: mongoose.Schema.ObjectId, ref: 'Product'},
  quantity: Number,
  shop: {type: mongoose.Schema.ObjectId, ref: 'Shop'},
  status: {type: String,
    default: 'Not processed',
    enum: ['Not processed', 'Processing', 'Shipped', 'Delivered',
    'Cancelled']}
})
const CartItem = mongoose.model('CartItem', CartItemSchema)
```

The `status` of the product can only have the values defined in the `enums`, with the default value set to "Not Processed". This represents the current state of the product order, as updated by the seller.

The `Order` schema and `CartItem` schema defined here will allow us to record details about the customer and products that were ordered as required to complete the purchase steps for the products that were bought by a user. Next, we will discuss the backend API implementation that allows the frontend to create an order document in the Orders collection in the database.

Create order API

The create order API in the backend will take a POST request from the frontend to create the order in the database. The API route will be declared in `server/routes/order.routes.js`, along with the other order routes. These order routes will be very similar to the user routes. To load the order routes in the Express app, we need to mount the routes in `express.js`, just like we did for the auth and user routes.

mern-marketplace/server/express.js:

```
| app.use('/', orderRoutes)
```

A number of actions, in the following sequence, take place when the create order API receives a POST request at `/api/orders/:userId`:

- It is ensured that the current user is signed in.
- A Stripe `Customer` is either created or updated using the `stripeCustomer` user controller method, which we discussed earlier in the *Stripe Customer to record card details* section.
- The stock quantities are updated for all the ordered products using the `decreaseQuantity` product controller method.
- The order is created in the Order collection with the `create` order controller method.

The route for this create order API is defined as follows.

mern-marketplace/server/routes/order.routes.js:

```
| router.route('/api/orders/:userId')
  .post(authCtrl.requireSignin, userCtrl.stripeCustomer,
        productCtrl.decreaseQuantity, orderCtrl.create)
```

To retrieve the user associated with the `:userId` parameter in the route, we will use the `userByID` user controller method. We will write the

code to handle this parameter in the route URL, along with the other order route declaration.

mern-marketplace/server/routes/order.routes.js:

```
| router.param('userId', userCtrl.userByID)
```

The `userByID` method gets the user from the User collection and attaches it to the request object so that it can be accessed by the next few methods. Among the next few methods that are invoked when this API receives a request includes the product controller method to decrease stock quantities and the order controller method to save a new order to the database. We will discuss the implementation of these two methods in the following sections.

Decrease product stock quantity

When an order is placed, we will decrease the stock quantity of each product ordered according to the quantity ordered by the user. This will automatically reflect the updated quantities of the products in the associated shops after an order is placed. We will implement this decrease product quantity update in the `decreaseQuantity` controller method, which will be added with the other product controller methods, as follows.

mern-marketplace/server/controllers/product.controller.js:

```
const decreaseQuantity = async (req, res, next) => {
  let bulkOps = req.body.order.products.map((item) => {
    return {
      "updateOne": {
        "filter": { "_id": item.product._id },
        "update": { "$inc": { "quantity": -item.quantity} }
      }
    }
  })
  try {
    await Product.bulkWrite(bulkOps, {})
    next()
  } catch (err){
    return res.status(400).json({
      error: "Could not update product"
    })
  }
}
```

Since the update operation, in this case, involves a bulk update of multiple products in the collection after matching with an array of products ordered, we use the `bulkWrite` method in MongoDB to send multiple `updateOne` operations to the MongoDB server with one command. The multiple `updateOne` operations that are required are listed in `bulkOps` using the `map` function. This will be faster than sending multiple independent save or update operations because with `bulkWrite()`, there is only one round trip to MongoDB.

Once the product quantities have been updated by this method, the next method is invoked to save the new order in the database. In the next section, we will see the implementation of this method, which creates this new order.

Create controller method

The `create` controller method, defined in the order controllers, is the last method that's invoked when the create order API receives a request. This method takes the order details, creates a new order, and saves it to the Order collection in MongoDB. The `create` controller method is implemented as follows.

mern-marketplace/server/controllers/order.controller.js:

```
const create = async (req, res) => {
  try {
    req.body.order.user = req.profile
    let order = new Order(req.body.order)
    let result = await order.save()
    res.status(200).json(result)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

With this implemented, orders can be created and stored in the backend by any signed-in user on the MERN Marketplace. Now, we can set up APIs to fetch lists of orders by user, orders by shop, or read an individual order and display the fetched data to views in the frontend. In the next section, we will learn how to list the orders per shop so that shop owners can process and manage the orders they receive for their products.

Listing orders by shop

An important feature of the marketplace is allowing sellers to see and update the status of orders they've received for products in their shops. To implement this, we will set up backend APIs to list orders by shop and update an order as a seller changes the status of a purchased product. Then, we will add some frontend views that will display the orders and allow the seller to interact with each order.

The list by shop API

In this section, we will implement an API to get orders for a specific shop so that authenticated sellers can view the orders for each of their shops in one place. The request for this API will be received at `/api/orders/shop/:shopId`, with the route defined in `order.routes.js`, as follows.

mern-marketplace/server/routes/order.routes.js:

```
| router.route('/api/orders/shop/:shopId')
|   .get(authCtrl.requireSignin, shopCtrl.isOwner, orderCtrl.listByShop)
| router.param('shopId', shopCtrl.shopByID)
```

To retrieve the shop associated with the `:shopId` parameter in the route, we will use the `shopByID` shop controller method, which gets the shop from the Shop collection and attaches it to the request object so that it can be accessed by the next methods.

The `listByShop` controller method will retrieve the orders that have products purchased with the matching shop ID, then populate the ID, name, and price fields for each product, with orders sorted by date from most recent to oldest. The `listByShop` controller method is defined as follows.

mern-marketplace/server/controllers/order.controller.js:

```
| const listByShop = async (req, res) => {
|   try {
|     let orders = await Order.find({ "products.shop": req.shop._id })
|       .populate({ path: 'products.product', select: '_id name price' })
|       .sort('-created')
|       .exec()
|     res.json(orders)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

To fetch this API in the frontend, we will add a corresponding `listByShop` method in `api-order.js`, similar to our other API implementations. This fetch method will be used in the `ShopOrders` component to show the orders for each shop. We will look at the implementation of the `ShopOrders` component in the next section.

The ShopOrders component

The `ShopOrders` component will be the view where sellers will be able to see the list of orders that have been received for a given shop. In this view, each order will only show the purchased products that are relevant to the shop and allow the seller to change the status of the ordered product with a dropdown of possible status values, as shown in the following screenshot:

Orders in Toyish

Order # 5a770fa5e96294d3a1208af7 ^
Sun Feb 04 2018

	Joker Figurine Quantity: 1	Update Status Not processed ▾
	Wonder Woman Figurine Quantity: 1	Not processed Processing Shipped Delivered Cancelled ▾
Deliver to: Jamie Woolcraft (jamie@wool.info) 1234 Panchita Ave Milpitas, CA 95035 USA		
Order # 5a2fdffce96294d3a1208afd Tue Dec 12 2017		

To render this view at a frontend route, we will update `MainRouter` with a `PrivateRoute` in order to load the `ShopOrders` component at the `/seller/orders/:shop/:shopId` route, as shown in the following code.

mern-marketplace/client/MainRouter.js:

```
| <PrivateRoute path="/seller/orders/:shop/:shopId" component=| {ShopOrders}/>
```

Going to this link will load the `ShopOrders` component in the view. In the `ShopOrders` component, we will fetch and list the orders for the given shop, and for each order, we'll render the order details along with the list of products that were ordered in a React component named `ProductOrderEdit`. In the following sections, we will learn how to load the list of orders and discuss the implementation of the `ProductOrderEdit` component.

List orders

When the `ShopOrders` component mounts in the view, we will retrieve the list of orders for the provided shop ID from the database and set it to the state to be rendered in the view. We will make a request to the backend API to list orders by shop using the `listByShop` fetch method and set the retrieved orders to the state in a `useEffect` hook, as shown in the following code.

mern-marketplace/client/order/ShopOrders.js:

```
useEffect(() => {
  const jwt = auth.isAuthenticated()
  const abortController = new AbortController()
  const signal = abortController.signal
  listByShop({
    shopId: match.params.shopId
  }, {t: jwt.token}, signal).then((data) => {
    if (data.error) {
      console.log(data)
    } else {
      setOrders(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

In the view, we will iterate through the list of orders and render each order in a collapsible list from `Material-UI`, which will expand when it's clicked. The code for this view will be added as follows.

mern-marketplace/client/order/ShopOrders.js:

```
<Typography type="title"> Orders in {match.params.shop} </Typography>
<List dense> {orders.map((order, index) => { return
  <span key={index}>
    <ListItem button onClick={handleClick(index)}>
      <ListItemText primary={'Order # '+order._id}
        secondary={(new Date(order.created)).toDateString()} />
      {open == index ? <ExpandLess /> : <ExpandMore />}
    </ListItem>
    <Collapse component="li" in={open == index}>
```

```

        timeout="auto" unmountOnExit>
      <ProductOrderEdit shopId={match.params.shopId}>
        order={order} orderIndex={index}
        updateOrders={updateOrders}/>
        <Typography type="subheading"> Deliver to:</Typography>
        <Typography type="subheading" color="primary">
          {order.customer_name} ({order.customer_email})
        </Typography>
        <Typography type="subheading" color="primary">
          {order.delivery_address.street}</Typography>
        <Typography type="subheading" color="primary">
          {order.delivery_address.city},
          {order.delivery_address.state}
          {order.delivery_address.zipcode}</Typography>
        <Typography type="subheading" color="primary">
          {order.delivery_address.country}</Typography>
      </Collapse>
    </span>))
  </List>

```

Each expanded order will show the order details and the `ProductOrderEdit` component. The `ProductOrderEdit` component will display the purchased products and allow the seller to edit the status of each product. The `updateOrders` method is passed as a prop to the `ProductOrderEdit` component so that the status can be updated when a product status is changed. The `updateOrders` method is defined as follows.

`mern-marketplace/client/order/ShopOrders.js`:

```

const updateOrders = (index, updatedOrder) => {
  let updatedOrders = orders
  updatedOrders[index] = updatedOrder
  setOrders([...updatedOrders])
}

```

In the `ProductOrderEdit` component, we will invoke this `updateOrders` method when the seller interacts with the status update dropdown for any product that will be rendered in the `ProductOrderEdit` component. In the next section, we will look into the implementation of this `ProductOrderEdit` component.

The ProductOrderEdit component

In this section, we will implement a `ProductOrderEdit` component to render all the products in the order with an edit status option. This `ProductOrderEdit` component will take an order object as a prop and iterate through the order's `products` array to display only the products that have been purchased from the current shop, along with a dropdown to change the status value of each product. The code for this view, which renders the products for each order, will be added as follows.

mern-marketplace/client/order/ProductOrderEdit.js:

```
{props.order.products.map((item, index) => { return <span key={index}>
    { item.shop == props.shopId &&
        <ListItem button>
            <ListItemText primary={ <div>
                <img src=
                    {'/api/product/image/' + item.product._id}/>
                {item.product.name}
                <p>{"Quantity: " + item.quantity}</p>
            </div>}>
            <TextField id="select-status" select
                label="Update Status" value={item.status}
                onChange={handleStatusChange(index)}
                SelectProps={{
                    MenuProps: { className: classes.menu },
                }}>
                {statusValues.map(option => (
                    <MenuItem key={option} value={option}>
                        {option}
                    </MenuItem>
                )))
            </TextField>
        </List Item>}
```

To be able to list the valid status values in the dropdown option for updating an ordered product's status, we will retrieve the list of possible status values from the server in a `useEffect` hook in the `ProductOrderEdit` component, as shown in the following code.

mern-marketplace/client/order/ProductOrderEdit.js:

```
useEffect(() => {
  const abortController = new AbortController()
  const signal = abortController.signal
  getStatusValues(signal).then((data) => {
    if (data.error) {
      setValues({...values, error: "Could not get status"})
    } else {
      setValues({...values, statusValues: data, error: ''})
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

The status values that are retrieved from the server are set to state and rendered in the dropdown as a `MenuItem`. When an option is selected from the possible status values in the dropdown, the `handleStatusChange` method is called to update the orders in the state, as well as to send a request to the appropriate backend API based on the value that's selected. The `handleStatusChange` method will be structured as follows, with a different backend API invoked, depending on the selected status value.

mern-marketplace/client/order/ProductOrderEdit.js:

```
const handleStatusChange = productIndex => event => {
  let order = props.order
  order.products[productIndex].status = event.target.value
  let product = order.products[productIndex]

  if (event.target.value == "Cancelled") {
    // 1. ... call the cancel product API ...
  } else if (event.target.value == "Processing") {
    // 2. ... call the process charge API ...
  } else {
    // 3. ... call the order update API ...
  }
```

Updating the status of an ordered product will have different implications, depending on the value that's selected from the dropdown. Selecting to cancel or process a product order will invoke separate APIs in the backend rather than the API called when selecting any of the other status values. In the following sections, we

will learn how each of these actions is handled in the `handleStatusChange` method when a user interacts with the dropdown and selects a status value.

Handling actions to cancel a product order

If the seller wishes to cancel the order for a product and selects Cancelled from the status values dropdown for a specific product in the order, we will call the `cancelProduct` fetch method inside the `handleStatusChange` method, as shown in the following code.

mern-marketplace/client/order/ProductOrderEdit.js:

```
cancelProduct({
    shopId: props.shopId,
    productId: product.product._id
}, {
    t: jwt.token
}, {
    cartItemId: product._id,
    status: event.target.value,
    quantity: product.quantity
})
.then((data) => {
    if (data.error) {
        setValues({
            ...values,
            error: "Status not updated, try again"
        })
    } else {
        props.updateOrders(props.orderIndex, order)
        setValues({
            ...values,
            error: ''
        })
    }
})
```

The `cancelProduct` fetch method will take the corresponding shop ID, product ID, cartItem ID, selected status value, ordered quantity for the product, and user credentials to send, along with the request to the cancel product API in the backend. On a successful response from the backend, we will update the orders in the view.

This cancel product API will update the database for the order and the product affected by this action. Before getting into the implementation for this cancel product order API, next, we will look at how the process charge API is invoked if the seller chooses to process a product order instead of canceling it.

Handling the action to process charge for a product

If a seller chooses to process the order for a product, we will need to invoke an API that will charge the customer for the total cost of the product ordered. So, when a seller selects Processing from the status values dropdown for a specific product in the order, we will call the `processCharge` fetch method inside the `handleStatusChange` method, as shown in the following code.

mern-marketplace/client/order/ProductOrderEdit.js:

```
processCharge({
    userId: jwt.user._id,
    shopId: props.shopId,
    orderId: order._id
}, {
    t: jwt.token
}, {
    cartItemId: product._id,
    status: event.target.value,
    amount: (product.quantity * product.product.price)
})
.then((data) => {
    if (data.error) {
        setValues({
            ...values,
            error: "Status not updated, try again"
        })
    } else {
        props.updateOrders(props.orderIndex, order)
        setValues({
            ...values,
            error: ''
        })
    }
})
```

The `processCharge` fetch method will take the corresponding order ID, shop ID, customer's user ID, cartItem ID, selected status value, total cost for the ordered product, and user credentials to send, along with the request to the process charge API in the backend. On a

successful response from the backend, we will update the orders in the view accordingly.

This process charge API will update the database for the order and the user affected by this action. Before getting into the implementation for this API, next, we will look at how the update order API is invoked if the seller chooses to update the status of a product that's been ordered to any value other than Cancelled or Processing.

Handling the action to update the status of a product

If a seller chooses to update the status of an ordered product so that it has a value other than Cancelled or Processing, we will need to invoke an API that will update the order in the database with this changed product status. So, when a seller selects other status values from the dropdown for a specific product in the order, we will call the `update` fetch method inside the `handleStatusChange` method, as shown in the following code.

mern-marketplace/client/order/ProductOrderEdit.js:

```
update({
    shopId: props.shopId
}, {
    t: jwt.token
}, {
    cartItemId: product._id,
    status: event.target.value
})
.then((data) => {
    if (data.error) {
        setValues({
            ...values,
            error: "Status not updated, try again"
        })
    } else {
        props.updateOrders(props.orderIndex, order)
        setValues({
            ...values,
            error: ''
        })
    }
})
```

The `update` fetch method will take the corresponding shop ID, cartItem ID, selected status value, and user credentials to send, along with the request to the update order API in the backend. On a successful response from the backend, we will update the orders in the view.

The `cancelProduct`, `processCharge`, and `update` `fetch` methods are defined in `api-order.js` so that they can call the corresponding APIs in the backend to update a canceled product's stock quantity, to create a charge on the customer's credit card when the order for a product is processing, and to update the order with the product status change, respectively. We will look at the implementation of these APIs in the following section.

APIs for products ordered

Allowing sellers to update the status of a product will require having to set up four different APIs, including an API to retrieve the possible status values. Then, the actual status update actions will need APIs to handle updates to the order itself as the status is changed in order to initiate related actions, such as increasing the stock quantity of a canceled product, and to create a charge on the customer's credit card when a product is being processed. In the following sections, we will look at the API implementations for retrieving possible status values, updating an order status, canceling a product order, and processing a charge for an ordered product.

Get status values

The possible status values of an ordered product are set as enums in the `CartItem` schema. To show these values as options in the dropdown view, we will set up a GET API route at `/api/order/status_values` that retrieves these values. This API route will be declared as follows.

mern-marketplace/server/routes/order.routes.js:

```
| router.route('/api/order/status_values')
|   .get(orderCtrl.getStatusValues)
```

The `getStatusValues` controller method will return the enum values for the `status` field from the `CartItem` schema. The `getStatusValues` controller method is defined as follows.

mern-marketplace/server/controllers/order.controller.js:

```
| const getStatusValues = (req, res) => {
|   res.json(CartItem.schema.path('status').enumValues)
| }
```

We will also need to set up a corresponding `fetch` method in `api-order.js`, which is used in the view, in the `ProductOrderEdit` component, to make a request to this API, retrieve the status values, and render these as options in the dropdown. In the next section, we will look at the update order API endpoint, which needs to be called when the seller selects a relevant status value from the dropdown.

Update order status

When a product's status is changed to any value other than **Processing** or **Cancelled**, a PUT request to

'/api/order/status/:shopId' will directly update the order in the database, given that the current user is the verified owner of the shop with the ordered product. We will declare the route for this update API like so.

mern-marketplace/server/routes/order.routes.js:

```
| router.route('/api/order/status/:shopId')
|   .put(authCtrl.requireSignin, shopCtrl.isOwner, orderCtrl.update)
```

The `update` controller method will query the Order collection and find the order with the `CartItem` object that matches the updated product and set the `status` value of this matched `CartItem` in the `products` array of the order. The `update` controller method is defined as follows.

mern-marketplace/server/controllers/order.controller.js:

```
| const update = async (req, res) => {
|   try {
|     let order = await Order.updateOne({'products._id': req.body.cartItemId}, {
|       'products.$status': req.body.status
|     })
|     res.json(order)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

To access this API from the frontend, we will add an `update` fetch method in `api-order.js` to make a call to this update API with the required parameters passed from the view. The `update` fetch method will be defined as follows.

mern-marketplace/client/order/api-order.js:

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

This `update` fetch method is called in the `ProductOrderEdit` view when the seller selects any value other than Processing or Cancelled from the options in the dropdown for an ordered product. In the next section, we will look at the cancel product order API, which is invoked if the seller selects Cancelled as a value instead.

Cancel product order

When a seller decides to cancel the order for a product, a PUT request will be sent to `/api/order/:shopId/cancel/:productId` so that the product's stock quantity can be increased and the order can be updated in the database. To implement this cancel product order API, we will declare the API route as follows.

mern-marketplace/server/routes/order.routes.js:

```
| router.route('/api/order/:shopId/cancel/:productId')
|   .put(authCtrl.requireSignin, shopCtrl.isOwner,
|         productCtrl.increaseQuantity, orderCtrl.update)
| router.param('productId', productCtrl.productByID)
```

To retrieve the product associated with the `productId` parameter in the route, we will also use the `productByID` product controller method. This will retrieve the product and attach it to the request object for the `next` methods.

To update the product's stock quantity when this API receives a request, we will use the `increaseQuantity` controller method, which is added to `product.controller.js`, as follows.

mern-marketplace/server/controllers/product.controller.js:

```
| const increaseQuantity = async (req, res, next) => {
|   try {
|     await Product.findByIdAndUpdate(req.product._id,
|       {$inc: {"quantity": req.body.quantity}}, {new: true})
|     .exec()
|     next()
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

The `increaseQuantity` controller method finds the product by the matching ID in the Product collection and increases the quantity value by the quantity that was ordered by the customer. It does this now that the order for this product has been canceled.

From the view, we will use the corresponding fetch method, which is added in `api-order.js`, to call this cancel product order API. The `cancelProduct` fetch method is defined as follows.

mern-marketplace/client/order/api-order.js:

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

This `cancelProduct` fetch method is called in the `ProductOrderEdit` view when the seller selects Cancelled from the dropdown for an ordered product. In the next section, we will look at the process charge API, which is invoked if the seller selects Processing as a status value instead.

Process charge for a product

When a seller changes the status of an ordered product to **Processing**, we will set up a backend API to not only update the order but to also create a charge on the customer's credit card for the price of the product multiplied by the quantity ordered. The route for this API will be declared as follows.

mern-marketplace/server/routes/order.routes.js:

```
router.route('/api/order/:orderId/charge/:userId/:shopId')
    .put(authCtrl.requireSignin, shopCtrl.isOwner,
        userCtrl.createCharge, orderCtrl.update)
router.param('orderId', orderCtrl.orderByID)
```

To retrieve the order associated with the `orderId` parameter in the route, we will use the `orderByID` order controller method, which gets the order from the Order collection and attaches it to the request object so that it can be accessed by the `next` methods. This `orderByID` method is defined as follows.

mern-marketplace/server/controllers/order.controller.js:

```
const orderByID = async (req, res, next, id) => {
  try {
    let order = await Order.findById(id)
      .populate('products.product', 'name price')
      .populate('products.shop', 'name').exec()
    if (!order)
      return res.status('400').json({
        error: "Order not found"
      })
    req.order = order
    next()
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The process charge API will receive a PUT request at `/api/order/:orderId/charge/:userId/:shopId`. After successfully authenticating the user, it will create the charge by calling the `createCharge` user controller, as we discussed in the *Using Stripe for payments* section. Finally, the corresponding order will be updated with the `update` controller method, as discussed in the *Update order status* section.

From the view, we will use the `processCharge` fetch method in `api-order.js` and provide the required route parameter values, credentials, and product details, including the amount to charge.

The `processCharge` fetch method is defined as follows.

mern-marketplace/client/order/api-order.js:

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

This `processCharge` fetch method is called in the `ProductOrderEdit` view when the seller selects Processing from the dropdown for an ordered product.

With these implementations in place, sellers can view orders that have been received for their products in each of their shops and easily update the status of each product ordered while the application takes care of additional tasks, such as updating stock quantity and initiating payment. This covers the basic order management features for the MERN Marketplace application, which

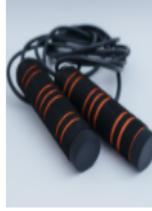
can be extended further as required. In the next section, we will discuss how the current implementations can be easily extended to implement other views for displaying order details.

Viewing single-order details

With the Order collection and the database access all set up, moving forward, it is easy to add the features of listing orders for each user and showing details of a single order in a separate view where the user can track the status of each ordered product. A view to render the details of a single order to the customer can be designed and implemented to look as follows:

Order Details

Order Code: 5a770fa5e96294d3a1208af7
Placed on Sun Feb 04 2018

	Joker Figurine	\$ 51.45 x 1	\$51.45
		Shop: Toyish	
		Status: Processing	
	Wonder Woman Figurine	\$ 11.45 x 1	\$11.45
		Shop: Toyish	
		Status: Cancelled	
	Speed Skipping Rope	\$ 7.55 x 2	\$15.1
		Shop: Athletico	
		Status: Not processed	

Deliver to:

Jamie Woolcraft
jamie@wool.info

1234 Panchita Ave
Milpitas, CA 95035
USA

*Thank you for shopping with us!
You can track the status of your purchased
items on this page.*

Total: \$66.55

Following the steps that have been repeated throughout this book to set up backend APIs to retrieve data and use it in the frontend to construct frontend views, you can develop order-related views as desired. For example, a view to display the orders that have been placed by a single user can be rendered as follows:

The screenshot shows a light gray card with a title 'Your Orders' at the top. Below the title, there are three horizontal entries, each representing an order. Each entry consists of an order ID, a date, and a horizontal line below it.

Order #	Date
5a770fa5e96294d3a1208af7	Sun Feb 04 2018
5a4e3236e96294d3a1208afb	Thu Jan 04 2018
5a2fdffce96294d3a1208afd	Tue Dec 12 2017

You can apply the lessons you learned while building out the full-stack features of the MERN Marketplace application to implement these order detail views, taking inspiration from the snapshots of these sample views from the MERN Marketplace application.

The MERN Marketplace application that we developed in this chapter and [Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*, by building on the MERN skeleton application covered the crucial features for a standard online marketplace application. This, in turn, demonstrated how the MERN stack can be extended to incorporate complex features.

Summary

In this chapter, we extended the MERN Marketplace application and explored how to add a shopping cart for buyers, a checkout process with credit card payments, and order management for the sellers in an online marketplace application.

We discovered how the MERN stack technologies can work well with third-party integrations as we implemented the cart checkout flow and processed credit card charges on ordered products using the tools provided by Stripe for managing online payments.

We also unlocked more of what is possible with MERN, such as optimized bulk write operations in MongoDB for updating multiple documents in response to a single API call. This allowed us to decrease the stock quantities of multiple products in one go, such as when a user placed an order for multiple products from different stores.

With these new approaches and implementations that we explored, you can easily integrate payment processing, use offline storage in browsers, and perform bulk database operations for any MERN-based application you choose to build.

The marketplace features that you developed in the MERN Marketplace application revealed how this stack and structure can be utilized to design and build growing applications by adding features that may be simple or more complex in nature.

In the next chapter, we will take the lessons we've learned so far in this book and explore more advanced possibilities with this stack by extending this MERN Marketplace application so that it incorporates real-time bidding capabilities.

Adding Real-Time Bidding Capabilities to the Marketplace

In a world more connected than ever before, instant communication and real-time updates are expected behaviors in any application that enables interaction between users. Adding real-time features to your application can keep your users engaged, and because of that, they will be spending more time on your platform. In this chapter, we will learn how to use the MERN stack technologies, along with Socket.IO, to easily integrate real-time behavior in a full-stack application. We will do this by incorporating an auctioning feature with real-time bidding capabilities in the MERN Marketplace application that we developed in [Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*, and [Chapter 8](#), *Extending the Marketplace for Orders and Payments*. After going through the implementation of this auction and bidding feature, you will know how to utilize sockets in a MERN stack application to add real-time features of your choice.

In this chapter, we will extend the online marketplace application by covering the following topics:

- Introducing real-time bidding in the MERN Marketplace
- Adding auctions to the marketplace
- Displaying the auction view
- Implementing real-time bidding with Socket.IO

Introducing real-time bidding in the MERN Marketplace

The MERN Marketplace application already allows its users to become sellers and maintain shops with products that can be bought by regular users. In this chapter, we will extend these functionalities to allow sellers to create auctions for items that other users can place bids on in a fixed duration of time. The auction view will describe the item for sale and let signed in users place bids when the auction is live. Different users can place their own bids, and also see other users placing bids in real-time, with the view updating accordingly. The completed auction view, with an auction in a live state, will render as follows:

Rutilated Quartz Ring

Auction Live



1 h 50 m 19 s left (ends at 1/16/2020, 4:14:00 PM)

Last bid: \$ 28

Your Bid (\$)

29

Enter \$29 or more

PLACE BID

About Item

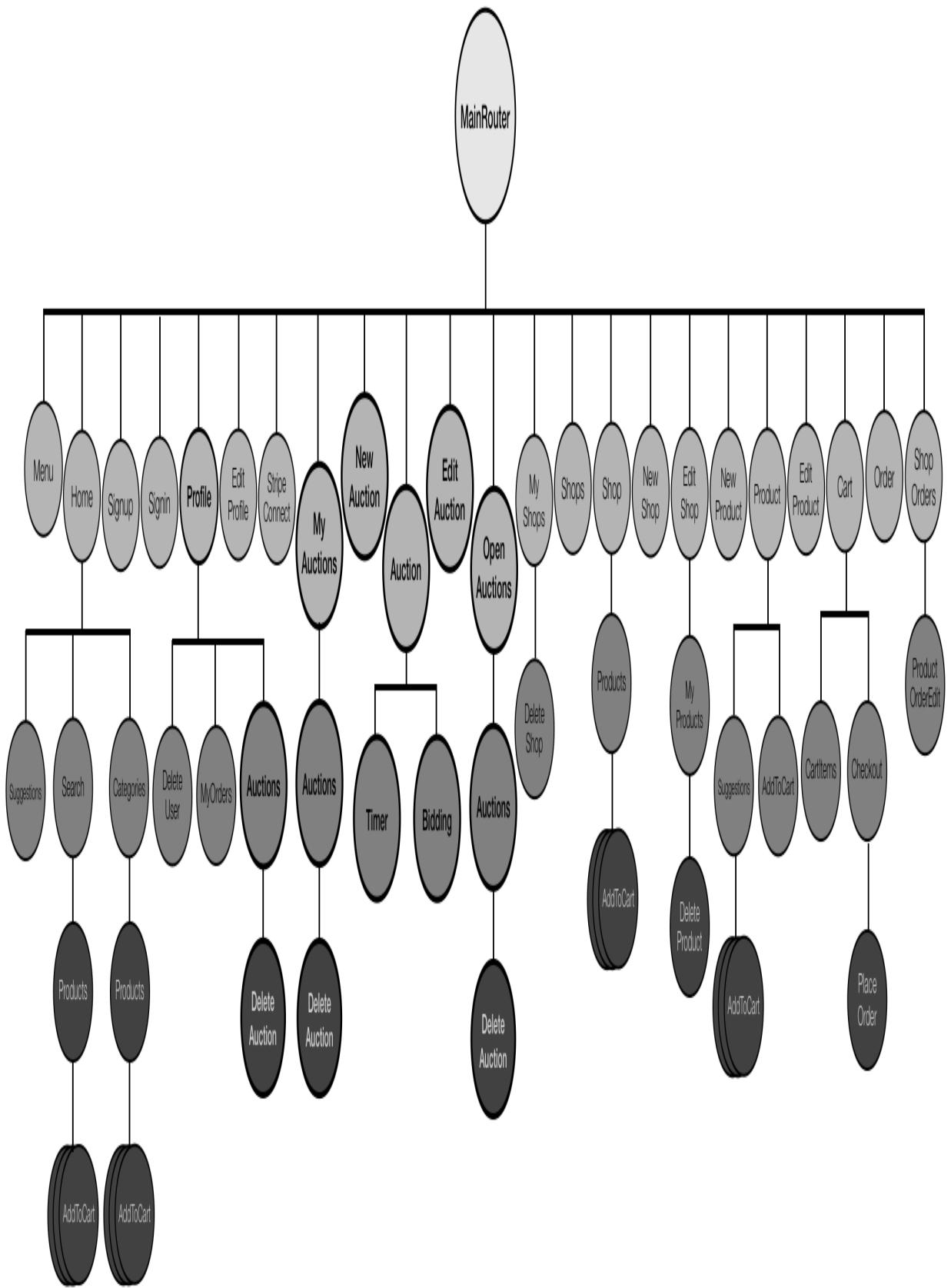
Handmade with love from pure sterling silver and genuine golden-brown Rutilated Quartz gemstone in an oval shape.

All bids

Bid Amount	Bid Time	Bidder
\$28	1/16/2020, 2:21:15 PM	Mary
\$26	1/16/2020, 2:20:11 PM	Sarah Atkin
\$24	1/16/2020, 2:19:46 PM	Jane
\$23	1/16/2020, 2:18:37 PM	Sarah Atkin

 *The code for the complete MERN Marketplace application is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter09/mern-marketplace-bidding>. The implementations discussed in this chapter can be accessed in the bidding branch of the repository. You can clone this code and run the application as you go through the code explanations in the rest of this chapter.*

The following component tree diagram shows the custom components that make up the entire MERN Marketplace frontend, including components for the auction and bidding-related features that will be implemented in the rest of this chapter:



The features that will be discussed in this chapter modify some of the existing components, such as `Profile` and `Menu`, and also add new components, such as `NewAuction`, `MyAuctions`, `Auction`, and `Bidding`. In the next section, we will begin extending this online marketplace by integrating the option to add auctions to the platform.

Adding auctions to the marketplace

In the MERN Marketplace, we will allow a user who is signed in and has an active seller account to create auctions for items that they want other users to place bids on. To enable the features of adding and managing auctions, we will need to define how to store auction details and implement the full-stack slices that will let users create, access and update auctions on the platform. In the following sections, we will build out this auction module for the application. First, we will define the auction model with a Mongoose Schema for storing details about each auction. Then, we will discuss implementations for the backend APIs and frontend views that are needed to create new auctions, list auctions that are ongoing, created by the same seller and bid on by the same user, and modify existing auctions by either editing details of, or deleting an auction from the application.

Defining an Auction model

We will implement a Mongoose model that will define an Auction model for storing the details of each auction. This model will be defined in `server/models/auction.model.js`, and the implementation will be similar to other Mongoose model implementations we've covered in previous chapters, such as the Shop model we defined in [Chapter 7, Exercising MERN Skills with an Online Marketplace](#). The Auction Schema in this model will have fields to store auction details such as the name and description of the item being auctioned, along with an image and a reference to the seller creating this auction. It will also have fields that specify the start and end time for bidding on this auction, a starting value for bids, and the list of bids that have been placed for this auction. The code for defining these auction fields is as follows:

- **Item name and description:** The auction item name and description fields will be string types, with `itemName` as a required field:

```
itemName: {  
    type: String,  
    trim: true,  
    required: 'Item name is required'  
,  
description: {  
    type: String,  
    trim: true  
,
```

- **Item image:** The `image` field will store the image file representing the auction item so that it can be uploaded by the user and stored as data in the MongoDB database:

```
image: {  
    data: Buffer,  
    contentType: String  
,
```

- **Seller:** The `seller` field will reference the user who is creating the auction:

```
    seller: {
      type: mongoose.Schema.ObjectId,
      ref: 'User'
    },
```

- **Created and updated at times:** The `created` and `updated` fields will be `Date` types, with `created` generated when a new auction is added, and `updated` changed when any auction details are modified:

```
    updated: Date,
    created: {
      type: Date,
      default: Date.now
    },
```

- **Bidding start time:** The `bidStart` field will be a `Date` type that will specify when the auction goes live so that users can start placing bids:

```
    bidStart: {
      type: Date,
      default: Date.now
    },
```

- **Bidding end time:** The `bidEnd` field will be a `Date` type that will specify when the auction ends, after which the users cannot place bids on this auction:

```
    bidEnd: {
      type: Date,
      required: "Auction end time is required"
    },
```

- **Starting bid:** The `startingBid` field will store values of the `Number` type, and it will specify the starting price for this auction:

```
    startingBid: {
      type: Number,
      default: 0
    },
```

- **List of bids:** The `bids` field will be an array containing details of each bid placed against the auction. When we store bids in this array, we will push the latest bid to the beginning of the array. Each bid will contain a reference to the user placing the bid, the bid amount the user offered, and the timestamp when the bid was placed:

```
bids: [{  
  bidder: {type: mongoose.Schema.ObjectId, ref: 'User'},  
  bid: Number,  
  time: Date  
}]
```

These auction-related fields will allow us to implement auction and bidding-related features for the MERN Marketplace application. In the next section, we will start developing these features by implementing the full-stack slice, which will allow sellers to create new auctions.

Creating a new auction

For a seller to be able to create a new auction on the platform, we will need to integrate a full-stack slice that allows the user to fill out a form view in the frontend, and then save the entered details to a new auction document in the database in the backend. To implement this feature, in the following sections, we will add a create auction API in the backend, along with a way to fetch this API in the frontend, and a create new auction form view that takes user input for auction fields.

The create auction API

For the implementation of the backend API, which will allow us to create a new auction in the database, we will declare a POST route, as shown in the following code.

mern-marketplace/server/routes/auction.routes.js:

```
| router.route('/api/auctions/by/:userId')
|   .post(authCtrl.requireSignin, authCtrl.hasAuthorization,
|         userCtrl.isSeller, auctionCtrl.create)
```

A POST request to this route at `/api/auctions/by/:userId` will ensure the requesting user is signed in and is also authorized. In other words, it is the same user associated with the `:userId` specified in the route param. Then, before creating the auction, it is checked if this given user is a seller using the `isSeller` method that's defined in the user controller methods.

To process the `:userId` parameter and retrieve the associated user from the database, we will utilize the `userByID` method from the user controller methods. We will add the following to the `Auction` routes in `auction.routes.js` so that the user is available in the `request` object as `profile`.

mern-marketplace/server/routes/auction.routes.js:

```
| router.param('userId', userCtrl.userByID)
```

The `auction.routes.js` file, which contains the auction routes, will be very similar to the `user.routes` file. To load these new auction routes in the Express app, we need to mount the auction routes in `express.js`, as we did for the auth and user routes.

mern-marketplace/server/express.js:

```
| app.use('/', auctionRoutes)
```

The `create` method in the auction controller, which is invoked after a seller is verified, uses the `formidable` node module to parse the multipart request that may contain an image file uploaded by the user for the item image. If there is a file, `formidable` will store it temporarily in the filesystem, and we will read it using the `fs` module to retrieve the file type and data so that we can store it in the `image` field in the auction document.

The `create` controller method will look as follows.

mern-marketplace/server/controllers/auction.controller.js:

```
const create = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Image could not be uploaded"
      })
    }
    let auction = new Auction(fields)
    auction.seller= req.profile
    if(files.image){
      auction.image.data = fs.readFileSync(files.image.path)
      auction.image.contentType = files.image.type
    }
    try {
      let result = await auction.save()
      res.status(200).json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}
```

The item image file for the auction is uploaded by the user and stored in MongoDB as data. Then, in order to be shown in the views, it is retrieved from the database as an image file at a separate GET API. The GET API is set up as an Express route at `/api/auctions/image/:auctionId`, which gets the image data from MongoDB and sends it as a file in the response. The implementation

steps for file upload, storage, and retrieval are outlined in detail in the *Upload profile photo* section in [Chapter 5](#), *Growing the Skeleton into a Social Media Application*.

This create auction API endpoint can now be used in the frontend to make a POST request. Next, we will add a fetch method on the client-side to make this request from the application's client interface.

Fetching the create API in the view

In the frontend, to make a request to this create API, we will set up a `fetch` method on the client-side to make a POST request to the API route and pass it the multipart form data containing details of the new auction in the `body`. This fetch method will be defined as follows.

mern-marketplace/client/auction/api-auction.js:

```
const create = (params, credentials, auction) => {
  return fetch('/api/auctions/by/' + params.userId, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Authorization': 'Bearer ' + credentials.t
    },
    body: auction
  })
  .then((response) => {
    return response.json()
  }).catch((err) => console.log(err))
}
```

The response that's received from the server will be returned to the component calling this fetch method. We will use this method in the new auction form view to send the user-entered auction details to the backend and create a new auction in the database. In the next section, we will implement this new auction form view in a React component.

The NewAuction component

Sellers in the marketplace application will interact with a form view to enter details of a new auction and create the new auction. We will render this form in the `NewAuction` component, which will allow a seller to create an auction by entering an item name and description, uploading an image file from their local filesystem, specifying the starting bid value, and creating date-time values for starting and ending bidding on this auction.

This form view will render as follows:

New Auction

UPLOAD IMAGE

Item Name

Description

Starting Bid (\$)

0

Auction Start Time

01/17/2020, 02:27 PM

Auction End Time

01/17/2020, 03:27 PM

SUBMIT

CANCEL

The implementation for this `NewAuction` component is similar to other create form implementations that we have discussed previously, such as the `NewShop` component implementation from [Chapter 7, Exercising MERN Skills with an Online Marketplace](#). The fields that are different in this form component are the date-time input options for the auction start and end timings. To add these fields, we'll use Material-UI `TextField` components with `type` set to `datetime-local`, as shown in the following code.

`mern-marketplace/client/auction/NewAuction.js`:

```

<TextField
  label="Auction Start Time"
  type="datetime-local"
  defaultValue={defaultStartTime}
  onChange={handleChange('bidStart')}>
/>
<TextField
  label="Auction End Time"
  type="datetime-local"
  defaultValue={defaultEndTime}
  onChange={handleChange('bidEnd')}>
/>

```

We also assign default date-time values for these fields in the format expected by this input component. We set the default start time to the current date-time and the default end time to an hour after the current date-time, as shown here.

mern-marketplace/client/auction/NewAuction.js:

```

const currentDate = new Date()
const defaultStartTime = getDateString(currentDate)
const defaultEndTime = getDateString(new
  Date(currentDate.setHours(currentDate.getHours() + 1)))

```

The `TextField` with the type as `datetime-local` takes dates in the format `yyyy-mm-ddThh:mm`. So, we define a `getDateString` method that takes a JavaScript date object and formats it accordingly. The `getDateString` method is implemented as follows.

mern-marketplace/client/auction/NewAuction.js:

```

const getDateString = (date) => {
  let year = date.getFullYear()
  let day = date.getDate().toString().length === 1 ? '0' + date.getDate()
    : date.getDate()
  let month = date.getMonth().toString().length === 1 ? '0' +
    (date.getMonth() + 1) : date.getMonth() + 1
  let hours = date.getHours().toString().length === 1 ? '0' +
    date.getHours() : date.getHours()
  let minutes = date.getMinutes().toString().length === 1 ? '0' +
    date.getMinutes() : date.getMinutes()
  let dateString = `${year}-${month}-${day}T${hours}:${minutes}`
  return dateString
}

```

In order to ensure the user has entered the dates correctly, with the start time set to a value before the end time, we need to add a check

before submitting the form details to the backend. The validation of the date combination can be confirmed with the following code.

mern-marketplace/client/auction/NewAuction.js:

```
| if(values.bidEnd < values.bidStart) {  
|     setValues({...values, error: "Auction cannot end before it starts"})  
| }
```

If the date combination is found to be invalid, then the user will be informed and form data will not be sent to the backend.

This `NewAuction` component can only be viewed by a signed-in user who is also a seller. Therefore, we will add a `PrivateRoute` in the `MainRouter` component. This will render this form for authenticated users at `/auction/new`.

mern-marketplace/client/MainRouter.js:

```
| <PrivateRoute path="/auction/new" component={NewAuction} />
```

This link can be added to any of the view components that may be accessed by the seller, for example, in a view where a seller manages their auctions in the marketplace. Now that it is possible to add new auctions in the marketplace, in the next section, we will discuss how to fetch these auctions from the database in the backend so that they can be listed in the views in the frontend.

Listing auctions

In the MERN Marketplace application, we will display three different lists of auctions to the users. All users browsing through the platform will be able to view the currently open auctions, in other words, auctions that are live or are going to start at a future date. The sellers will be able to view a list of auctions that they created, while signed in users will be able to view the list of auctions they placed bids in. The list displaying the open auctions to all the users will render as follows, providing a summary of each auction, along with an option so that the user can view further details in a separate view:

All Auctions

	Antique Elephant Piggy Bank Auction is live 12 bids 1 h 45 m 12 s left Last bid: \$ 32	
	Antique Rhodochrosite Gemstone Ring Auction is live 3 bids 2 h 43 m 12 s left Last bid: \$ 26	
	Vintage Macintosh Set Auction is live 7 bids 2 h 39 m 12 s left Last bid: \$ 314	
	Old Pocket Watch Auction is live 2 bids 2 d 49 m 12 s left Last bid: \$ 54	
	Miniature Old English Frigate Auction is live 0 bids 3 d 4 h 14 m 12 s left	
	Egyptian Jewelry Box Auction Starts at 1/16/2020, 9:30:00 PM	
	Antique Rose Quartz Oval Ring Auction Starts at 1/17/2020, 8:13:00 PM	

In the following sections, in order to implement these different auction lists so that they're displayed in the application, we will define the three separate backend APIs to retrieve open auctions, auctions by a seller, and auctions by a bidder, respectively. Then, we will implement a reusable React component that will take any list of auctions provided to it as a prop and render it to the view. This will allow us to display all three lists of auctions while utilizing the same component.

The open Auctions API

To retrieve the list of open auctions from the database, we will define a backend API that accepts a GET request and queries the Auction collection to return the open auctions that are found in the response. To implement this open auctions API, we will declare a route, as shown here.

mern-marketplace/server/routes/auction.routes.js:

```
| router.route('/api/auctions')
|   .get(auctionCtrl.listOpen)
```

A GET request that's received at the `/api/auctions` route will invoke the `listOpen` controller method, which will query the Auction collection in the database so that it returns all the auctions with ending dates greater than the current date. The `listOpen` method is defined as follows.

mern-marketplace/server/controllers/auction.controller.js:

```
const listOpen = async (req, res) => {
  try {
    let auctions = await Auction.find({ bidEnd: { $gt: new Date() } })
      .sort('bidStart')
      .populate('seller', '_id name')
      .populate('bids.bidder', '_id name')

    res.json(auctions)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The auctions that are returned by the query in this `listOpen` method will be sorted by the starting date, with auctions that start earlier shown first. These auctions will also contain the ID and name details of the seller and each bidder. The resulting array of auctions will be sent back in the response to the requesting client.

To fetch this API in the frontend, we will add a corresponding `listOpen` method in `api-auction.js`, similar to other API implementations. This fetch method will be used in the frontend component that displays the open auctions to the user. Next, we will implement another API to list all the auctions that a specific user placed bids in.

The Auctions by bidder API

To be able to display all the auctions that a given user placed bids in, we will define a backend API that accepts a GET request and queries the Auction collection so that it returns the relevant auctions in the response. To implement this auctions by bidder API, we will declare a route, as shown here.

```
mern-marketplace/server/routes/auction.routes.js
```

```
| router.route('/api/auctions/bid/:userId')
|   .get(auctionCtrl.listByBidder)
```

A GET request, when received at the `/api/auctions/bid/:userId` route, will invoke the `listByBidder` controller method, which will query the Auction collection in the database so that it returns all the auctions that contain bids with a bidder matching the user specified by the `userId` parameter in the route. The `listByBidder` method is defined as follows.

```
mern-marketplace/server/controllers/auction.controller.js:
```

```
const listByBidder = async (req, res) => {
  try {
    let auctions = await Auction.find({'bids.bidder': req.profile._id})
      .populate('seller', '_id name')
      .populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This method will return the resulting auctions in response to the requesting client, and each auction will also contain the ID and name details of the seller and each bidder. To fetch this API in the frontend, we will add a corresponding `listByBidder` method in `api-auction.js`, similar to other API implementations. This fetch method will be used in the frontend component that displays the auctions related to a specific

bidder. Next, we will implement an API that will list all the auctions that a specific seller created in the marketplace.

The Auctions by seller API

Sellers in the marketplace will see a list of auctions that they created. To retrieve these auctions from the database, we will define a backend API that accepts a GET request and queries the Auction collection so that it returns the auctions by a specific seller. To implement this auctions by seller API, we will declare a route, as shown here.

mern-marketplace/server/routes/auction.routes.js:

```
router.route('/api/auctions/by/:userId')
  .get(authCtrl.requireSignin, authCtrl.hasAuthorization,
    auctionCtrl.listBySeller)
```

A GET request, when received at the `/api/auctions/by/:userId` route, will invoke the `listBySeller` controller method, which will query the Auction collection in the database so that it returns all the auctions with sellers matching the user specified by the `userId` parameter in the route. The `listBySeller` method is defined as follows.

mern-marketplace/server/controllers/auction.controller.js:

```
const listBySeller = async (req, res) => {
  try {
    let auctions = await Auction.find({seller: req.profile._id})
      .populate('seller', '_id name')
      .populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This method will return the auctions for the specified seller in response to the requesting client, and each auction will also contain the ID and name details of the seller and each bidder.

To fetch this API in the frontend, we will add a corresponding `listBySeller` method in `api-auction.js`, similar to other API implementations. This fetch method will be used in the frontend component that displays the auctions related to a specific seller. In the next section, we will look at the implementation of the Auctions component, which will take any of these lists of auctions and display it to the end user.

The Auctions component

The different auction lists in the application will be rendered using a React component that takes an array of auction objects as props. We will implement this reusable `Auctions` component and add it to the views that will retrieve and display either the open auctions, auctions by a bidder, or auctions by a seller. The view that retrieves and renders the list of auctions created by a specific seller using the `Auctions` component will look as follows:

The screenshot shows a user interface for managing auctions. At the top left is a title "Your Auctions". To the right is a button labeled "NEW AUCTION" with a plus sign icon. Below this is a list of five auction items, each with a small thumbnail image, the item name, its status, and bidding information. To the right of each item are three icons: a blue eye, a blue pencil, and a green trash can.

Auction Item	Status	Bidding Details	Actions
Rutilated Quartz Ring	Auction Ended	6 bids Last bid: \$ 54	
Antique Rhodochrosite Gemstone Ring	Auction is live	3 bids 2 h 37 m 39 s left Last bid: \$ 26	
Antique Rose Quartz Oval Ring	Auction is live	0 bids 15 h 44 m 39 s left	
Snowflake Obsidian Ring	Auction Starts at	1/18/2020, 8:00:00 PM	
Green-striped Agate Teardrop Ring	Auction Starts at	1/19/2020, 8:00:00 PM	

The `Auctions` component will iterate over the array of auctions received as a prop and display each auction in a Material-UI `List` component, as shown in the following code.

mern-marketplace/client/auction/Auctions.js:

```
export default function Auctions(props) {
  return (
    <List dense>
      {props.auctions.map((auction, i) => {
        return <span key={i}>
```

```

        <ListItem button>
          <ListItemAvatar>
            <Avatar src={'/api/auctions/image/' + auction._id + "?" +
                         + new Date().getTime()}/>
          </ListItemAvatar>
          <ListItemText primary={auction.itemName} secondary={auctionState(auction)}/>
          <ListItemSecondaryAction>
            <Link to={"/auction/" + auction._id}>
              <IconButton aria-label="View" color="primary">
                <ViewIcon/>
              </IconButton>
            </Link>
          </ListItemSecondaryAction>
        </ListItem>
        <Divider/>
      </span>)) }
    </List>
  )
}

```

For each auction item, besides displaying some basic auction details, we give the users an option to open each auction in a separate link. We also conditionally render details such as when an auction will start, whether bidding has started or ended, how much time is left, and what the latest bid is. These details of each auction's state are determined and rendered with the following code.

mern-marketplace/client/auction/Auctions.js:

```

const currentDate = new Date()
const auctionState = (auction)=>{
  return ( <span>
    {currentDate < new Date(auction.bidStart) &&
      `Auction Starts at ${new
Date(auction.bidStart).toLocaleString()} `}
      {currentDate > new Date(auction.bidStart) &&
        currentDate < new Date(auction.bidEnd) && <>
          {`Auction is live | ${auction.bids.length} bids |`}
          {showTimeLeft(new Date(auction.bidEnd)) }
        </>}
      {currentDate > new Date(auction.bidEnd) &&
        `Auction Ended | ${auction.bids.length} bids `}
      {currentDate > new Date(auction.bidStart) && auction.bids.length> 0
&& `

        | Last bid: $ ${auction.bids[0].bid}`}
      </span>
    )
}

```

To calculate and render the time left for auctions that have already started, we define a `showTimeLeft` method, which takes the end date as an argument and uses the `calculateTimeLeft` method to construct the time string rendered in the view. The `showTimeLeft` method is defined as follows.

mern-marketplace/client/auction/Auctions.js:

```
const showTimeLeft = (date) => {
  let timeLeft = calculateTimeLeft(date)
  return !timeLeft.timeEnd && <span>
    {timeLeft.days != 0 && `${timeLeft.days} d `}
    {timeLeft.hours != 0 && `${timeLeft.hours} h `}
    {timeLeft.minutes != 0 && `${timeLeft.minutes} m `}
    {timeLeft.seconds != 0 && `${timeLeft.seconds} s`} left
  </span>
}
```

This method uses the `calculateTimeLeft` method to determine the breakdown of the time left in days, hours, minutes, and seconds.

The `calculateTimeLeft` method takes the end date and compares it with the current date to calculate the difference and makes a `timeLeft` object that records the remaining days, hours, minutes, and seconds, as well as a `timeEnd` state. If the time has ended, the `timeEnd` state is set to true. The `calculateTimeLeft` method is defined as follows.

mern-marketplace/client/auction/Auctions.js:

```
const calculateTimeLeft = (date) => {
  const difference = date - new Date()
  let timeLeft = {}

  if (difference > 0) {
    timeLeft = {
      days: Math.floor(difference / (1000 * 60 * 60 * 24)),
      hours: Math.floor((difference / (1000 * 60 * 60)) % 24),
      minutes: Math.floor((difference / 1000 / 60) % 60),
      seconds: Math.floor((difference / 1000) % 60),
      timeEnd: false
    }
  } else {
    timeLeft = {timeEnd: true}
  }
  return timeLeft
}
```

This `Auctions` component, which renders a list of auctions with the details and a status of each, can be added to other views that will display different auction lists. If the user who's currently viewing an auction list happens to be a seller for a given auction in the list, we also want to render the option to edit or delete the auction to this user. In the next section, we will learn how to incorporate these options to edit or delete an auction from the marketplace.

Editing and deleting auctions

A seller in the marketplace will be able to manage their auctions by either editing or deleting an auction that they've created. The implementations of the edit and delete features will require building backend APIs that save changes to the database and remove an auction from the collection. These APIs will be used in frontend views to allow users to edit auction details using a form and initiate delete with a button click. In the following sections, we will learn how to add these options conditionally to the auction list and discuss the full-stack implementation to complete these edit and delete functions.

Updating the list view

We will update the code for the auctions list view to conditionally show the edit and delete options to the seller. In the `Auctions` component, which is where a list of auctions is iterated over to render each item in `ListItem`, we will add two more options in the `ListItemSecondaryAction` component, as shown in the following code.

mern-marketplace/client/auction/Auctions.js:

```
<ListItemSecondaryAction>
  <Link to={"/auction/" + auction._id}>
    <IconButton aria-label="View" color="primary">
      <ViewIcon/>
    </IconButton>
  </Link>
  { auth.isAuthenticated().user &&
    auth.isAuthenticated().user._id == auction.seller._id &&
    (<>
      <Link to={"/auction/edit/" + auction._id}>
        <IconButton aria-label="Edit" color="primary">
          <Edit/>
        </IconButton>
      </Link>
      <DeleteAuction auction={auction} onRemove={props.removeAuction}/>
    </>)
  }
</ListItemSecondaryAction>
```

The link to the edit view and the delete component are rendered conditionally if the currently signed in user's ID matches the ID of the auction seller. The implementation for the Edit view component and Delete component is similar to the `EditShop` component and `DeleteShop` component we discussed in [Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*. These same components will call backend APIs to complete the edit and delete actions. We will look at the required backend APIs in the next section.

Edit and delete auction APIs

To complete the edit auction and delete auction operations initiated by sellers from the frontend, we need to have the corresponding APIs in the backend. The route for these API endpoints, which will accept the update and delete requests, can be declared as follows.

mern-marketplace/server/routes/auction.routes.js:

```
router.route('/api/auctions/:auctionId')
    .put(authCtrl.requireSignin, auctionCtrl.isSeller, auctionCtrl.update)
    .delete(authCtrl.requireSignin, auctionCtrl.isSeller,
    auctionCtrl.remove)
router.param('auctionId', auctionCtrl.auctionByID)
```

The `:auctionId` param in the `/api/auctions/:auctionId` route URL will invoke the `auctionByID` controller method, which is similar to the `userByID` controller method. It retrieves the auction from the database and attaches it to the request object so that it can be used in the `next` method. The `auctionByID` method is defined as follows.

mern-marketplace/server/controllers/auction.controller.js:

```
const auctionByID = async (req, res, next, id) => {
    try {
        let auction = await Auction.findById(id)
            .populate('seller', '_id name')
            .populate('bids.bidder', '_id
name').exec()
        if (!auction)
            return res.status('400').json({
                error: "Auction not found"
            })
        req.auction = auction
        next()
    } catch (err) {
        return res.status('400').json({
            error: "Could not retrieve auction"
        })
    }
}
```

The auction object that's retrieved from the database will also contain the name and ID details of the seller and bidders, as we specified in the `populate()` methods. For these API endpoints, the `auction` object is used next to verify that the currently signed-in user is the seller who created this given auction by invoking the `isSeller` method, which is defined in the auction controller as follows.

`mern-marketplace/server/controllers/auction.controller.js`:

```
const isSeller = (req, res, next) => {
  const isSeller = req.auction && req.auth && req.auction.seller._id ==
  req.auth._id
  if(!isSeller){
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}
```

Once the seller has been verified, the `next` method is invoked to either update or delete the auction, depending on whether a PUT or DELETE request was received. The controller methods for updating and deleting auctions are similar to the previous implementations for update and delete, as we discussed for the edit shop API and delete shop API in [Chapter 7, Exercising MERN Skills with an Online Marketplace](#).

We have the auction module for the marketplace ready with an Auction model for storing auction and bidding data and backend APIs and frontend views for creating new auctions, displaying different auction lists, and modifying an existing auction. In the next section, we will extend this module further and implement a view for individual auctions where, besides learning more about the auction, users will also be able to see live bidding updates.

Displaying the auction view

The view for displaying a single auction will contain the core functionality of the real-time auction and bidding features for the marketplace. Before getting into the implementation of real-time bidding, we will set up the full-stack slice for retrieving details of a single auction and display these details in a React component that will house the auction display, timer, and bidding capabilities. In the following sections, we will start by discussing the backend API for fetching a single auction. Then, we will look at the implementation of an Auction component, which will use this API to retrieve and display the auction details, along with the state of the auction. To give users a real-time update of the state of the auction, we will also implement a timer in this view to indicate the time left until a live auction ends.

The read auction API

To display the details of an existing auction in a view of its own, we need to add a backend API that will receive a request for the auction from the client and return its details in the response. Therefore, we will implement a read auction API in the backend that will accept a GET request with a specified auction ID and return the corresponding auction document from the `Auction` collection in the database. We will start adding this API endpoint by declaring a GET route, as shown in the following code.

mern-marketplace/server/routes/auction.routes.js:

```
| router.route('/api/auction/:auctionId')
|   .get(auctionCtrl.read)
```

The `:auctionId` param in the route URL invokes the `auctionByID` controller method when a GET request is received at this route.

The `auctionByID` controller method retrieves the auction from the database and attaches it to the request object to be accessed in the `read` controller method, which is called next. The `read` controller method, which returns this auction object in response to the client, is defined as follows.

mern-marketplace/server/controllers/auction.controller.js:

```
| const read = (req, res) => {
|   req.auction.image = undefined
|   return res.json(req.auction)
| }
```

We are removing the `image` field before sending the response, since images will be retrieved as files in separate routes. With this API ready in the backend, we can now add the implementation to call it in the frontend by adding a `fetch` method in `api-auction.js`, similar to the other `fetch` methods we've discussed for completing API implementations. We will use the `fetch` method to call the `read`

auction API in a React component that will render the retrieved auction details. The implementation of this React component is discussed in the next section.

The Auction component

We will implement an Auction component to fetch and display the details of a single auction to the end user. This view will also have real-time update functionalities that will render based on the current state of the auction and on whether the user viewing the page is signed in. For example, the following screenshot shows how the Auction component renders to a visitor when a given auction has not started yet. It only displays the description details of the auction and specifies when the auction will start:

Antique Elephant Piggy Bank

Auction Not Started



Auction Starts at 1/16/2020, 4:00:00 PM

[About Item](#)

Rare antique elephant piggy bank with a medieval flare! Add this one-of-a-kind piece to your collection.

The implementation of the `Auction` component will retrieve the auction details by calling the read auction API in a `useEffect` hook. This part of the component implementation is similar to the `Shop` component we

discussed in [Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*.

The completed `Auction` component will be accessed in the browser at the `/auction/:auctionId` route, which is defined in `MainRouter` as follows.

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

```
| <Route path="/auction/:auctionId" component={Auction}/>
```

This route can be used in any component to link to a specific auction, as we did in the auction lists. This link will take the user to the corresponding `Auction` view with the auction details loaded.

In the component view, we will render the auction state by considering the current date and the given auction's bidding start and end timings. The code to generate these states, which will be shown in the view, can be added as follows.

`mern-marketplace/client/auction/Auction.js`:

```
const currentDate = new Date()
...
<span>
  {currentDate < new Date(auction.bidStart) && 'Auction Not Started'}
  {currentDate > new Date(auction.bidStart) && currentDate < new
Date(auction.bidEnd) && 'Auction Live'}
  {currentDate > new Date(auction.bidEnd) && 'Auction Ended'}
</span>
```

In the preceding code, if the current date is before the `bidStart` date, we show a message indicating that the auction has not started yet. If the current date is between the `bidStart` and `bidEnd` dates, then the auction is live. If the current date is after the `bidEnd` date, then the auction has ended.

The `Auction` component will also conditionally render a timer and a bidding section, depending on whether the current user is signed in, and also on the state of the auction at the moment. The code to render this part of the view will be as follows.

mern-marketplace/client/auction/Auction.js:

```
<Grid item xs={7} sm={7}>
  {currentDate > new Date(auction.bidStart)
  ? (<>
      <Timer endTime={auction.bidEnd} update={update}/>
      { auction.bids.length > 0 &&
        <Typography component="p" variant="subtitle1">
          {`Last bid: ${auction.bids[0].bid}`}
        </Typography>
      }
    { !auth.isAuthenticated() &&
      <Typography>
        Please, <Link to='/signin'>
          sign in</Link> to place your bid.
      </Typography>
    }
    { auth.isAuthenticated() &&
      <Bidding auction={auction} justEnded={justEnded} updateBids={updateBids}/>
    }
  </>)
  : <Typography component="p" variant="h6">
    {`Auction Starts at ${new Date(auction.bidStart).toLocaleString()}`}
  </Typography>
}
</Grid>
```

If the current date happens to be after the bid starting time, instead of showing the start time, we render the `Timer` component to show the time remaining until bidding ends. Then, we show the last bid amount, which will be the first item in the `auction.bids` array if some bids were already placed. If the current user is signed in when the auction is in this state, we also render a `Bidding` component, which will allow them to bid and see the bidding history. In the next section, we will learn how to implement the `Timer` component we added in this view to show the remaining time for the auction.

Adding the Timer component

When the auction is live, we will give the users a real-time update of how long they have before bidding ends on this given auction. We will implement a `Timer` component and conditionally render it in the `Auction` component to achieve this feature. The timer will count down the seconds and show how much time is left to the users viewing the live auction. The following screenshot shows what the `Auction` component looks like when it renders a live auction to a user who is not signed in yet:



The remaining time decreases per second as the user is viewing the live auction. We will implement this countdown feature in the `Timer` component, which is added to the `Auction` component. The `Auction` component provides it with props containing the auction end time value, as well as a function to update the auction view when the time ends, as shown in the following code.

`mern-marketplace/client/auction/Auction.js`:

```
| <Timer endTime={auction.bidEnd} update={update}>/>
```

The `update` function that's provided to the `Timer` component will help set the value of the `justEnded` variable from `false` to `true`. This `justEnded` value is passed to the `Bidding` component so that it can be used to disable the option to place bids when the time ends. The `justEnded` value is initialized and the `update` function is defined as follows.

mern-marketplace/client/auction/Auction.js:

```
const [justEnded, setJustEnded] = useState(false)
const updateBids = () => {
    setJustEnded(true)
}
```

These props will be used in the `Timer` component to calculate time left and to update the view when time is up.

In the `Timer` component definition, we will initialize the `timeLeft` variable in the state, using the end time value sent in the props from the `Auction` component, as shown in the following code.

mern-marketplace/client/auction/Timer.js:

```
export default function Timer (props) {
    const [timeLeft, setTimeLeft] = useState(calculateTimeLeft(new
Date(props.endTime)))
    ...
}
```

To calculate the time left until the auction ends, we utilize the `calculateTimeLeft` method we discussed previously in the *The Auctions component* section of this chapter.

To implement the time countdown functionality, we will use `setTimeout` in a `useEffect` hook in the `Timer` component, as shown in the following code.

mern-marketplace/client/auction/Timer.js:

```
useEffect(() => {
    let timer = null
    if(!timeLeft.timeEnd) {
        timer = setTimeout(() => {
```

```

        setTimeLeft(calculateTimeLeft(new
Date(props.endTime)))
    }, 1000)
} else{
    props.update()
}
return () => {
    clearTimeout(timer)
}
})
})

```

If the time has not ended already, we will use `setTimeout` to update the `timeLeft` value after 1 second has passed. This `useEffect` hook will run after every render caused by the state update with `setTimeLeft`.

As a result, the `timeLeft` value will keep updating every second until the `timeEnd` value is `true`. When the `timeEnd` value does become `true` as the time is up, we will execute the `update` function that's sent in the props from the `Auctions` component.

To avoid a memory leak and to clean up in the `useEffect` hook, we will use `clearTimeout` to stop any pending `setTimeout` calls. To show this updating `timeLeft` value, we just need to render it in the view, as shown in the following code.

mern-marketplace/client/auction/Timer.js:

```

return (<div className={props.style}>
    {!timeLeft.timeEnd ?
        <Typography component="p" variant="h6" >
            {timeLeft.days != 0 && `${timeLeft.days} d `}
            {timeLeft.hours != 0 && `${timeLeft.hours} h `}
            {timeLeft.minutes != 0 && `${timeLeft.minutes} m `}
            {timeLeft.seconds != 0 && `${timeLeft.seconds} s`} left
        <span style={{fontSize:'0.8em'}}>
            {` (ends at ${new Date(props.endTime).toLocaleString()})`}
        </span>
    </Typography> :
        <Typography component="p" variant="h6">Auction
ended</Typography>
    }
    </div>
)

```

If there is time left, we render the days, hours, minutes, and seconds remaining until the auction ends using the `timeLeft` object. We also

indicate the exact date and time when the auction ends. If the time is up, we just indicate that the auction ended.

In the `Auction` component we've implemented so far, we are able to fetch the auction details from the backend and render it along with the state of the auction. If an auction is in a live state, we are able to indicate the time left until it ends. When an auction is in this live state, users will also be able to place bids against the auction and see the bids being placed by other users on the platform from within this view in real-time. In the next section, we will discuss how to use Socket.IO to integrate this real-time bidding feature for all live auctions on the platform.

Implementing real-time bidding with Socket.IO

Users who are signed in to the marketplace platform will be able to take part in live auctions. They will be able to place their bids and get real-time updates in the same view while other users on the platform are counteracting their bids. To implement this functionality, we will integrate Socket.IO with our full-stack MERN application before implementing the frontend interface to allow users to place their bids and see the changing bidding history.

Integrating Socket.IO

Socket.IO will allow us to add the real-time bidding feature to auctions in the marketplace application. Socket.IO is a JavaScript library with a client-side module that runs in the browser and a server-side module that integrates with Node.js. Integrating these modules with our MERN-based application will enable bidirectional and real-time communication between the clients and the server.



The client-side part of Socket.IO is available as the Node module `socket.io-client`, while the server-side part is available as the Node module `socket.io`. You can learn more about Socket.IO and try their getting started tutorials at <https://socket.io>.

Before we can start using `socket.io` in our code, we will install the client and server libraries with Yarn by running the following command from the command line:

```
| yarn add socket.io socket.io-client
```

With the Socket.IO libraries added to the project, we will update our backend to integrate Socket.IO with the server code. We need to initialize a new instance of `socket.io` using the same HTTP server that we are using for our application.

In our backend code, we are using Express to start the server. Therefore, we will update the code in `server.js` to get a reference to the HTTP server that our Express app is using to listen for requests from clients, as shown in the following code.

mern-marketplace/server/server.js:

```
import bidding from './controllers/bidding.controller'

const server = app.listen(config.port, (err) => {
  if (err) {
    console.log(err)
  }
  console.info('Server started on port %s.', config.port)
})
```

```
bidding(server)
```

Then, we will pass the reference for this server to a bidding controller function. This `bidding.controller` function will contain the Socket.IO code that's needed on the server-side to implement real-time features.

The `bidding.controller` function will initialize `socket.io` and then listen on the `connection` event for incoming socket messages from clients, as shown in the following code.

mern-marketplace/server/controllers/bidding.controller.js:

```
export default (server) => {
  const io = require('socket.io').listen(server)
  io.on('connection', function(socket){
    socket.on('join auction room', data => {
      socket.join(data.room);
    })
    socket.on('leave auction room', data => {
      socket.leave(data.room)
    })
  })
}
```

When a new client first connects and then disconnects to the socket connection, we will subscribe and unsubscribe the client socket to a given channel. The channel will be identified by the auction ID that will be passed in the `data.room` property from the client. This way, we will have a different channel or room for each auction.

With this code, the backend is ready to receive communication from clients over sockets, and we can now add the Socket.IO integration to our frontend. In the frontend, only the auction view – specifically, the bidding section – will be using sockets for real-time communication. Therefore, we will only integrate Socket.IO in the `Bidding` component that we add to the Auction component in the frontend, as shown in the following code.

mern-marketplace/client/auction/Auction.js:

```
<Bidding auction={auction} justEnded={justEnded} updateBids={updateBids}/>
```

The Bidding component takes the `auction` object, the `justEnded` value, and an `updateBids` function as props from the Auction component, and uses these in the bidding process. To start implementing the Bidding component, we will integrate sockets using the Socket.IO client-side library, as shown in the following code.

mern-marketplace/client/auction/Bidding.js:

```
const io = require('socket.io-client')
const socket = io()

export default function Bidding (props) {
  useEffect(() => {
    socket.emit('join auction room', {room: props.auction._id})
    return () => {
      socket.emit('leave auction room', {
        room: props.auction._id
      })
    }
  }, [])
  ...
}
```

In the preceding code, we require the `socket.io-client` library and initialize the `socket` for this client. Then, in our `Bidding` component definition, we utilize the `useEffect` hook and the initialized `socket` to emit the *auction room joining* and *auction room leaving* socket events when the component mounts and unmounts, respectively. We pass the current auction's ID as the `data.room` value with these emitted socket events.

These events will be received by the server socket connection, resulting in subscription or unsubscription of the client to the given auction room. Now that the clients and the server are able to communicate in real-time over sockets, in the next section, we will learn how to use this capability to let users place instant bids on the auction.

Placing bids

When a user on the platform is signed in and viewing an auction that is currently live, they will see an option to place their own bid. This option will be rendered within the `Bidding` component, as shown in the following screenshot:

The screenshot shows a web-based auction interface. At the top, the title "Antique Elephant Piggy Bank" is displayed, followed by the status "Auction Live". Below the title is a large image of a bronze-colored elephant piggy bank. To the right of the image, there is a timer indicating "6 h 56 m 39 s left" and the auction ends at "1/16/2020, 10:01:00 PM". Below the timer, it says "Last bid: \$ 12". A bidding form is present with a placeholder "Your Bid (\$)" and a note "Enter \$13 or more". A "PLACE BID" button is located within this form. At the bottom of the interface, there are two buttons: "About Item" on the left and "All bids" on the right.

To allow users to place their bids, in the following sections, we will add a form that lets them enter a value more than the last bid and submit it to the server using socket communication. Then, on the server, we will handle this new bid that's been sent over the socket so that the changed auction bids can be saved in the database and the view can be updated instantly for all connected users when the server accepts this bid.

Adding a form to enter a bid

We will add the form to place a bid for an auction in the `Bidding` component that we started building in the previous section. Before we add the form elements in the view, we will initialize the `bid` value in the state, add a change handling function for the form input, and keep track of the minimum bid amount allowed, as shown in the following code.

mern-marketplace/client/auction/Bidding.js:

```
const [bid, setBid] = useState('')
const handleChange = event => {
    setBid(event.target.value)
}
const minBid = props.auction.bids && props.auction.bids.length > 0
    ? props.auction.bids[0].bid
    : props.auction.startingBid
```

The minimum bid amount is determined by checking the latest bid placed. If any bids were placed, the minimum bid needs to be higher than the latest bid; otherwise, it needs to be higher than the starting bid that was set by the auction seller.

The form elements for placing a bid will only render if the current date is before the auction end date. We also check if the `justEnded` value is `false` so that the form can be hidden when the time ends in real-time as the timer counts down to 0. The form elements will contain an input field, a hint at what minimum amount should be entered, and a submit button, which will remain disabled unless a valid bid amount is entered. These elements will be added to the `Bidding` component view as follows.

mern-marketplace/client/auction/Bidding.js:

```
{!props.justEnded && new Date() < new Date(props.auction.bidEnd) && <>
<TextField label="Your Bid ($)"
```

```
        value={bid} onChange={handleChange}
        type="number" margin="normal"
        helperText={`Enter ${Number(minBid)+1} or more`}/><br/>
    <Button variant="contained" color="secondary"
        disabled={bid < (minBid + 1)}
        onClick={placeBid}>Place Bid
    </Button><br/>
</>}
```

When the user clicks on the submit button, the `placeBid` function will be called. In this function, we construct a bid object containing the new bid's details, including the bid amount, bid time, and the bidder's user reference. This new bid is emitted to the server over the socket communication that's already been established for this auction room, as shown in the following code:

```
const placeBid = () => {
  const jwt = auth.isAuthenticated()
  let newBid = {
    bid: bid,
    time: new Date(),
    bidder: jwt.user
  }
  socket.emit('new bid', {
    room: props.auction._id,
    bidInfo: newBid
  })
  setBid('')
}
```

Once the message has been emitted over the socket, we will empty the input field with `setBid('')`. Then, we need to update the bidding controller in the backend to receive and handle this new bid message that's been sent from the client. In the next section, we will add the socket event handling code to complete this process to place a bid.

Receiving a bid on the server

When a new bid is placed by a user and emitted over a socket connection, it will be handled on the server so that it's stored in the corresponding auction in the database.

In the bidding controller, we will update the socket event handlers in the socket connection listener code in order to add a handler for the *new bid* socket message, as shown in the following code.

mern-marketplace/server/controllers/bidding.controller.js:

```
io.on('connection', function(socket) {
  ...
  socket.on('new bid', data => {
    bid(data.bidInfo, data.room)
  })
})
```

In the preceding code, when the socket receives the emitted *new bid* message, we use the attached data to update the specified auction with the new bid information in a function called `bid`. The `bid` function is defined as follows.

mern-marketplace/server/controllers/bidding.controller.js:

```
const bid = async (bid, auction) => {
  try {
    let result = await Auction.findOneAndUpdate({ _id: auction, $or:
      [ { 'bids.0.bid': { $lt: bid.bid } }, { bids: { $eq: [] } } ] },
      { $push: { bids: { $each: [ bid ], $position: 0 } } },
      { new: true })
      .populate('bids.bidder', '_id name')
      .populate('seller', '_id name')
      .exec()
  }
  io.to(auction).emit('new bid', result)
}
catch(err) {
  console.log(err)
}
```

The bid function takes the new bid details and the auction ID as arguments and performs a `findOneAndUpdate` operation on the Auction collection. To find the auction to be updated, besides querying with the auction ID, we also ensure that the new bid amount is larger than the last bid placed at position `0` of the `bids` array in this auction document. If an auction is found that matches the provided ID and also meets this condition of the last bid being smaller than the new bid, then this auction is updated by pushing the new bid into the first position of the `bids` array.

After the update to the auction in the database, we emit the *new bid* message over the `socket.io` connection to all the clients currently connected to the corresponding auction room. On the client-side, we need to capture this message in a socket event handler code and update the view with the latest bids. In the next section, we will learn how to handle and display this updated list of bids for all the clients viewing the live auction.

Displaying the changing bidding history

After a new bid is accepted on the server and stored in the database, the new array of bids will be updated in the view for all the clients currently on the auctions page. In the following sections, we will extend the `Bidding` component so that it handles the updated bids and displays the complete bidding history for the given auction.

Updating the view state with a new bid

Once the placed bid has been handled on the server, the updated auction containing the modified array of bids is sent to all the clients connected to the auction room. To handle this new data on the client-side, we need to update the `Bidding` component to add a listener for this specific socket message.

We will use an `useEffect` hook to add this socket listener to the `Bidding` component when it loads and renders. We will also remove the listener with `socket.off()` in the `useEffect` cleanup when the component unloads. This `useEffect` hook with the socket listener for receiving the new bid data will be added as follows.

mern-marketplace/client/auction/Bidding.js:

```
useEffect(() => {
  socket.on('new bid', payload => {
    props.updateBids(payload)
  })
  return () => {
    socket.off('new bid')
  }
})
```

When the new auction with updated bids is received from the server in the socket event, we execute the `updateBids` function that was sent as a prop from the `Auction` component. The `updateBids` function is defined in the `Auction` component as follows:

```
const updateBids = (updatedAuction) => {
  setAuction(updatedAuction)
}
```

This will update the auction data that was set in the state of the `Auction` component and, as a result, rerender the complete auction

view with the updated auction data. This view will also include the bidding history table, which we'll discuss in the next section.

Rendering the bidding history

In the `Bidding` component, we will render a table that displays the details of all the bids that were placed for the given auction. This will inform the user of the bids that were already placed and are being placed in real-time as they are viewing a live auction. The bidding history for an auction will render in the view as follows:

All bids		
Bid Amount	Bid Time	Bidder
\$29	1/16/2020, 2:29:35 PM	Sarah Atkin
\$28	1/16/2020, 2:21:15 PM	Mary
\$26	1/16/2020, 2:20:11 PM	Sarah Atkin
\$24	1/16/2020, 2:19:46 PM	Jane
\$23	1/16/2020, 2:18:37 PM	Sarah Atkin

This bidding history view will basically iterate over the `bids` array for the auction and display the bid amount, bid time, and bidder name for each bid object that's found in the array. The code for rendering this table view will be added as follows:

```
<div>
  <Typography variant="h6"> All bids </Typography>
  <Grid container spacing={4}>
    <Grid item xs={3} sm={3}>
      <Typography variant="subtitle1"
        color="primary">Bid Amount</Typography>
    </Grid>
    <Grid item xs={5} sm={5}>
      <Typography variant="subtitle1"
        color="primary">Bid Time</Typography>
    </Grid>
    <Grid item xs={4} sm={4}>
      <Typography variant="subtitle1"
        color="primary">Bidder</Typography>
    </Grid>
  </Grid>
  {props.auction.bids.map((item, index) => {
    return <Grid container spacing={4} key={index}>
```

```

        <Grid item xs={3} sm={3}>
          <Typography variant="body2">${item.bid}</Typography>
        </Grid>
        <Grid item xs={5} sm={5}>
          <Typography variant="body2">
            {new Date(item.time).toLocaleString()}
          </Typography></Grid>
        <Grid item xs={4} sm={4}>
          <Typography variant="body2">{item.bidder.name}</Typography>
        </Grid>
      </Grid>
    </div>
  )})
</div>

```

We added table headers using Material-UI `Grid` components, before iterating over the `bids` array to generate the table rows with individual bid details.

When a new bid is placed by any user viewing this auction and the updated auction is received in the socket and set to state, this table containing the bidding history will update for all its viewers and show the latest bid at the top of the table. By doing this, it gives all the users in the auction room a real-time update of bidding. With that, we have a complete auction and real-time bidding feature integrated with the MERN Marketplace application.

Summary

In this chapter, we extended the MERN Marketplace application and added an auctioning feature with real-time bidding capabilities. We designed an auction model for storing auction and bidding details and implemented the full-stack CRUD functionalities that allow users to create new auctions, edit and delete auctions, and see different lists of auctions, along with individual auctions.

We added an auction view representing a single auction where users can watch and participate in the auction. In the view, we calculated and rendered the current state of the given auction, along with a countdown timer for live auctions. While implementing this timer that counts down seconds, we learned how to use `setTimeout` in a React component with the `useEffect` hook.

For each auction, we implemented real-time bidding capabilities using Socket.IO. We discussed how to integrate Socket.IO on both the client-side and the server-side of the application to establish real-time and bidirectional communication between clients and servers. With these approaches for extending the MERN stack to incorporate real-time communication functionalities, you can implement even more exciting real-time features using sockets in your own full-stack applications.

Using the experiences you've gained here building out the different features for the MERN Marketplace application, you can also grow the auctioning feature that was covered in this chapter and integrate it with the existing order management and payment processing functionalities in this application.

In the next chapter, we will expand our options with the MERN stack technologies by building an expense tracking application with data visualization features by extending the MERN skeleton.

Advancing to Complex MERN Applications

In this part, we explore how to implement MERN applications with advanced and complex features, including data visualization, media streaming, and VR capabilities.

This section comprises the following chapters:

- [Chapter 10](#), *Integrating Data Visualization with an Expense Tracking Application*
- [Chapter 11](#), *Building a Media Streaming Application*
- [Chapter 12](#), *Customizing the Media Player and Improving SEO*
- [Chapter 13](#), *Developing a Web-Based VR Game*
- [Chapter 14](#), *Making the VR Game Dynamic using MERN*

Integrating Data Visualization with an Expense Tracking Application

These days, it is easy to collect and add data to applications on the internet. As more and more data becomes available, it becomes necessary to process the data and present insights extracted from this data in meaningful and appealing visualizations to end users. In this chapter, we will learn how to use MERN stack technologies along with Victory—a charting library for React—to easily integrate data visualization features in a full-stack application. We will extend the MERN skeleton application to build an expense tracking application, which will incorporate data processing and visualization features for expense data recorded by a user over time.

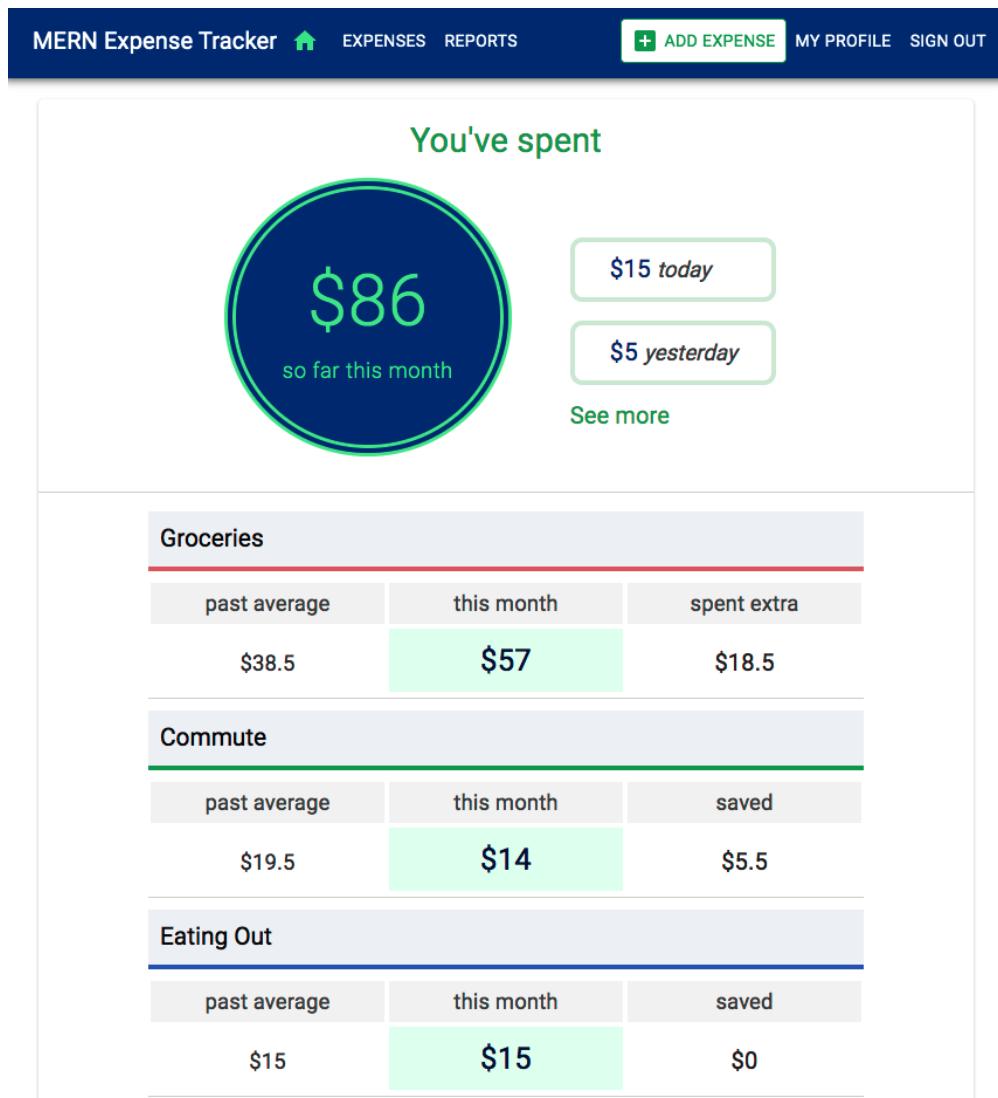
After going through the implementation of these features, you should have a grasp of how to utilize the MongoDB aggregation framework and the Victory charting library to add data visualization features of your choice to any full-stack MERN web application.

In this chapter, we will build an expense tracking application integrated with data visualization features by covering the following topics:

- Introducing MERN Expense Tracker
- Adding expense records
- Visualizing expense data over time

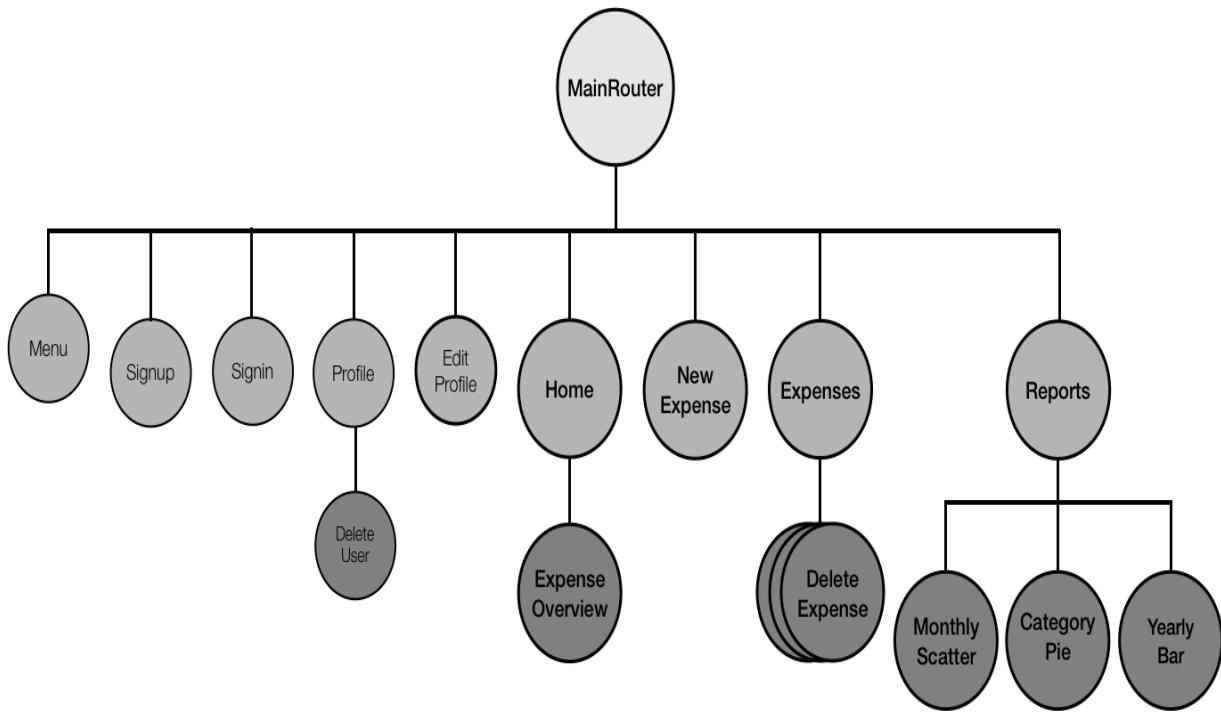
Introducing MERN Expense Tracker

The MERN Expense Tracker application will allow users to keep track of their day-to-day expenses. Users who are signed in to their accounts will be able to add their expense records with details such as expense description, category, amount, and when the given expense was incurred or paid. The application will store these expense records and extract meaningful data patterns to give the user a visual representation of how their expense habits fare as time progresses. The following screenshot shows the home page view for a signed-in user on the MERN Expense Tracker application, and it gives the user an overview of their expenses for the current month:



The code for the complete MERN Expense Tracker application is available on GitHub at: <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter10/mern-expense-tracker>. You can clone this code and run the application as you go through the code explanations for the rest of this chapter.

In this chapter, we will extend the MERN skeleton to build the expense tracking application with data visualization features. The views required for these expense tracking and visualization features will be developed by extending and modifying the existing React components in the MERN skeleton application. The component tree in the following screenshot shows all the custom React components that make up the MERN Expense Tracker frontend developed in this chapter:



We will add new React components to implement views for creating expense records, listing and modifying already recorded expenses, and displaying reports giving insights into expenses incurred by a user over time. We will also modify existing components such as the Home component to render an overview of current expenses by a user. Before we can implement visualizations for the user's expense data, we need to start by adding the capability to record day-to-day expenses on the application. In the next section, we will discuss how to implement this feature allowing signed-in users to create and modify their expense records on the application.

Adding expense records

In the MERN Expense Tracker application, a user who is signed in will be able to create and manage their expense records. To enable these features of adding and managing expense records, we will need to define how to store expense details, and implement the full-stack slices that will let users create new expenses, view these expenses, and update or delete existing expenses on the application.

In the following sections, first, we will define the Expense model with a Mongoose Schema to store the details of each expense record. Then, we will discuss implementations for the backend APIs and frontend views that are needed to allow a user to create new expenses, view a list of their expenses, and modify existing expenses by either editing details of or deleting an expense from the application.

Defining an Expense model

We will implement a Mongoose model to define an Expense model for storing the details of each expense record. This model will be defined in `server/models/expense.model.js`, and the implementation will be similar to other Mongoose Model implementations covered in previous chapters, such as the Course model defined in [Chapter 6, Building a Web-Based Classroom Application](#). The Expense schema in this model will have simple fields to store details about each expense, such as a title, the amount, category, and date when it was incurred, along with a reference to the user who created the record. The code defining the expense fields are given in the following list with explanations:

- **Expense title:** The `title` field will describe the expense. It is declared to be a `String` type and will be a required field:

```
title: {  
  type: String,  
  trim: true,  
  required: 'Title is required'  
},
```

- **Expense amount:** The `amount` field will store the monetary cost of the expense as a value of the `Number` type, and it will be a required field with a minimum allowed value of 0:

```
amount: {  
  type: Number,  
  min: 0,  
  required: 'Amount is required'  
},
```

- **Expense category:** The `category` field will define the expense type, so expenses can be grouped by this value. It is declared to be a `String` type and will be a required field:

```
category: {  
  type: String,
```

```
    trim: true,  
    required: 'Category is required'  
},
```

- **Incurred on:** The `incurred_on` field will store the date-time when the expense was incurred or paid. It is declared to be a `Date` type and will default to the current date-time if no value is provided:

```
incurred_on: {  
    type: Date,  
    default: Date.now  
},
```

- **Notes:** The `notes` field, defined as a `String` type, will allow the recording of additional details or notes for a given expense record:

```
notes: {  
    type: String,  
    trim: true  
},
```

- **Expense recorded by:** The `recorded_by` field will reference the user who is creating the expense record:

```
recorded_by: {  
    type: mongoose.Schema.ObjectId,  
    ref: 'User'  
}
```

- **Created and updated at times:** The `created` and `updated` fields will be `Date` types, with `created` generated when a new expense is added, and `updated` changed when any expense details are modified:

```
updated: Date,  
created: {  
    type: Date,  
    default: Date.now  
},
```

The fields added to this schema definition will enable us to implement all the expense-related features in MERN Expense Tracker. In the next section, we will start developing these features

by implementing the full-stack slice that will allow users to create new expense records.

Creating a new expense record

In order to create a new expense record on the application, we will need to integrate a full-stack slice that allows the user to fill out a form view in the frontend, and then save the entered details to a new expense document in the database in the backend. To implement this feature, in the following sections, we will add a create expense API in the backend, along with a way to fetch this API in the frontend, and a create new expense form view that takes user input for expense details.

The create expense API

For the implementation of the create expense API that will allow creating new expenses in the database, we will first add a `POST` route, as shown in the following code.

`mern-expense-tracker/server/routes/expense.routes.js`:

```
| router.route('/api/expenses')
|   .post(authCtrl.requireSignin, expenseCtrl.create)
```

A `POST` request to this route at `/api/expenses` will first ensure that the requesting user is signed in with the `requireSignin` method from the controllers, before invoking the `create` method to add a new expense record in the database. This `create` method is defined in the following code.

`mern-expense-tracker/server/controllers/expense.controller.js`:

```
| const create = async (req, res) = {
|   try {
|     req.body.recorded_by = req.auth._id
|     const expense = new Expense(req.body)
|     await expense.save()
|     return res.status(200).json({
|       message: "Expense recorded!"
|     })
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

In this `create` method, we set the `recorded_by` field to the user currently signed in, before using the expense data provided in the request body to save the new expense in the `Expense` collection in the database.

The `expense.routes.js` file containing the expense routes will be very similar to the `user.routes` file. To load these new expense routes in the Express app, we need to mount the expense routes in `express.js`, as shown in the following code, in the same way that we did for the auth and user routes.

```
mern-expense-tracker/server/express.js:
```

```
| app.use('/', expenseRoutes)
```

This create expense API endpoint is now ready in the backend and can be used in the frontend to make a `POST` request. To fetch this API in the frontend, we will add a corresponding `create` method in `api-expense.js`, similar to the other API implementations that we discussed in previous chapters, such as the *Creating a new auction* section from [Chapter 9, Adding Real-Time Bidding Capabilities to the Marketplace](#).

This fetch method will be used in the frontend component that will display a form where the user can enter details of the new expense and save it on the application. In the next section, we will implement the React component that will render the form for recording a new expense.

The NewExpense component

Signed-in users on this expense tracking application will interact with a form view in order to enter details of a new expense record. This form view will be rendered in the `NewExpense` component, which will allow users to create a new expense by entering the expense title, the amount spent, the category of the expense, the date-time of when the expense was incurred, and any additional notes.

This form will render as follows:

Expense Record

Title

Amount (\$)

Category

Incurred on

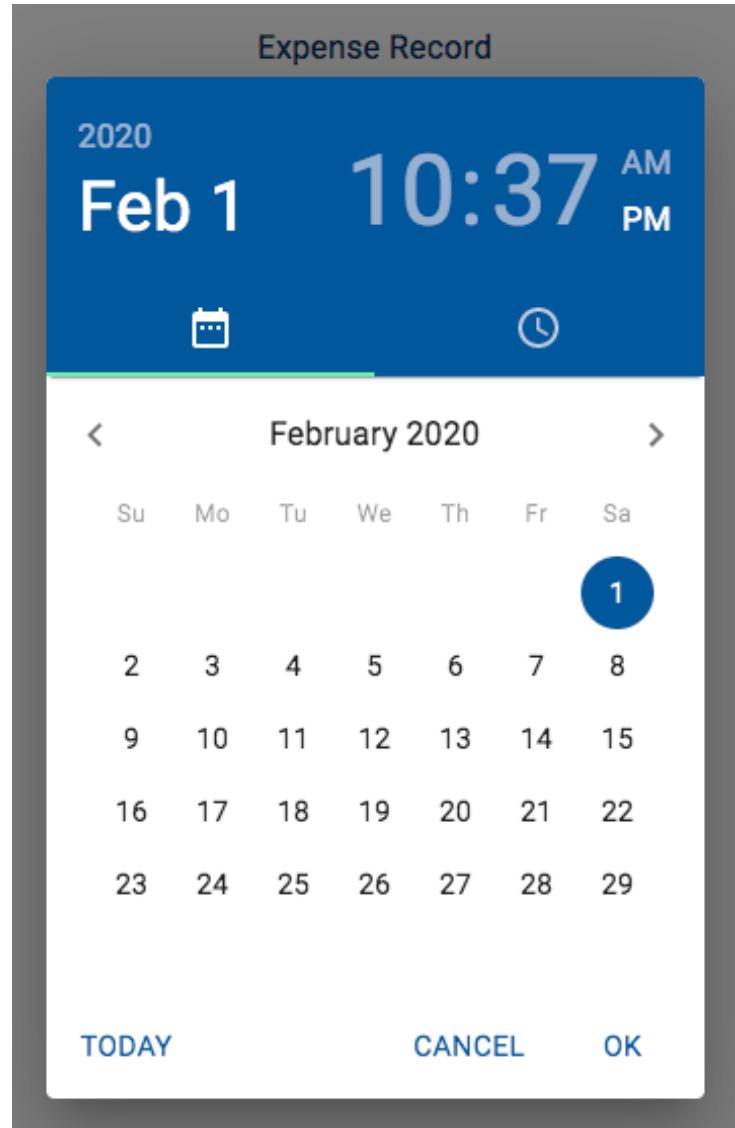
February 1st 09:34 p.m.

Notes

SUBMIT

CANCEL

The implementation for this `NewExpense` component is similar to other form implementations that we have discussed previously, such as the `Signup` component implementation from [Chapter 4, Adding a React Frontend to Complete MERN](#). The only different field in this form component is the date-time input for the Incurred on timing. Clicking on this field will render a date-time picker widget, as shown in the following screenshot:



To implement this date-time picker for the form, we will use Material-UI Pickers along with a date management library. Before we can integrate these libraries, we first need to install the following Material-UI Pickers and `date-fns` modules by running the following `yarn` command from the command line:

```
| yarn add @material-ui/pickers @date-io/date-fns@1.x date-fns
```

Once these modules are installed, we can import the required components and modules in the `NewExpense` component and add the date-time picker widget to the form, as shown in the following code.

mern-expense-tracker/client/expense/NewExpense.js:

```
import DateFnsUtils from '@date-io/date-fns'
import { DateTimePicker, MuiPickersUtilsProvider } from "@material-
ui/pickers"
...
<MuiPickersUtilsProvider utils={DateFnsUtils}>
  <DateTimePicker
    label="Incurred on"
    views={['year', 'month', 'date']}
    value={values.incurred_on}
    onChange={handleDateChange}
    showTodayButton
  />
</MuiPickersUtilsProvider>
```

This widget will render options to pick a year, month, date, and time along with a TODAY button to set the current time as the selected value. When the user is done picking a date-time, we will capture the value with the `handleDateChange` method and set it to state with the other expense-related values collected from the form. The `handleDateChange` method is defined as follows.

mern-expense-tracker/client/expense/NewExpense.js:

```
const handleDateChange = date = {
  setValues({...values, incurred_on: date })
}
```

Using this, we will have a `date` value set for the `incurred_on` field in the new expense record.

This `NewExpense` component can only be viewed by signed-in users. So, we will add a `PrivateRoute` in the `MainRouter` component, which will render this form only for authenticated users at `/expenses/new`.

mern-expense-tracker/client/MainRouter.js:

```
| PrivateRoute path="/expenses/new" component={NewExpense} /
```

This link can be added to any view, such as the `Menu` component, to be rendered conditionally when users are signed in. Now that it is possible to add new expense records in this expense tracking

application, in the next section, we will discuss the implementation to fetch and list these expenses from the database in the backend to the views in the frontend.

Listing expenses

In MERN Expense Tracker, users will be able to view the list of expenses that they already recorded on the application and incurred within a provided date range. In the following sections, we will add this ability by implementing a backend API to retrieve the list of expenses recorded by the currently signed-in user, and add a frontend view that will use this API to render the returned list of expenses to the end user.

The expenses by user API

We will implement an API to get the expenses recorded by a specific user and incurred between a provided date range. The request for this API will be received at `'/api/expenses'`, with the route defined in `expense.routes.js` as follows.

mern-expense-tracker/server/routes/expense.routes.js:

```
| router.route('/api/expenses')
|   .get(authCtrl.requireSignin, expenseCtrl.listByUser)
```

A `GET` request to this route will first ensure that the requesting user is signed in, before invoking the controller method to fetch the expenses from the database. In this application, users will only be able to view their own expenses. After the user authentication is confirmed, in the `listByUser` controller method we query the `Expense` collection in the database using date range specified in the request and the ID of the user who is signed in. The `listByUser` method is defined in the following code.

mern-expense-tracker/server/controllers/expense.controller.js:

```
| const listByUser = async (req, res) => {
|   let firstDay = req.query.firstDay
|   let lastDay = req.query.lastDay
|   try {
|     let expenses = await Expense.find({ '$and': [ { 'incurred_on':
|       { '$gte': firstDay, '$lte': lastDay } },
|         { 'recorded_by': req.auth._id } ] }).sort('incurred_on')
|         .populate('recorded_by', '_id name')
|         res.json(expenses)
|     } catch (err) {
|       console.log(err)
|       return res.status(400).json({
|         error: errorHandler.getErrorMessage(err)
|       })
|     }
|   }
```

In this method, we start by gathering the first day and the last day of the date range specified in the request query. From the database, we then retrieve the expenses incurred by the signed-in user within these dates. The signed-in user is matched against the user referenced in the `recorded_by` field. The `find` query against the `Expense` collection using these values will return matching expenses sorted by the `incurred_on` field, with the recently incurred expenses listed first.

The API to retrieve expenses recorded by a specific user can be used in the frontend to retrieve and display the expenses to the end user. To fetch this API in the frontend, we will add a corresponding `listByUser` method in `api-expense.js`, as shown in the following code.

mern-expense-tracker/client/expense/api-expense.js:

```
const listByUser = async (params, credentials, signal) => {
  const query = queryString.stringify(params)
  try {
    let response = await fetch('/api/expenses?' + query, {
      method: 'GET',
      signal: signal,
      headers: {
        'Accept': 'application/json',
        'Authorization': 'Bearer ' + credentials.token
      }
    })
    return await response.json()
  } catch (err) {
    console.log(err)
  }
}
```

In this method, before making the request to the list expenses API, we form the query string containing the date range with the `queryString` library. Then, this query string is attached to the request URL.

This fetch method will be used in the `Expenses` component to retrieve and show the expenses to the user. We will take a look at the implementation of the `Expenses` component in the next section.

The Expenses component

The list of expenses retrieved from the database will be rendered using a React component called `Expenses`. This component, on the initial load, will render the expenses incurred by the signed-in user in the current month. In this view, the user will also have the option to pick a date range to retrieve expenses incurred within specific dates, as shown in the following screenshot:

SHOWING RECORDS FROM
01/02/2020

TO
29/02/2020

GO

\$ 15	Brunch at Cafe Bistro	
Eating Out		
2/1/2020		
\$ 14	Metro Ticket	
Commute		
2/10/2020		
\$ 57	Trip tp Whole Foods	
Groceries	Fresh greens	
2/19/2020		

While defining the `Expenses` component, we first use a `useEffect` hook to make a fetch call to the list expenses API in order to retrieve the initial list of expenses. We also initialize the values that are necessary for making this request and for rendering the response to be received from the server, as shown in the following code.

`mern-expense-tracker/client/expense/Expenses.js`:

```

export default function Expenses() {
  const date = new Date(), y = date.getFullYear(), m = date.getMonth()
  const [firstDay, setFirstDay] = useState(new Date(y, m, 1))
  const [lastDay, setLastDay] = useState(new Date(y, m + 1, 0))

  const jwt = auth.isAuthenticated()
  const [redirectToSignin, setRedirectToSignin] = useState(false)
  const [expenses, setExpenses] = useState([])

  useEffect(() = {
    const abortController = new AbortController()
    const signal = abortController.signal
    listByUser({firstDay: firstDay, lastDay: lastDay},
      {t: jwt.token}, signal)
    .then((data) = {
      if (data.error) {
        setRedirectToSignin(true)
      } else {
        setExpenses(data)
      }
    })
    return function cleanup(){
      abortController.abort()
    }
  }, [])
...
}

```

We first determine the dates of the first day and the last day of the current month. These dates are set in the state to be rendered in the search form fields and provided as the date range query parameters in the request to the server. Because we will only fetch the expenses associated with the current user, we retrieve the signed-in user's `auth` credentials to be sent with the request. If the request to the server results in an error, we will redirect the user to the login page. Otherwise, we will set the received expenses in the state to be rendered in the view.

In the view part of the `Expenses` component, we will add a form to search by date range, before iterating through the resulting expenses array to render individual expense details. In the following sections, we will look at the implementation of the search form and expenses list in the component view.

Searching by date range

In the `Expenses` view, users will have the option to view a list of expenses incurred within a specific date range. To implement a search form that allows users to pick a start and end date for the range, we will use `DatePicker` components from Material-UI Pickers.

In the view, we will add two `DatePicker` components to collect the first day and the last day of the query range, and also add a button to initiate the search, as shown in the following code.

`mern-expense-tracker/client/expense/Expenses.js`:

```
div className={classes.search}
  <MuiPickersUtilsProvider utils={DateFnsUtils}>
    <DatePicker
      disableFuture
      format="dd/MM/yyyy"
      label="SHOWING RECORDS FROM"
      views={[ "year", "month", "date" ] }
      value={firstDay}
      onChange={handleSearchFieldChange('firstDay')}>
    />
    <DatePicker
      format="dd/MM/yyyy"
      label="TO"
      views={[ "year", "month", "date" ] }
      value={lastDay}
      onChange={handleSearchFieldChange('lastDay')}>
    />
  </MuiPickersUtilsProvider>
  Button variant="contained" color="secondary"
    onClick= {searchClicked} GO </Button>
</div>
```

When a user interacts with the `DatePicker` components to select a date, we will invoke the `handleSearchFieldChange` method to get the selected `date` value. This method gets the `date` value and sets it to either the `firstDay` or `lastDay` value in the state accordingly.

The `handleSearchFieldChange` method is defined in the following code.

`mern-expense-tracker/client/expense/Expenses.js`:

```
const handleSearchFieldChange = name = date = {
  if(name=='firstDay'){
    setFirstDay(date)
  }else{
    setLastDay(date)
  }
}
```

After the two dates are selected and set in state, when the user clicks on the Search button, we will invoke the `searchClicked` method. In this method, we make another call to the list expenses API with the new dates sent in the query parameters. The `searchClicked` method is defined as follows.

mern-expense-tracker/client/expense/Expenses.js:

```
const searchClicked = () = {
  listByUser({firstDay: firstDay, lastDay: lastDay}, {t:
jwt.token}).then((data) = {
  if (data.error) {
    setRedirectToSignin(true)
  } else {
    setExpenses(data)
  }
})
```

Once the expenses resulting from this new query are received from the server, we set it to the state to be rendered in the view. In the next section, we will look at the implementation for displaying this retrieved list of expenses.

Rendering expenses

In the `Expenses` component view, we iterate through the list of expenses retrieved from the database and display each expense record to the end user in a Material-UI `ExpansionPanel` component. In the `ExpansionPanel` component, we show details of the individual expense record in the *Summary* section. Then, on the expansion of the panel, we will give the user the option to edit details of the expense or delete the expense, as discussed in the next section.

In the following code added to the view code after the search form elements, we use `map` to iterate through the `expenses` array and render each `expense` in an `ExpansionPanel` component.

mern-expense-tracker/client/expense/Expenses.js:

```
{expenses.map((expense, index) = {
  return span key={index}
    <ExpansionPanel className={classes.panel}>
      <ExpansionPanelSummary
        expandIcon={ Edit / } >
        <div className={classes.info}
          Typography className={classes.amount} $ {expense.amount}
</Typography>
          <Divider style={{marginTop: 4, marginBottom: 4}}/>
          <Typography {expense.category} </Typography>
          <Typography className={classes.date}
            {new Date(expense.incurred_on).toLocaleDateString()}>
          </Typography>
        </div>
        <div>
          <Typography className={classes.heading} {expense.title}>
</Typography>
          <Typography className={classes.notes} {expense.notes}>
</Typography>
        </div>
      </ExpansionPanelSummary>
      <Divider/>
      <ExpansionPanelDetails style={{display: 'block'}}>
        ...
      </ExpansionPanelDetails>
    </ExpansionPanel>
  </span>
})
})}
```

The expense details are rendered in the `ExpansionPanelSummary` component, giving the user an overview of the expense that they recorded on the application. The `ExpansionPanelDetails` component will contain the options to modify the given expense and complete the feature allowing users to manage the expenses they have recorded on the application. In the next section, we will discuss the implementation of these options to modify the recorded expense.

Modifying an expense record

Users on MERN Expense Tracker will be able to modify the expenses they have already recorded on the application by either updating the details of an expense or deleting the expense record altogether.

In the frontend of the application, they will receive these modification options in the expenses list after expanding to see details of an individual expense in the list, as shown in the following screenshot:

The screenshot shows a form for modifying an expense record. At the top, it displays the amount '\$ 15' and the title 'Brunch at Cafe Bistro'. Below this, there are fields for 'Category' (set to 'Eating Out') and 'Incurred on' (set to '2/1/2020'). The main data section contains two rows: 'Title' (Brunch at Cafe Bistro) and 'Amount (\$)' (15). Underneath these, there are additional fields for 'Incurred on' (February 1st 03:22 p.m.) and 'Category' (Eating Out). A 'Notes' field is present with a blank line. At the bottom right, there is a blue 'UPDATE' button and a small trash can icon.

\$ 15	Brunch at Cafe Bistro
Eating Out	
2/1/2020	
Title	Amount (\$)
Brunch at Cafe Bistro	15
Incurred on	Category
February 1st 03:22 p.m.	Eating Out
Notes	

UPDATE trash can icon

To implement these expense modification features, we will have to update the view to render this form and the delete option. Additionally, we will add edit and delete expense API endpoints on the server. In the following sections, we will discuss how to render

these edit and delete elements in the frontend, and then implement the edit and delete APIs in the backend.

Rendering the edit form and delete option

We will render the edit expense form and delete option in the `Expenses` component view. For each expense record rendered in a Material-UI `ExpansionPanel` component in this view, we will add form fields in the `ExpansionPanelDetails` section, with each field pre-populated with the corresponding expense detail value. Users will be able to interact with these form fields to change the values and then click on the Update button to save the changes to the database. We will add these form fields in the view along with the Update button and delete option, as shown in the following code.

mern-expense-tracker/client/expense/Expenses.js:

```
<ExpansionPanelDetails style={{display: 'block'}}>
  <div>
    <TextField label="Title" value={expense.title}
              onChange={handleChange('title', index)}/>
    <TextField label="Amount ($)" value={expense.amount}
              onChange={handleChange('amount', index)} type="number"/>
  </div>
  <div>
    <MuiPickersUtilsProvider utils={DateFnsUtils}>
      <DateTimePicker
        label="Incurred on"
        views={[ "year", "month", "date" ]}
        value={expense.incurred_on}
        onChange={handleDateChange(index)}
        showTodayButton
      />
    </MuiPickersUtilsProvider>
    <TextField label="Category" value={expense.category}
              onChange={handleChange('category', index)}/>
  </div>
  <TextField label="Notes" multiline rows="2"
            value={expense.notes}
            onChange={handleChange('notes', index)} />
</div>
<div className={classes.buttons}
  { error && ( Typography component="p" color="error"
    <Icon color="error" className={classes.error} error </Icon>
    {error}>
```

```

        </Typography> )
    }
    { saved && Typography component="span" color="secondary" Saved
</Typography> }
    <Button color="primary" variant="contained"
        onClick={()= clickUpdate(index)} Update </Button>
    DeleteExpense expense={expense} onRemove={removeExpense}/
    </div>
</ExpansionPanelDetails>

```

The form fields added here are similar to the fields added in the `NewExpense` component to create new expense records. When the user interacts with these fields to update the values, we invoke the `handleChange` method with the corresponding index of the given expense in the `expenses` array, the name of the field, and the changed value. The `handleChange` method is defined in the following code.

`mern-expense-tracker/client/expense/Expenses.js`:

```

const handleChange = (name, index) = event = {
    const updatedExpenses = [...expenses]
    updatedExpenses[index][name] = event.target.value
    setExpenses(updatedExpenses)
}

```

The expense object at the given index in the `expenses` array is updated with the changed value of the specified field and set to state. This will render the view with the latest values as the user is updating the edit form. When the user is done making changes and clicks on the `Update` button, we will invoke the `clickUpdate` method, which is defined as follows.

`mern-expense-tracker/client/expense/Expenses.js`:

```

const clickUpdate = (index) = {
    let expense = expenses[index]
    update({
        expenseId: expense._id
    }, {
        t: jwt.token
    }, expense)
    .then((data) = {
        if (data.error) {
            setError(data.error)
        } else {
            setSaved(true)
            setTimeout(()= {setSaved(false)}, 3000)
        }
    })
}

```

```
| } }
```

In this `clickUpdate` method, we send the updated expense to the backend in a fetch call to an edit expense API. The implementation of this edit expense API is discussed in the next section.

The `DeleteExpense` component added to the edit form renders a Delete button and uses the `expense` object passed as a prop to delete the associated expense from the database by calling the delete expense API. The implementation for this `DeleteExpense` is similar to the `DeleteShop` component discussed in [Chapter 7, *Exercising MERN Skills with an Online Marketplace*](#). In the next section, we will discuss the implementation of the edit and delete expense APIs used by the edit form and delete the option to relay the expense-related updates made by the user to the Expense collection in the database.

Editing and deleting an expense in the backend

In order to complete the edit and delete expense operations initiated by signed-in users from the frontend, we need to have the corresponding APIs in the backend. The route for these API endpoints that will accept the update and delete requests can be declared in the following code.

mern-expense-tracker/server/routes/expense.routes.js:

```
router.route('/api/expenses/:expenseId')
  .put(authCtrl.requireSignin, expenseCtrl.hasAuthorization,
    expenseCtrl.update)
  .delete(authCtrl.requireSignin, expenseCtrl.hasAuthorization,
    expenseCtrl.remove)
router.param('expenseId', expenseCtrl.expenseByID)
```

A `PUT` or `DELETE` request to this route will first ensure that the current user is signed in with the `requireSignin` auth controller method, before checking authorization and performing any operations in the database.

The `:expenseId` parameter in the route URL, `/api/expenses/:expenseId`, will invoke the `expenseByID` controller method, which is similar to the `userByID` controller method. It retrieves the expense from the database and attaches it to the request object to be used in the `next` method. The `expenseByID` method is defined in the following code.

mern-expense-tracker/server/controllers/expense.controller.js:

```
const expenseByID = async (req, res, next, id) => {
  try {
    let expense = await Expense.findById(id).populate
      ('recorded_by', '_id name').exec()
    if (!expense)
      return res.status('400').json({
```

```
        error: "Expense record not found"
    })
    req.expense = expense
    next()
} catch (err) {
    return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
    })
}
}
```

The expense object retrieved from the database will also contain the name and ID details of the user who recorded the expense, as we specified in the `populate()` method. For these API endpoints, next, we verify that this expense object was actually recorded by the signed-in user with the `hasAuthorization` method, which is defined in the expense controller as follows.

mern-expense-tracker/server/controllers/expense.controller.js

```
const hasAuthorization = (req, res, next) = {
  const authorized = req.expense && req.auth &&
    req.expense.recorded_by._id == req.auth._id
  if (!authorized) {
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}
```

Once it has been confirmed that the user trying to update the expense is the one who recorded it and if it is a `PUT` request, then the `update` method is invoked next to update the expense document with the new changes in the Expense collection. The `update` controller method is defined in the following code.

mern-expense-tracker/server/controllers/expense.controller.js

```
const update = async (req, res) => {
  try {
    let expense = req.expense
    expense = extend(expense, req.body)
    expense.updated = Date.now()
    await expense.save()
    res.json(expense)
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

```
        return res.status(400).json({
          error: errorHandler.getErrorMessage(err)
        })
    }
}
```

The method retrieves the expense details from `req.expense`, then uses the `lodash` module to extend and merge the changes that came in the request body to update the expense data. Before saving this updated expense to the database, the `updated` field is populated with the current date to reflect the last updated timestamp. On the successful save of this update, the updated expense object is sent back in the response.

If it is a `DELETE` request instead of a `PUT` request, the `remove` method is invoked instead in order to delete the specified expense document from the collection in the database. The `remove` controller method is defined in the following code.

mern-expense-tracker/server/controllers/expense.controller.js:

```
const remove = async (req, res) = {
  try {
    let expense = req.expense
    let deletedExpense = await expense.remove()
    res.json(deletedExpense)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The `remove` operation in this method will permanently delete the expense from the application.

We have all the features in place for users on the application to start recording and managing their day-to-day expenses. We defined an Expense model for storing expense data, and backend APIs and frontend views for creating new expenses, displaying a list of expenses for a given user, and modifying an existing expense. We are now ready to implement data visualization features based on the

expense data that will be recorded by users on the application over time. We will discuss these implementations in the next section.

Visualizing expense data over time

Aside from allowing users to keep logs of their expenses, the MERN Expense Tracker application will process the collected expense data to give users insights into their spending habits over time. We will implement simple data aggregation and visualization features to demonstrate how the MERN stack can accommodate such requirements in any full-stack application. To enable these features, we will utilize MongoDB's aggregation framework and also the React-based charting and data visualization library—Victory—by Formidable.

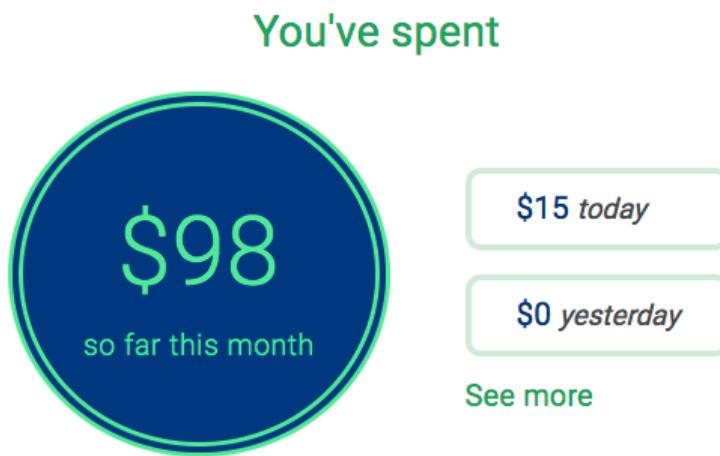
In the following sections, we will first add features to summarize a user's expenses in the current month and also show how they are doing compared to previous months. Then, we will add different Victory charts to give them a visual representation of their spending patterns over a month, and a year, and per expense category.

Summarizing recent expenses

When a user signs in to their account on the application, they will see a preview of the expenses they incurred so far in the current month. They will also see a comparison of how much more or less they are spending in each category in comparison to the averages from previous months. To implement these features, we will have to add backend APIs that will run aggregation operations on the relevant expense data in the database and return the computed results to be rendered in the frontend. In the following sections, we will implement the full-stack slices—first to show a preview of all the expenses incurred so far in the current month, and then a comparison of the average expenses per category with respect to expenditures in the current month.

Previewing expenses in the current month

After a user signs in to the application, we will show a preview of their current expenses, including their total expenditure for the current month and how much they spent on the current date and the day before. This preview will be displayed to the end user, as shown in the following screenshot:



In order to implement this feature, we need to add a backend API that will process the existing expense data to return these three values, so it can be rendered in a React component. In the following sections, we will take a look at the implementation and integration of this API with a frontend view to complete this preview feature.

The current month preview API

We will add an API to the backend that will return the preview of expenses incurred so far in the current month. To implement this API, we will first declare a `GET` route, as shown in the following code.

`mern-expense-tracker/server/routes/expense.routes.js`:

```
| router.route('/api/expenses/current/preview')
|   .get(authCtrl.requireSignin, expenseCtrl.currentMonthPreview)
```

A `GET` request to this route at `'/api/expenses/current/preview'` will first ensure the requesting client is a signed-in user, and then it will invoke the `currentMonthPreview` controller method. In this method, we will use MongoDB's aggregation framework to perform three sets of aggregations on the `Expense` collection and retrieve the total expenses for the current month, the current date, and the day before.

The `currentMonthPreview` controller method will be defined with the following structure, where we first determine the dates needed to find matching expenses, and then we perform the aggregations before returning the results in the response.

`mern-expense-tracker/server/controllers/expense.controller.js`:

```
const currentMonthPreview = async (req, res) => {
  const date = new Date(), y = date.getFullYear(), m = date.getMonth()
  const firstDay = new Date(y, m, 1)
  const lastDay = new Date(y, m + 1, 0)

  const today = new Date()
  today.setUTCHours(0,0,0,0)

  const tomorrow = new Date()
  tomorrow.setUTCHours(0,0,0,0)
  tomorrow.setDate(tomorrow.getDate() + 1)

  const yesterday = new Date()
  yesterday.setUTCHours(0,0,0,0)
  yesterday.setDate(yesterday.getDate() - 1)
```

```

        try {
            /* ... Perform aggregation operations on the Expense collection
               to compute current month's numbers ... */
            /* ... Send computed result in response ... */
        } catch (err) {
            console.log(err)
            return res.status(400).json({
                error: errorHandler.getErrorMessage(err)
            })
        }
    }
}

```

We first determine the dates for the current month's first day and last day, then the dates for today, tomorrow, and yesterday with the minutes and seconds set to zero. We will need these dates to specify the ranges for finding the matching expenses that were incurred in the current month, today, and yesterday. Then, with these values and the signed-in user's ID reference, we construct the aggregation pipelines necessary to retrieve the total expenses for the current month, today, and yesterday. We group these three different aggregation pipelines using the `$facet` stage in MongoDB's aggregation framework, as shown in the following code.

mern-expense-tracker/server/controllers/expense.controller.js:

```

let currentPreview = await Expense.aggregate([
    { $facet: { month: [
        { $match: { incurred_on: { $gte: firstDay, $lt: lastDay },
                    recorded_by: mongoose.Types.ObjectId(req.auth._id) }},
        { $group: { _id: "currentMonth" , totalSpent: {$sum: "$amount"} } },
        ],
        today: [
            { $match: { incurred_on: { $gte: today, $lt: tomorrow },
                        recorded_by: mongoose.Types.ObjectId(req.auth._id) }},
            { $group: { _id: "today" , totalSpent: {$sum: "$amount"} } },
            ],
        yesterday: [
            { $match: { incurred_on: { $gte: yesterday, $lt: today },
                        recorded_by: mongoose.Types.ObjectId(req.auth._id) }},
            { $group: { _id: "yesterday" , totalSpent: {$sum: "$amount"} } }
            ],
        ]
    }})
let expensePreview = {month: currentPreview[0].month[0], today:
    currentPreview[0].today[0], yesterday: currentPreview[0].yesterday[0] }
res.json(expensePreview)

```

For each aggregation pipeline, we first match the expenses using the date range values for the `incurred_on` field, and also the `recorded_by` field with the current user's reference, so the aggregation is only performed on the expenses recorded by the current user. Then, the matching expenses in each pipeline are grouped to calculate the total amount spent.



In the faceted aggregation operation result, each pipeline has its own field in the output document where the results are stored as an array of documents.

After the aggregation operations are completed, we access the computed results and compose the response to be sent back in the response to the requesting client. This API can be used in the frontend with a fetch request. You can define a corresponding fetch method to make the request, similar to other API implementations. Then, the fetch method can be used in a React component to retrieve and render these aggregated values to the user. In the next section, we will discuss the implementation of this view to render the preview of current expenses for a user.

Rendering the preview of current expenses

We can give the user a glimpse of their current expenses in any React component, which is accessible to a signed-in user and added to the frontend of the application. To retrieve the expense totals and render these in the view, we can call the current month preview API either in a `useEffect` hook or when a button is clicked on.

In the MERN Expense Tracker application, we render these details in a React component that is added to the home page. We use a `useEffect` hook, as shown in the following code, to retrieve the current expense preview data.

mern-expense-tracker/client/expense/ExpenseOverview.js:

```
useEffect(() = {
  const abortController = new AbortController()
  const signal = abortController.signal
  currentMonthPreview({t: jwt.token}, signal).then((data) = {
    if (data.error) {
      setRedirectToSignin(true)
    } else {
      setExpensePreview(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

Once the data is received from the backend, we set it to state in a variable called `expensePreview`, so the information can be displayed in the view. In the view of the component, we use this state variable to compose an interface with these details as desired. In the following code, we render the total expenses for the current month, for the current date, and for the day before.

mern-expense-tracker/client/expense/ExpenseOverview.js:

```
<Typography variant="h4" color="textPrimary"> You've spent </Typography>
<div>
<Typography component="span"
    ${expensePreview.month ? expensePreview.month.totalSpent : '0'}
    span so far this month </span>
</Typography>
<div>
    <Typography variant="h5" color="primary"
        ${expensePreview.today ? expensePreview.today.totalSpent : '0'}
        span today </span>
    </Typography>
    <Typography variant="h5" color="primary"
        ${expensePreview.yesterday
            ? expensePreview.yesterday.totalSpent: '0'}
        <span className={classes.day} yesterday </span>
    </Typography>
    <Link to="/expenses/all" Typography variant="h6"> See more
</Typography>  </Link>
    </div>
</div>
```

These values are only rendered if the corresponding value is returned in the aggregation results from the backend; otherwise, we render a "0."

With this current expenses preview feature implemented, we are able to process the expense data recorded by the user to give them an idea of how much they are spending currently. In the next section, we will follow similar implementation steps to inform the user about their spending status for each expense category.

Tracking current expenses by category

In this application, we will give the user an overview of how much they are currently spending in each expense category in comparison to previous averages. For each category, we will display the monthly average based on previous expense data, show the total spent so far in the current month, and show the difference to indicate whether they are spending extra or are saving money in the current month. The following screenshot shows what this feature will look like to the end user for their expense data:

Eating Out		
past average	this month	saved
\$15	\$15	\$0
Groceries		
past average	this month	spent extra
\$38.5	\$57	\$18.5
Commute		
past average	this month	saved
\$19.5	\$14	\$5.5

To implement this feature, we need to add a backend API that will process the existing expense data to return the monthly average

along with the total spent in the current month for each category, so it can be rendered in a React component. In the following sections, we will look at the implementation and integration of this API and frontend view to complete this feature to track expenses by category.

The current expenses by category API

We will add an API to the backend that will return the average monthly expenses and the total spent in the current month for each expense category. To implement this API, we will first declare a `GET` route, as shown in the following code.

`mern-expense-tracker/server/routes/expense.routes.js`:

```
| router.route('/api/expenses/by/category')
|   .get(authCtrl.requireSignin, expenseCtrl.expenseByCategory)
```

A `GET` request to this route at `'/api/expenses/by/category'` will first ensure that the requesting client is a signed-in user, and then it will invoke the `expenseByCategory` controller method. In this method, we will use different features of MongoDB's aggregation framework to separately calculate the monthly expense averages for each category and the total spent in the current month per category, before combining the two results to return these two values associated with each category to the requesting client.

The `expenseByCategory` controller method will be defined with the following structure, where we first determine the dates required to find matching expenses, and then we perform the aggregations before returning the results in the response.

`mern-expense-tracker/server/controllers/expense.controller.js`:

```
| const expenseByCategory = async (req, res) => {
|   const date = new Date(), y = date.getFullYear(), m = date.getMonth()
|   const firstDay = new Date(y, m, 1)
|   const lastDay = new Date(y, m + 1, 0)
|
|   try {
|     let categoryMonthlyAvg = await Expense.aggregate([/*... aggregation
|     ... */]).exec()
```

```

        res.json(categoryMonthlyAvg)
    } catch (err) {
        console.log(err)
        return res.status(400).json({
            error: errorHandler.getErrorMessage(err)
        })
    }
}

```

In this method, we will use an aggregation pipeline containing a `$facet` with two sub-pipelines for calculating the monthly average per category and the total spent per category in the current month. Then, we take these two resulting arrays from the sub-pipelines to merge the results. The code for this aggregation pipeline is defined in the following code.

`mern-expense-tracker/server/controllers/expense.controller.js`:

```

[
  { $facet: {
      average: [
        { $match: { recorded_by: mongoose.Types.ObjectId(req.auth._id) }},
        { $group: { _id: {category: "$category", month: {$month: "$incurred_on"}}, totalSpent: {$sum: "$amount"} } },
        { $group: { _id: "$_id.category", avgSpent: { $avg: "$totalSpent" }}},
        { $project: {
            _id: "$_id", value: {average: "$avgSpent"}, }
        }
      ],
      total: [
        { $match: { incurred_on: { $gte: firstDay, $lte: lastDay }, recorded_by: mongoose.Types.ObjectId(req.auth._id) }},
        { $group: { _id: "$category", totalSpent: {$sum: "$amount"} } },
        { $project: {
            _id: "$_id", value: {total: "$totalSpent"}, }
        }
      ]
    },
    { $project: {
        overview: { $setUnion:['$average','$total'] },
    }},
    { $unwind: '$overview' },
    { $replaceRoot: { newRoot: "$overview" } },
    { $group: { _id: "$_id", mergedValues: { $mergeObjects: "$value" } } }
  ]
]

```

While projecting the output of the sub-pipelines in the `$facet` stage, we make sure that the keys of the result objects are `_id` and `value` in both output arrays, so they can be merged uniformly. Once the faceted aggregation operations are done, we use a `$setUnion` on the results to combine the arrays. Then, we make the resulting combined array the new root document in order to run a `$group` aggregation on it to merge the values for the averages and totals per category.

The final output from this aggregation pipeline will contain an array with an object for each expense category. Each object in this array will have the category name as the `_id` value and a `mergedValues` object containing the average and total values for the category. Then, this final output array generated from the aggregation is sent back in the response to the requesting client.

We can use this API in the frontend with a fetch request. You can define a corresponding fetch method to make the request, similar to other API implementations. Then, the fetch method can be used in a React component to retrieve and render these aggregated values to the user. In the next section, we will discuss the implementation of this view to render the comparison of expenses in each category by a user in the current month versus previous months.

Rendering an overview of expenses per category

Besides informing the user of how much they are spending currently, we can give them an idea of how they are doing in comparison to previous expenditures. We can tell them whether they are spending more or saving money in the current month for each category. We can implement a React component, that calls the current expenses by category API to render the average and total values sent by the backend and also displays the computed difference between these two values.

The API can be fetched either in a `useEffect` hook or when a button is clicked on. In the MERN Expense Tracker application, we render these details in a React component that is added to the home page. We use a `useEffect` hook, as shown in the following code, to retrieve the expenses per category data.

`mern-expense-tracker/client/expense/ExpenseOverview.js`:

```
useEffect(() = {
  const abortController = new AbortController()
  const signal = abortController.signal
  expenseByCategory({t: jwt.token}, signal).then((data) = {
    if (data.error) {
      setRedirectToSignin(true)
    } else {
      setExpenseCategories(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

We will set the values received from the backend to the state in an `expenseCategories` variable, and render its details in the view. This variable will contain an array, which we will iterate through in the

view code to display three values for each category—the monthly average, the current month's total expenditure, and the difference between the two with an indication of whether money was saved or not.

In the following code, we use a `map` to iterate over the received data array and, for each item in the array, generate the view to display the average and total values received with the item. Besides this, we also show a computed value using these two values.

mern-expense-tracker/client/expense/ExpenseOverview.js:

```
{expenseCategories.map((expense, index) = {
  return( div key={index}
    <Typography variant="h5" {expense._id} </Typography>
    <Divider style={{ backgroundColor:
      indicateExpense(expense.mergedValues) }}/>
    <div>
      <Typography component="span" past average </Typography>
      <Typography component="span" this month </Typography>
      <Typography component="span" {expense.mergedValues.total
        && expense.mergedValues.total-
          expense.mergedValues.average > 0 ?
            "spent extra" : "saved" }>
      </Typography>
    </div>
    <div>
      <Typography component="span" ${expense.mergedValues.average}>
    </Typography>
    <Typography component="span" ${expense.mergedValues.total ?
      expense.mergedValues.total : 0}>
    </Typography>
    <Typography component="span" ${expense.mergedValues.total ?
      Math.abs(expense.mergedValues.total-
        expense.mergedValues.average) :
        expense.mergedValues.average}>
    </Typography>
  </div>
  <Divider/>
</div> )
})
})}
```

For each item in the array, we first render the category name, then the headings of the three values we will display. The third heading is rendered conditionally depending on whether the current total is more or less than the monthly average. Then, under each heading,

we render the corresponding values for the monthly average, the current total—which will be zero if no value was returned—and then the difference between this average and the total. For the third value, we render the absolute value of the computed difference between the average and total values using `Math.abs()`.

Based on this difference, we also render the divider under the category name with different colors to indicate whether money was saved, extra money was spent, or the same amount of money was spent. To determine the color, we define a method called `indicateExpense`, as shown in the following code:

```
const indicateExpense = (values) => {
  let color = '#4f83cc'
  if(values.total){
    const diff = values.total - values.average
    if( diff < 0){
      color = '#e9858b'
    }
    if( diff > 0 ){
      color = '#2bbd7e'
    }
  }
  return color
}
```

A different color is returned if the current total is more than, less than, or equal to the monthly average. This gives the user a quick visual indicator of how they are faring in terms of incurring expenses per category for the current month.

We have added simple data visualization features to the expense tracking application by utilizing existing capabilities of MERN stack technologies such as the aggregation framework in MongoDB. In the next section, we will demonstrate how to add even more complex data visualization features into this application by integrating an external charting library.

Displaying expense data charts

Graphs and charts are time-tested mechanisms for visualizing complex data patterns. In the MERN Expense Tracker application, we will add simple charts using Victory to report expense patterns over time in graphical representations to the user.



Victory is an open source charting and data visualization library for React and React Native developed by Formidable. Different types of charts are available as modular components that can be customized and added to any React application. To learn more about Victory, visit <https://formidable.com/open-source/victory>.

Before we get started with integrating Victory charts in the code, we will need to install the module by running the following command from the command line:

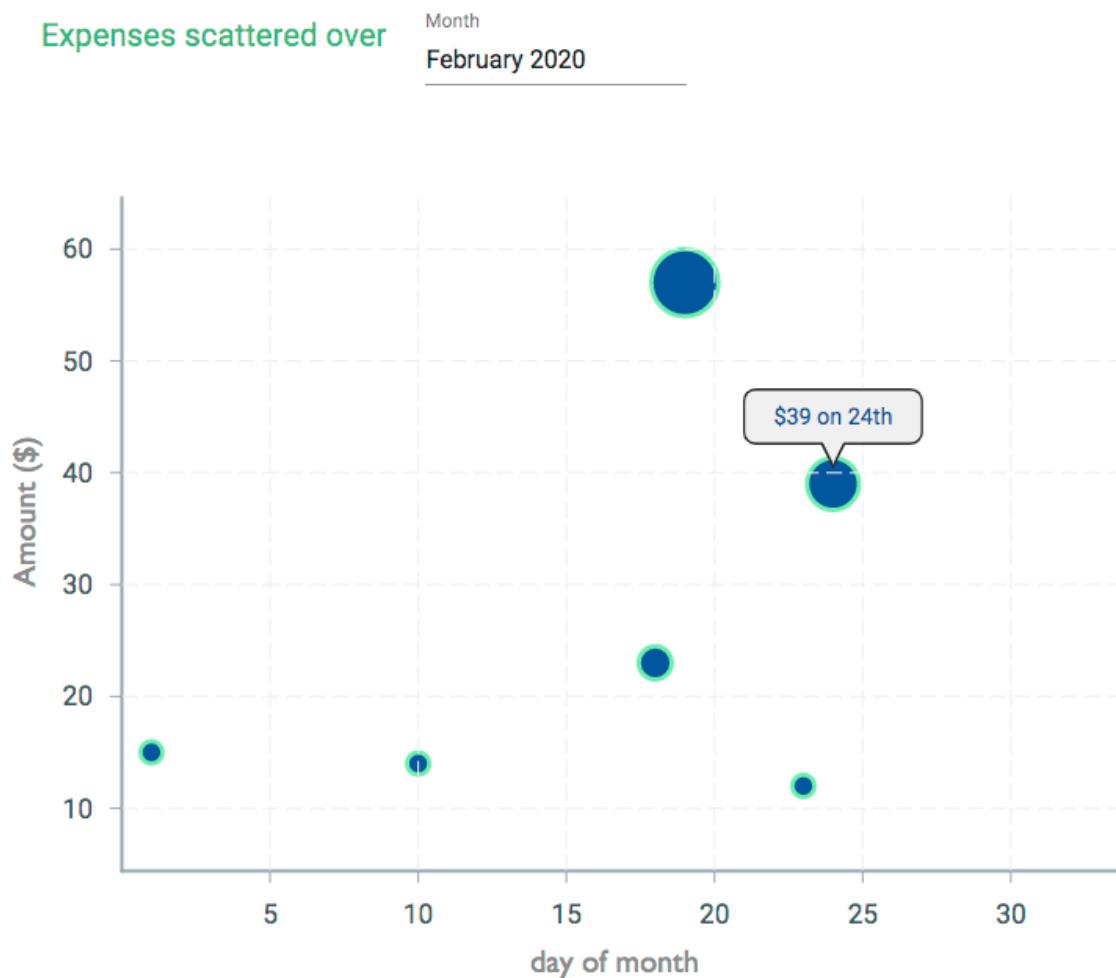
```
| yarn add victory
```

In the expense tracking application, we will add three different charts as a part of the interactive expense reports presented to the user. The three charts will include a scatter plot showing the expenses incurred in a given month, a bar chart showing the total expenses incurred per month in a given year, and a pie chart showing the average expenditure per category within a provided date range.

For each chart, we will add a corresponding backend API to retrieve the relevant expense data and a React component to the frontend that will use the retrieved data to render the associated Victory chart. In the following sections, we will implement the full-stack slices necessary to add a scatter plot chart for a month's expenses, a bar chart showing a year's monthly expenses, and a pie chart displaying the average expenses per category over a given period of time.

A month's expenses in a scatter plot

We will show the expenses incurred by a user over a given month in a scatter plot. This will provide them with a visual overview of how their expenses pan out over a month. The following screenshot shows how the scatter plot will render with user expense data:



We plot the expense amounts versus the day of the month when it was incurred on the y axis and x axis, respectively. Hovering over a

plotted bubble displays how much was spent on which date for that specific expense record. In the following sections, we will implement this feature by first adding a backend API that will return the expenses for the given month in the format needed to render it in a Victory Scatter chart. Then, we will add a React component that will retrieve this data from the backend and render it in the Victory Scatter chart.

The scatter plot data API

We will add an API to the backend that will return the expenses incurred over a given month in the data format needed to render the scatter chart in the frontend. To implement this API, we will first declare a `GET` route, as shown in the following code.

`mern-expense-tracker/server/routes/expense.routes.js`:

```
| router.route('/api/expenses/plot')
|   .get(authCtrl.requireSignin, expenseCtrl.plotExpenses)
```

A `GET` request to this route at `'/api/expenses/plot'` will first ensure that the requesting client is a signed-in user, and then it will invoke the `plotExpenses` controller method. The request will also take the value of the given month in a URL query parameter, which will be used in the `plotExpenses` method to determine the dates of the first day and the last day of the provided month. We will need these dates to specify the range for finding the matching expenses that were incurred in the specified month and recorded by the authenticated user while aggregating the expenses into the data format needed for the chart. The `plotExpenses` method is defined in the following code.

`mern-expense-tracker/server/controllers/expense.controller.js`:

```
| const plotExpenses = async (req, res) => {
|   const date = new Date(req.query.month), y = date.getFullYear(), m =
|   date.getMonth()
|   const firstDay = new Date(y, m, 1)
|   const lastDay = new Date(y, m + 1, 0)
|
|   try {
|
|     let totalMonthly = await Expense.aggregate([
|       { $match: { incurred_on: { $gte: firstDay, $lt: lastDay },
|                  recorded_by: mongoose.Types.ObjectId(req.auth._id)
|                } },
|       { $project: {x: {$dayOfMonth: '$incurred_on'}, y: '$amount' } }
|     ]).exec()
|   }
```

```
    res.json(totalMonthly)

} catch (err) {
  console.log(err)
  return res.status(400).json({
    error: errorHandler.getErrorMessage(err)
  })
}

}
```

We run a simple aggregation operation that finds the matching expenses and returns an output containing the values in the format needed for the `y` axis and `x` axis values of the scatter chart. The final result of the aggregation contains an array of objects, with each object containing an `x` attribute and a `y` attribute. The `x` attribute contains the day of the month value from the `incurred_on` date. The `y` attribute contains the corresponding expense amount. This final output array generated from the aggregation is sent back in the response to the requesting client.

We can use this API in the frontend with a `fetch` request. You can define a corresponding `fetch` method to make the request, similar to other API implementations. Then, the `fetch` method can be used in a React component to retrieve and render this array of `x` and `y` values in a scatter plot chart. In the next section, we will discuss the implementation of this view to render a scatter chart showing the expenses incurred over a given month.

The MonthlyScatter component

We will implement a React component that calls the scatter plot data API to render the received array of expenses incurred over a given month in a Victory Scatter chart.

The API can be fetched either in a `useEffect` hook or when a button is clicked on. In the MERN Expense Tracker application, we render this scatter chart in a React component called `MonthlyScatter`. When this component loads, we render a scatter chart for expenses in the current month. We also add a `DatePicker` component to allow users to select the desired month and retrieve data for that month with a button click. In the following code, we retrieve the initial scatter plot data with a `useEffect` hook when the component loads.

mern-expense-tracker/client/report/MonthlyScatter.js:

```
const [plot, setPlot] = useState([])
const [month, setMonth] = useState(new Date())
const [error, setError] = useState('')
const jwt = auth.isAuthenticated()
useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal

    plotExpenses({month: month}, {t: jwt.token}, signal).then((data) =>
    {
        if (data.error) {
            setError(data.error)
        } else {
            setPlot(data)
        }
    })
    return function cleanup() {
        abortController.abort()
    }
}, [])
```

When the plotted data is received from the backend and set in the state, we can render it in a Victory Scatter chart. Additionally, we can

add the following code in the component view to render a customized scatter chart with labels.

mern-expense-tracker/client/report/MonthlyScatter.js:

```
<VictoryChart
  theme={VictoryTheme.material}
  height={400}
  width={550}
  domainPadding={40}

  <VictoryScatter
    style={{
      data: { fill: "#01579b", stroke: "#69f0ae", strokeWidth: 2 },
      labels: { fill: "#01579b", fontSize: 10, padding: 8 }
    }}
    bubbleProperty="y"
    maxBubbleSize={15}
    minBubbleSize={5}
    labels={({ datum }) = ` $$ {datum.y} on ${datum.x}th`}
    labelComponent={ VictoryTooltip/ }
    data={plot}
    domain={{x: [0, 31]}}
  />
  <VictoryLabel
    textAnchor="middle"
    style={{ fontSize: 14, fill: '#8b8b8b' }}
    x={270} y={390}
    text={`day of month`}
  />
  <VictoryLabel
    textAnchor="middle"
    style={{ fontSize: 14, fill: '#8b8b8b' }}
    x={6} y={190}
    angle = {270}
    text={`${Amount ($)} `}
  />
</VictoryChart>
```

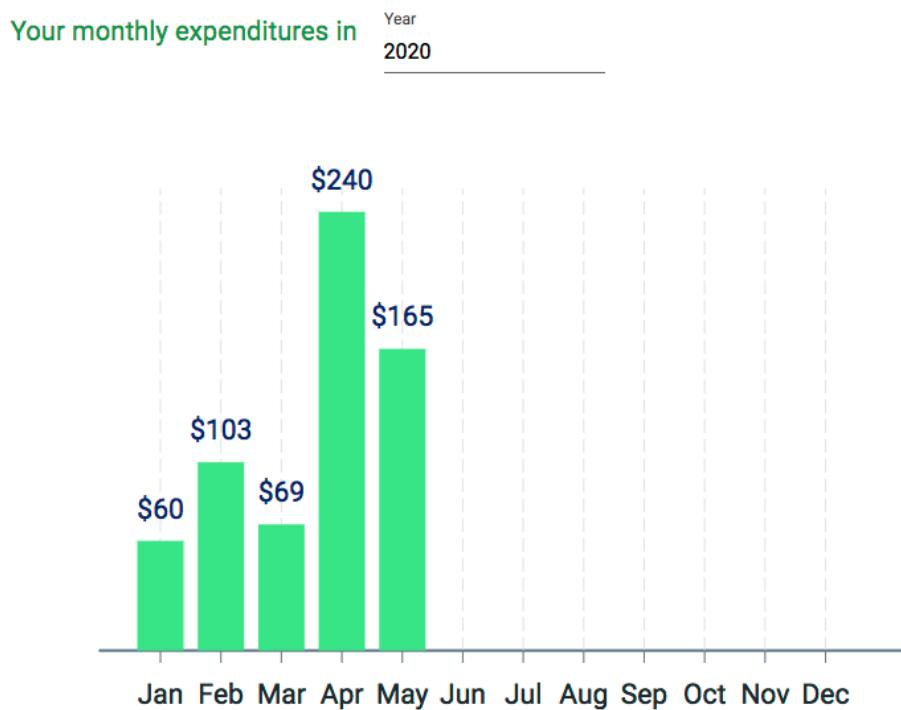
We place a `VictoryScatter` component in a `VictoryChart` component, giving us the flexibility to customize the scatter chart wrapper and place axis label texts outside the scatter chart. We pass the data to `VictoryScatter`, indicate which value the bubble property is based on, customize the styles, and specify the size range and labels for each bubble.

This code plots and renders the scatter chart against the data provided with the amount spent versus the day of the month on the y axis and x axis, respectively. In the next section, we will follow similar

steps to add a bar chart to graphically display the monthly expenses in a given year.

Total expenses per month in a year

We will show the user a bar chart representing their total monthly expenses over a given year. This will give them an overview of how their expenses are spread out annually. The following screenshot shows how the bar chart will render with user expense data:



Here, we populate the bar chart with the total expense value corresponding to each month in a given year. We add the monthly total value as labels to each bar. On the x-axis, we show the short name of each month. In the following sections, we will implement this feature by first adding a backend API that will return the total expenses incurred per month over a given year and in the format needed to render it in a Victory Bar chart. Then, we will add a React

component that will retrieve this data from the backend and render it in the Victory Bar chart.

The yearly expenses API

We will add an API to the backend that will return the total monthly expenses incurred over a given year in the data format needed to render the bar chart in the frontend.

To implement this API, we will first declare a `GET` route, as shown in the following code.

`mern-expense-tracker/server/routes/expense.routes.js`:

```
| router.route('/api/expenses/yearly')
|   .get(authCtrl.requireSignin, expenseCtrl.yearExpenses)
```

A `GET` request to this route at `'/api/expenses/yearly'` will first ensure that the requesting client is a signed-in user, and then it will invoke the `yearlyExpenses` controller method. The request will also take the value of the given year in a URL query parameter, which will be used in the `yearlyExpenses` method to determine the dates of the first day and the last day of the provided year. We will need these dates to specify the range for finding the matching expenses that were incurred in the specified year and recorded by the authenticated user while aggregating the total monthly expenses into the data format needed for the chart. The `yearlyExpenses` method is defined in the following code.

`mern-expense-tracker/server/controllers/expense.controller.js`:

```
| const yearlyExpenses = async (req, res) = {
|   const y = req.query.year
|   const firstDay = new Date(y, 0, 1)
|   const lastDay = new Date(y, 12, 0)
|   try {
|     let totalMonthly = await Expense.aggregate([
|       { $match: { incurred_on: { $gte: firstDay, $lt: lastDay } } },
|       { $group: { _id: { $month: "$incurred_on" }, totalSpent: { $sum:
|         "$amount" } } },
|       { $project: { x: '_id', y: '$totalSpent' } }
|     ]).exec()
|     res.json({monthTot:totalMonthly})
```

```
    } catch (err) {
      console.log(err)
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
}
```

We run an aggregation operation that finds the matching expenses, groups the expenses by month to calculate the total, and returns an output containing the values in the format needed for the `y` axis and `x` axis values of the bar chart. The final result of the aggregation contains an array of objects, with each object containing an `x` attribute and a `y` attribute.

The `x` attribute contains the month value from the `incurred_on` date. The `y` attribute contains the corresponding total expense amount for that month. This final output array generated from the aggregation is sent back in the response to the requesting client.

We can use this API in the frontend with a `fetch` request. You can define a corresponding `fetch` method to make the request, similar to other API implementations. Then, the `fetch` method can be used in a React component to retrieve and render this array of `x` and `y` values in a bar chart. In the next section, we will discuss the implementation of this view to render a bar chart showing the total monthly expenses incurred over a given year.

The YearlyBar component

We will implement a React component that calls the yearly expenses data API to render the received array of expenses incurred monthly over a given year in a Victory Bar chart.

The API can be fetched either in a `useEffect` hook or when a button is clicked on. In the MERN Expense Tracker application, we render this bar chart in a React component called `YearlyBar`. When this component loads, we render a bar chart for expenses in the current year. We also add a `DatePicker` component to allow users to select the desired year and retrieve data for that year with a button click. In the following code, we retrieve the initial yearly expense data with a `useEffect` hook when the component loads.

mern-expense-tracker/client/report/YearlyBar.js:

```
const [year, setYear] = useState(new Date())
const [yearlyExpense, setYearlyExpense] = useState([])
const [error, setError] = useState('')
const jwt = auth.isAuthenticated()
useEffect(() => {
    const abortController = new AbortController()
    const signal = abortController.signal
    yearlyExpenses({year: year.getFullYear()}, {t: jwt.token},
    signal).then((data) => {
        if (data.error) {
            setError(data.error)
        }
        setYearlyExpense(data)
    })
    return function cleanup() {
        abortController.abort()
    }
}, [])
```

With the data received from the backend and set in the state, we can render it in a Victory Bar chart. We can add the following code in the component view to render a customized bar chart with labels and only the x axis displayed.

mern-expense-tracker/client/report/YearlyBar.js:

```
const monthStrings = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
<VictoryChart
  theme={VictoryTheme.material}
  domainPadding={10}
  height={300}
  width={450}
  >
  <VictoryAxis/>
  <VictoryBar
    categories={{
      x: monthStrings
    }}
    style={{ data: { fill: "#69f0ae", width: 20 }, labels: {fill: "#01579b"} }}
    data={yearlyExpense.monthTot}
    x={monthStrings['x']}
    domain={{x: [0, 13]}}
    labels={({ datum }) = ` ${datum.y} `}
  />
</VictoryChart>
```

The month values returned from the database are zero-based indices, so we define our own array of month name strings to map to these indices. To render the bar chart, we place

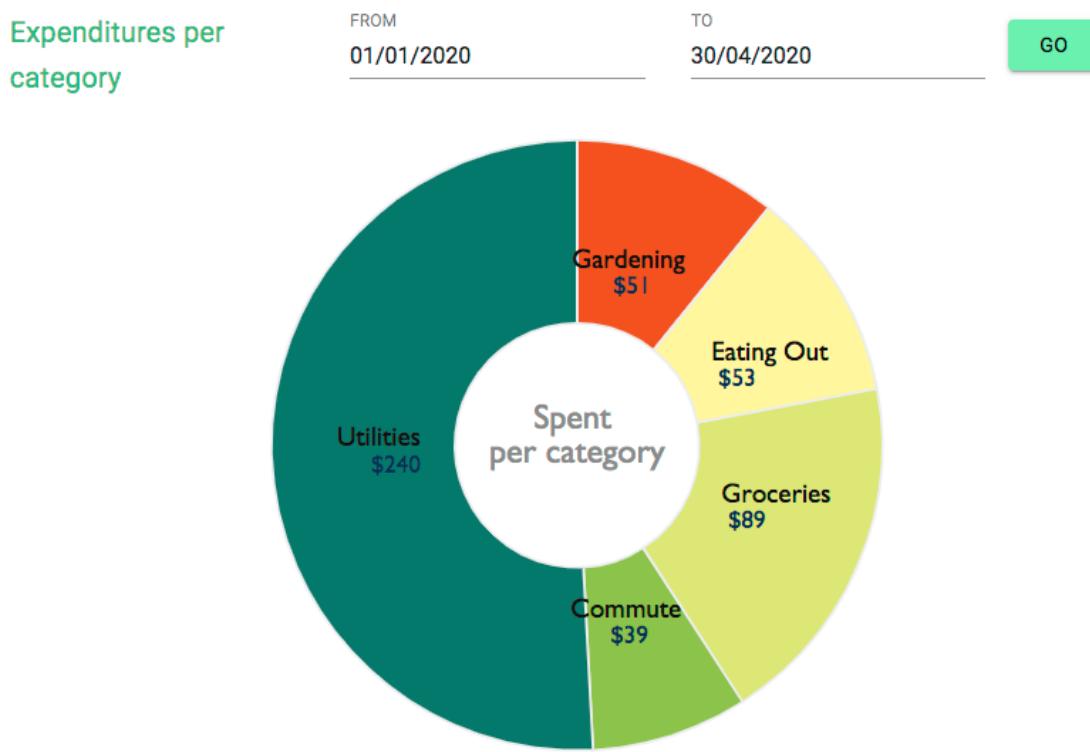
a `VictoryBar` component in a `VictoryChart` component, giving us the flexibility to customize the bar chart wrapper, and also the y axis with a `VictoryAxis` component, which is added without any props so that a y axis is not displayed at all.

We pass the data to `VictoryBar` and also define the categories for the x axis values using the month strings so that all months of the year are displayed on the chart, even if a corresponding total value does not exist yet. We render individual labels for each bar to show the total expense value for each month. To map the x axis value with the correct month string, we specify it in the `x` prop for the `VictoryBar` component.

This code plots and renders the bar chart against the data provided, with the monthly expense totals mapped for each month. In the next section, we will follow similar steps to add a pie chart to graphically display the average expenses per category in a given date range.

Average expenses per category in a pie chart

We can render a pie chart showing how much users spend on average per expense category over a given period of time. This will help users visualize which categories consume more or less of their wealth over time. The following screenshot shows how the pie chart will render with user expense data:



We populate the pie chart with each category and its average expenditure value, showing the corresponding name and amount as labels. In the following sections, we will implement this feature by first adding a backend API that will return the average expenses per category over the given date range and in the format needed to render it in a Victory Pie chart. Then, we will add a React component

that will retrieve this data from the backend and render it in the Victory Pie chart.

The average expenses by category API

We will add an API to the backend that will return the average expenses incurred in each category over a given time period and in the data format needed to render the pie chart in the frontend. To implement this API, we will first declare a `GET` route, as shown in the following code.

`mern-expense-tracker/server/routes/expense.routes.js`:

```
| router.route('/api/expenses/category/averages')
|   .get(authCtrl.requireSignin, expenseCtrl.averageCategories)
```

A `GET` request to this route at `'/api/expenses/category/averages'` will first ensure that the requesting client is a signed-in user, and then it will invoke the `averageCategories` controller method. The request will also take the values of the given date range in URL query parameters, which will be used in the `averageCategories` method to determine the dates of the first day and the last day of the provided range. We will need these dates to specify the range for finding the matching expenses that were incurred in the specified date range and recorded by the authenticated user while aggregating the expense averages per category into the data format needed for the chart. The `averageCategories` method is defined in the following code.

`mern-expense-tracker/server/controllers/expense.controller.js`:

```
const averageCategories = async (req, res) => {
  const firstDay = new Date(req.query.firstDay)
  const lastDay = new Date(req.query.lastDay)

  try {
    let categoryMonthlyAvg = await Expense.aggregate([
      { $match : { incurred_on : { $gte : firstDay, $lte: lastDay },
                  recorded_by: mongoose.Types.ObjectId(req.auth._id) } },
      { $group : { _id : {category: "$category"}, totalSpent: {$sum: "$amount"} } },
      { $project: {category: 1, totalSpent: 1}}
```

```

        { $group: { _id: "$_id.category", avgSpent:
            { $avg: "$totalSpent"}}, 
        { $project: {x: '$_id', y: '$avgSpent'} }
    ]).exec()
res.json({monthAVG:categoryMonthlyAvg})
} catch (err) {
    console.log(err)
    return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
    })
}
}
}

```

We run an aggregation operation that finds the matching expenses, groups the expenses by category to first calculate the total and then the average, and returns an output containing the values in the format needed for the `x` and `y` values of the pie chart. The final result of the aggregation contains an array of objects, with each object containing an `x` attribute and a `y` attribute. The `x` attribute contains the category name as the value. The `y` attribute contains the corresponding average expense amount for that category. This final output array generated from the aggregation is sent back in the response to the requesting client.

We can use this API in the frontend with a `fetch` request. You can define a corresponding `fetch` method to make the request, similar to other API implementations. Then, the `fetch` method can be used in a React component to retrieve and render this array of `x` and `y` values in a pie chart. In the next section, we will discuss the implementation of this view to render a pie chart showing the average expenses incurred per category over a given date range.

The CategoryPie component

We will implement a React component that calls the average expenses by category API to render the received array of average expenses incurred per category in a Victory Pie chart.

The API can be fetched either in a `useEffect` hook or when a button is clicked on. In the MERN Expense Tracker application, we render this pie chart in a React component called `CategoryPie`. When this component loads, we render a pie chart for the average expenses incurred per category in the given month. We also add two `DatePicker` components to allow users to select the desired date range and retrieve data for that range with a button click. In the following code, we retrieve the initial average expense data with a `useEffect` hook when the component loads.

`mern-expense-tracker/client/report/CategoryPie.js`:

```
const [error, setError] = useState('')
const [expenses, setExpenses] = useState([])
const jwt = auth.isAuthenticated()
const date = new Date(), y = date.getFullYear(), m = date.getMonth()
const [firstDay, setFirstDay] = useState(new Date(y, m, 1))
const [lastDay, setLastDay] = useState(new Date(y, m + 1, 0))
useEffect(() => {
  const abortController = new AbortController()
  const signal = abortController.signal
  averageCategories({firstDay: firstDay, lastDay: lastDay},
    {t: jwt.token}, signal).then((data) => {
    if (data.error) {
      setError(data.error)
    } else {
      setExpenses(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

With the data received from the backend and set in state, we can render it in a Victory Pie chart. We can add the following code in the

component view to render a customized pie chart with individual text labels for each slice and a center label for the chart.

mern-expense-tracker/client/report/CategoryPie.js:

```
<div style={{width: 550, margin: 'auto'}}>
  <svg viewBox="0 0 320 320">
    <VictoryPie standalone={false} data=
      {expenses.monthAVG} innerRadius={50}
      theme={VictoryTheme.material}
      labelRadius={({innerRadius}) = innerRadius + 14}
      labelComponent={ VictoryLabel angle={0} style={[{
        fontSize: '11px',
        fill: '#0f0f0f'
      },
      {
        fontSize: '10px',
        fill: '#013157'
      }]} text={( datum) = `${datum.x}\n ${datum.y}`}/>
    <VictoryLabel
      textAnchor="middle"
      style={{fontSize: 14, fill: '#8b8b8b' }}
      x={175} y={170}
      text={`Spent \nper category`}>
    />
  </svg>
</div>
```

To render the pie chart with a separate center label, we place a `VictoryPie` component in an `svg` element, giving us the flexibility to customize the pie chart wrapping and a separate circular label using a `VictoryLabel` outside the pie chart code.

We pass the data to `VictoryPie`, define customized labels for each slice, and make the pie chart standalone so that the center label can be placed over the chart. This code plots and renders the pie chart against the data provided with the average expense displayed for each category.

We have added three different Victory charts to the application based on the user-recorded expense data, which was processed as needed and retrieved from the database in the backend. The MERN Expense Tracker application is complete with abilities that allow users to record their day-to-day expenses, and then visualize data

patterns and expenditure habits extracted from the expense data recorded over time.

Summary

In this chapter, we extended the MERN skeleton application to develop an expense tracking application with data visualization features. We designed an **Expense** model for recording expense details and implemented the full-stack **CRUD (Create, Read, Update, Delete)** functionalities that allowed signed-in users to record their day-to-day expenses, see a list of their expenses, and modify existing expense records.

We added data processing and visualization features that gave users an overview of their current expenses and also an idea of how much more or less they are spending per expense category. We also incorporated different types of charts to show users their expenditure patterns over various time ranges.

While implementing these features, we learned about some of the data processing options with the aggregation framework in MongoDB and also incorporated some of the customizable chart components from Victory. You can explore the aggregation framework and the Victory library further to incorporate more complex data visualization features in your own full-stack applications.

In the next chapter, we will explore even more advanced possibilities with MERN stack technologies as we build a media streaming application by extending the MERN skeleton.

Building a Media Streaming Application

Uploading and streaming media content, specifically video content, has been a growing part of the internet culture for some time now. From individuals sharing personal video content to the entertainment industry disseminating commercial content on online streaming services, we all rely on web applications that enable smooth uploading and streaming. Capabilities within the MERN stack technologies can be used to build and integrate these core streaming features into any MERN-based full-stack application. In this chapter, we will extend the MERN skeleton application to build a media streaming application, while demonstrating how to utilize MongoDB GridFS and add media streaming features to your web applications.

In this chapter, we will cover the following topics to implement basic media uploading and streaming by extending the MERN skeleton application:

- Introducing MERN Mediastream
- Uploading videos to MongoDB GridFS
- Storing and retrieving media details
- Streaming videos from GridFS to a basic media player
- Listing, displaying, updating, and deleting media

Introducing MERN Mediastream

We will build the MERN Mediastream application by extending the skeleton application. This will be a simple video streaming application that allows registered users to upload videos that can be streamed by anyone browsing the application. The following screenshot shows the home page view on the MERN Mediastream application, along with a list of popular videos on the platform:

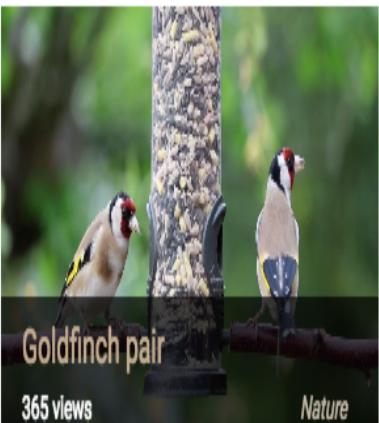
Popular Videos



Endeavour Lift-off

425 views

Space Exploration



Goldfinch pair

365 views

Nature



Ballerina

298 views

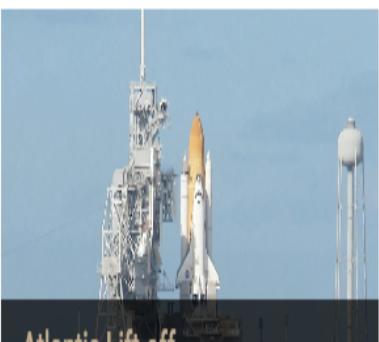
Music



Frozen Branches

270 views

Nature



Atlantis Lift-off

244 views

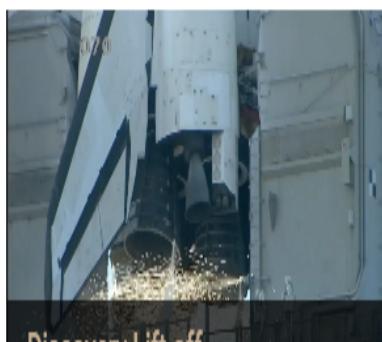
Space Exploration



Carousel Music Box

223 views

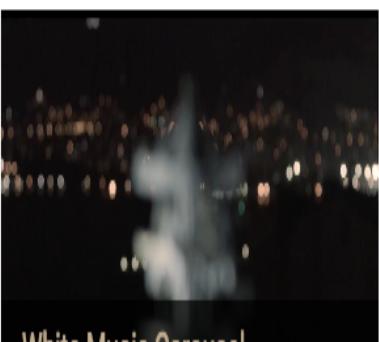
Music



Discovery Lift-off

201 views

Space Exploration



White Music Carousel

182 views



Flock of Ducks

146 views

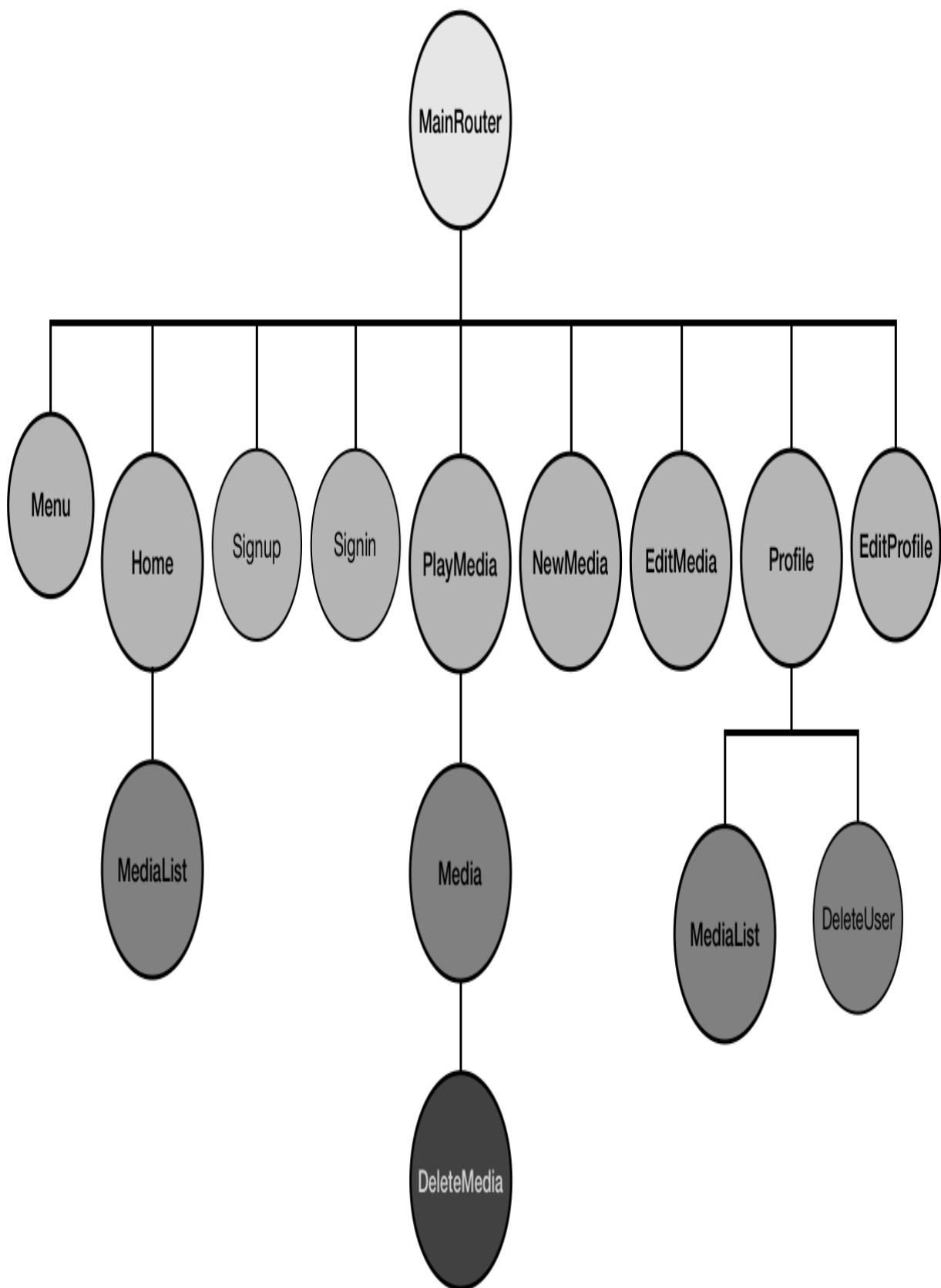
Nature



The code for the complete MERN Mediastream application is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter11%20and%2012/mern-mediastream>.

You can clone this code and run the application as you go through the code explanations in the rest of this chapter.

The frontend user interface views that are needed for the features related to media uploading, editing, and streaming in a simple media player will be developed by extending and modifying existing React components in the MERN skeleton application. The component tree shown in the following diagram shows all the custom React components that make up the MERN Mediastream frontend that will be developed in this chapter:



We will add new React components to implement views for uploading a new video, listing already posted media, modifying media post details, and displaying a video where users can interact with the video content to stream and watch it. We will also modify existing components such as the `Home` component so we can render a list of popular videos and the `Profile` component so we can list all the videos that are posted by a given user. These uploading and streaming capabilities in the application will rely on the user's ability to upload video content. In the next section, we will discuss how to allow signed-in users to add media to the application.

Uploading and storing media

Registered users on the MERN Mediastream application will be able to upload videos from their local files to store each video and related details directly on MongoDB using GridFS. To enable uploading media content to the application, we need to define how to store media details and the video content and implement a full-stack slice that will let users create a new media post and upload a video file. In the following sections, first we will define a media model for storing the details of each media post and configure GridFS to store the associated video content. Then, we will discuss implementations for the backend API, which will receive and store the video content with other media details, and the frontend form view, which will allow a user to create a new media post on the application.

Defining a Media model

We will implement a Mongoose model to define a Media model for storing the details of each piece of media that's posted to the application. This model will be defined in `server/models/media.model.js`, and the implementation will be similar to other Mongoose model implementations we covered in the previous chapters, such as the Course model we defined in [Chapter 6, Building a Web-Based Classroom Application](#). The Media schema in this model will have fields to record the media title, description, genre, number of views, dates of when the media was posted and updated, and a reference to the user who posted the media. The code for defining the media fields is as follows:

- **Media title:** The `title` field is declared to be of the `String` type and will be a required field for introducing the media that are uploaded to the application:

```
|   title: {  
|     type: String,  
|     required: 'title is required'  
|   }
```

- **Media description and genre:** The `description` and `genre` fields will be of type `String`, and these will store additional details about the media posted. The `genre` field will also allow us to group the different media uploaded to the application.

```
|   description: String,  
|   genre: String,
```

- **Number of views:** The `views` field is defined as a `Number` type and will keep track of how many times the uploaded media was viewed by users in the application:

```
|   views: {  
|     type: Number,
```

```
|     default: 0  
| },
```

- **Media posted by:** The `postedBy` field will reference the user who created the media post:

```
|     postedBy: {  
|       type: mongoose.Schema.ObjectId,  
|       ref: 'User'  
| },
```

- **Created and updated at times:** The `created` and `updated` fields will be `Date` types, with `created` generated when a new media is added and `updated` changed when any media details are modified:

```
|     updated: Date,  
|     created: {  
|       type: Date,  
|       default: Date.now  
| },
```

The fields that were added to the schema definition will only store details about each video that's posted to the application. In order to store the video content itself, we will use MongoDB GridFS. Before getting into the implementation of uploading a video file, in the next section we will discuss how GridFS makes it possible to store large files in MongoDB, and then add initialization code to start using GridFS in this streaming application.

Using MongoDB GridFS to store large files

In previous chapters, we discussed how files uploaded by users could be stored directly in MongoDB as binary data; for example, when adding a profile photo in the *Upload profile photo* section in [Chapter 5, Growing the Skeleton into a Social Media Application](#). But this only worked for files smaller than 16 MB. In order to store larger files in MongoDB, such as video files needed for this streaming application, we will need to use GridFS.

GridFS is a specification in MongoDB that allows us to store large files in MongoDB by dividing a given file into several chunks. Each chunk can be a maximum of 255 KB in size, and is stored as a separate document. When the file has to be retrieved in response to a query to GridFS, the chunks are reassembled as needed. This opens up the option to fetch and load only parts of the file as required, rather than retrieving the whole file.

In the case of storing and retrieving video files for the MERN Mediastream application, we will utilize GridFS to store video files and stream parts of the video, depending on which part the user skips to and starts playing from.



You can learn more about the GridFS specification and its features in the official MongoDB documentation at <https://docs.mongodb.com/manual/core/gridfs/>.

To access and work with MongoDB GridFS from our backend code, we will use the Node.js MongoDB driver's streaming API by creating a `GridFSBucket` with the established database connection.



GridFSBucket is the GridFS streaming interface that gives us access to the streaming GridFS API. It can be used to interact with files in GridFS. You can learn more about `GridFSBucket` and the streaming API in the Node.js MongoDB Driver API documentation at <https://mongodb.github.io/node-mongodb-native/3.2/api/GridFSBucket.html>.

Since we are using Mongoose to establish a connection with the MongoDB database for our application, we will add the following code to initialize a new `GridFSBucket` with this database connection after it has been established.

mern-mediastream/server/controllers/media.controller.js:

```
| import mongoose from 'mongoose'  
| let gridfs = null  
| mongoose.connection.on('connected', () => {  
|   gridfs = new mongoose.mongo.GridFSBucket(mongoose.connection.db)  
| })
```

The `gridfs` object we created here will give us access to the GridFS functionalities that are required to store the video file when new media is created and to fetch the file when the media is to be streamed back to the user. In the next section, we will add a create media form view and an API in the backend, which will use this `gridfs` object to save the video file that's uploaded with the request that's sent from the form view in the frontend.

Creating a new media post

For a user to be able to create a new media post on the application, we will need to integrate a full-stack slice that allows the user to fill out a form in the frontend and then save both the provided media details and the associated video file in the database in the backend. To implement this feature, in the following sections, we will add a create media API in the backend, along with a way to fetch this API in the frontend. Then, we will implement a create new media form view that allows the user to input media details and select a video file from their local filesystem.

The create media API

We will implement a create media API in the backend to allow users to create new media posts on the application. This API will receive a POST request at `'/api/media/new/:userId'` with the multipart body content containing the media fields and the uploaded video file. First, we will declare the create media route and utilize the `userByID` method from the user controller, as shown in the following code.

mern-mediastream/server/routes/media.routes.js:

```
| router.route('/api/media/new/:userId')
|   .post(authCtrl.requireSignin, mediaCtrl.create)
| router.param('userId', userCtrl.userByID)
```

The `userByID` method processes the `:userId` parameter that's passed in the URL and retrieves the associated user from the database. The user object becomes available in the request object to be used in the next method that will be executed. Similar to the user and auth routes, we will have to mount the media routes on the Express app in `express.js` as follows.

mern-mediastream/server/express.js:

```
| app.use('/', mediaRoutes)
```

A POST request to the create route URL, `/api/media/new/:userId`, will make sure the user is signed in and then initiate the `create` method in the media controller. The `create` controller method will use the `formidable` node module to parse the multipart request body that will contain the media details and video file uploaded by the user. You can install the module by running the following command from the command line:

```
| yarn add formidable
```

In the `create` method, we will use the media fields that have been received in the form data and parsed with `formidable` to generate a new Media object and then save it to the database. This `create` controller method is defined as follows.

mern-mediastream/server/controllers/media.controller.js:

```
const create = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        error: "Video could not be uploaded"
      })
    }
    let media = new Media(fields)
    media.postedBy = req.profile
    if(files.video){
      let writestream = gridfs.openUploadStream(media._id, {
        contentType: files.video.type || 'binary/octet-stream'})
      fs.createReadStream(files.video.path).pipe(writestream)
    }
    try {
      let result = await media.save()
      res.status(200).json(result)
    }
    catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}
```

If there is a file in the request, `formidable` will store it temporarily in the filesystem. We will use this temporary file and the media object's ID to create a writable stream with `gridfs.openUploadStream`. Here, the temporary file will be read and then written into MongoDB GridFS, while setting the `filename` value to the media ID. This will generate the associated chunks and file information documents in MongoDB, and when it is time to retrieve this file, we will identify it with the media ID.

To use this create media API in the frontend, we will add a corresponding `fetch` method in `api-media.js` to make a `POST` request to the

API by passing the multipart form data from the view. This method will be defined as follows.

mern-mediastream/client/media/api-media.js:

```
const create = async (params, credentials, media) => {
  try {
    let response = await fetch('/api/media/new/' + params.userId, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      },
      body: media
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

This `create` fetch method will take the current user's ID, user credentials, and the media form data to make a POST request to the create media API in the backend. We will use this method when the user submits the new media form to upload a new video and post it on the application. In the next section, we will look at the implementation of this form view in the frontend.

The NewMedia component

Registered users on the MERN Mediastream application will interact with a form view to enter details of a new media post. This form view will be rendered in the `NewMedia` component, which will allow a signed-in user to create a media post by entering the title, description, and genre of the video and uploading a video file from their local filesystem.

This form view will render as follows:

New Video

UPLOAD ↑

Title

Description

Genre

SUBMIT

We will implement this form in a React component named `NewMedia`. For the view, we will add the file upload elements using a Material-UI Button and an HTML5 file `input` element, as shown in the following code.

`mern-mediastream/client/media/NewMedia.js`:

```
| <input accept="video/*"  
|   onChange={handleChange('video')}
```

```

        id="icon-button-file"
        type="file"
        style={{display: none}}/>
<label htmlFor="icon-button-file">
    <Button color="secondary" variant="contained" component="span">
        Upload <FileUpload/>
    </Button>
</label>
<span>{values.video ? values.video.name : ''}</span>

```

In the file `input` element, we specify that it accepts video files, so when the user clicks on Upload and browses through their local folders, they only have the option to upload a video file.

Then, in the view, we add the title, description, and genre form fields with the `TextField` components, as shown in the following code.

`mern-mediastream/client/media/NewMedia.js`:

```

<TextField id="title" label="Title" value={values.title}
          onChange={handleChange('title')} margin="normal"/><br/>
<TextField id="multiline-flexible" label="Description"
          multiline rows="2"
          value={values.description}
          onChange={handleChange('description')}/><br/>
<TextField id="genre" label="Genre" value={values.genre}
          onChange={handleChange('genre')}/><br/>

```

These form field changes will be tracked with the `handleChange` method when a user interacts with the input fields to enter values. The `handleChange` function will be defined as follows.

`mern-mediastream/client/media/NewMedia.js`:

```

const handleChange = name => event => {
  const value = name === 'video'
    ? event.target.files[0]
    : event.target.value
  setValues({ ...values, [name]: value })
}

```

The `handleChange` method updates the state with the new values, including the name of the video file, if one is uploaded by the user.

Finally, you can complete this form view by adding a Submit button, which, when clicked, should send the form data to the server. We will define a `clickSubmit` method here, which will be called when the Submit button is clicked by the user.

mern-mediastream/client/media/NewMedia.js:

```
const clickSubmit = () => {
  let mediaData = new FormData()
  values.title && mediaData.append('title', values.title)
  values.video && mediaData.append('video', values.video)
  values.description && mediaData.append('description',
    values.description)
  values.genre && mediaData.append('genre', values.genre)
  create({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, mediaData).then((data) => {
    if (data.error) {
      setValues({...values, error: data.error})
    } else {
      setValues({...values, error: '', mediaId: data._id,
        redirect: true})
    }
  })
}
```

This `clickSubmit` function will take the input values and populate `mediaData`, which is a `FormData` object that ensures the data is stored in the correct format for the `multipart/form-data` encoding type. Then, the `create` fetch method is called to create the new media in the backend with this form data. On successful media creation, the user may be redirected to a different view as desired, for example, to a Media view with the new media details, as shown in the following code.

mern-mediastream/client/media/NewMedia.js:

```
if (values.redirect) {
  return (<Redirect to={'/media/' + values.mediaId}/>)
}
```

The `NewMedia` component can only be viewed by a signed-in user. Therefore, we will add a `PrivateRoute` in the `MainRouter` component, which

will render this form only for authenticated users at `/media/new`.

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

```
| <PrivateRoute path="/media/new" component={NewMedia} />
```

This link can be added to any view, such as in the `Menu` component, so that it's rendered conditionally when users are signed in. Now that it is possible to add new media posts in this media streaming application, in the next section we will discuss the implementation of retrieving and rendering the video content associated with each media post. This will allow users to stream and view video files stored in MongoDB GridFS from the frontend of the application.

Retrieving and streaming media

Any visitor browsing through the MERN Mediastream application will be able to view the media posted on the application by its users. Implementing this feature will require streaming the video files stored in MongoDB GridFS to the requesting client and rendering the stream in a media player. In the following sections, we will set up a backend API to retrieve a single video file, which we will then use as a source in a React-based media player to render the streaming video.

The video API

To retrieve the video file associated with a single media post, we will implement a get video API that will accept a GET request at `'/api/medias/video/:mediaId'` and query both the Media collection and GridFS files. We will start implementing this video API by declaring the route shown in the following code, along with a way to handle the `:mediaId` parameter in the URL.

mern-mediastream/server/routes/media.routes.js:

```
| router.route('/api/medias/video/:mediaId')
|   .get(mediaCtrl.video)
| router.param('mediaId', mediaCtrl.mediaByID)
```

The `:mediaId` parameter in the route URL will be processed in the `mediaByID` controller to fetch the associated document from the Media collection and file details from GridFS. These retrieved results are then attached to the request object so that it can be used in the `video` controller method as required. This `mediaByID` controller method is defined as follows.

mern-mediastream/server/controllers/media.controller.js:

```
const mediaByID = async (req, res, next, id) => {
  try{
    let media = await Media.findById(id).populate('postedBy',
      '_id name').exec()
    if (!media)
      return res.status('400').json({
        error: "Media not found"
      })
    req.media = media
    let files = await gridfs.find({filename:media._id}).toArray()
    if (!files[0]) {
      return res.status(404).send({
        error: 'No video found'
      })
    }
    req.file = files[0]
    next()
  }catch(err) {
```

```

        return res.status(404).send({
          error: 'Could not retrieve media file'
        })
      }
    }
  }
}

```

To retrieve the relevant file details from GridFS, we use `find` from the MongoDB streaming API. We query the files stored in GridFS by the filename value, which should match the corresponding media ID in the Media collection. Then, we receive the resulting matching file records in an array and attach the first result to the request object so that it can be used in the next method.

The next method that's invoked when this API receives a request is the `video` controller method. In this method, depending on whether the request contains range headers, we send back the correct chunks of video with the related content information set as response headers. The `video` controller method is defined with the following structure, with the response composed depending on the existence of range headers in the request.

mern-mediastream/server/controllers/media.controller.js:

```

const video = (req, res) => {
  const range = req.headers["range"]
  if (range && typeof range === "string") {
    ...
    ... consider range headers and send only relevant chunks in
    response ...
    ...
  } else {
    res.header('Content-Length', req.file.length)
    res.header('Content-Type', req.file.contentType)

    let downloadStream = gridfs.openDownloadStream(req.file._id)
    downloadStream.pipe(res)
    downloadStream.on('error', () => {
      res.sendStatus(404)
    })
    downloadStream.on('end', () => {
      res.end()
    })
  }
}

```

In the preceding code, if the request does not contain range headers, we stream back the whole video file using `gridfs.openDownloadStream`, which gives us a readable stream of the corresponding file stored in GridFS. This is piped with the response sent back to the client. In the response header, we set the content type and total length of the file.

If the request contains range headers – for example, when the user drags to the middle of the video and starts playing from that point – we need to convert the received range headers to the start and end positions, which will correspond with the correct chunks stored in GridFS, as shown in the following code.

mern-mediastream/server/controllers/media.controller.js:

```
const parts = range.replace(/bytes=/, "").split("-")
const partialstart = parts[0]
const partialend = parts[1]

const start = parseInt(partialstart, 10)
const end = partialend ? parseInt(partialend, 10) : req.file.length -
1
const chunksize = (end - start) + 1

res.writeHead(206, {
  'Accept-Ranges': 'bytes',
  'Content-Length': chunksize,
  'Content-Range': `bytes ${start} - ${end} / ${req.file.length}`,
  'Content-Type': req.file.contentType
})

let downloadStream = gridfs.openDownloadStream(req.file._id, {start,
end: end+1})
downloadStream.pipe(res)
downloadStream.on('error', () => {
  res.sendStatus(404)
})
downloadStream.on('end', () => {
  res.end()
})
```

We pass the start and end values that have been extracted from the header as a range to `gridfs.openDownloadStream`. These start and end values specify the 0-based offset in bytes to start streaming from and stop streaming before. We also set the response headers with

additional file details, including content length, range, and type. The content length will now be the total size of the content within the defined range. Therefore, the readable stream that's piped back to the response, in this case, will only contain the chunks of file data that fall within the start and end ranges.

The final readable stream that's piped to the response after a request is received at this get video API can be rendered directly in a basic HTML5 media player or a React-flavored media player in the frontend view. In the next section, we will look at how to render this video stream in a simple React media player.

Using a React media player to render the video

In the frontend of the application, we can render the video file being streamed from MongoDB GridFS in a media player. A good option for a React-flavored media player is the `ReactPlayer` component, available as a node module, which can be customized as required. Providing the video stream as a source to a default `ReactPlayer` component will render with basic player controls, as shown in the following screenshot:



To start using `ReactPlayer` in our frontend code, we need to install the corresponding node module by running the following Yarn command from the command line:

```
| yarn add react-player
```

Once installed, we can import it into any React component and add it to the view. For basic usage with the default controls provided by the browser, we can add it to any React view in any application that has access to the ID of the media to be rendered, as shown in the following code:

```
| <ReactPlayer url={'/api/media/video/' + media._id} controls/>
```

This will load the player with the video stream that was received from the get video API and provide the user with basic control options to interact with the stream being played. `ReactPlayer` can be customized so that more options are available. We will explore some of these advanced options for customizing this `ReactPlayer` with our own controls in the next chapter.



To learn more about what is possible with `ReactPlayer`, visit cookpete.com/react-player.

Now, it's possible to retrieve a single video file stored in MongoDB GridFS and stream it to a media player in the frontend for the user to view and play the video as desired. In the next section, we will discuss how to fetch and display lists of multiple videos from the backend to the frontend of the streaming application.

Listing media

In MERN Mediastream, we will add list views of relevant media with a snapshot of each video to give visitors easier access and an overview of the videos on the application. For example, in the following screenshot, the `Profile` component displays a list of media posted by the corresponding user, showing the video preview and other details of each media:

Profile

Violet Bernstein
violet@silent.info

Joined: Fri Dec 22 2017

Carousel Music Box 223 views Music	White Music Carousel 182 views Music	Ballerina 298 views Music
--	--	---------------------------------

We will set up list APIs in the backend to retrieve different lists, such as videos uploaded by a single user and the most popular videos with the highest views in the application. Then, these retrieved lists can be rendered in a reusable `MediaList` component, which will receive a list of media objects as a prop from a parent component that fetches the specific API. In the following sections, we will implement the `MediaList` component and the backend APIs to retrieve the two different lists of media from the database.

The MediaList component

The `MediaList` component is a reusable component that will take a list of media and iterate through it to render each media item in the view. In MERN Mediastream, we use it to render a list of the most popular media in the home view and a list of media uploaded by a specific user in their profile.

In the view part of the `MediaList` component, we will iterate through the `media` array that's received in `props` using `map`, as shown in the following code.

`mern-mediastream/client/media/MediaList.js`:

```
<GridList cols={3}>
  {props.media.map((tile, i) => (
    <GridListTile key={i}>
      <Link to={"/media/" + tile._id}>
        <ReactPlayer url={'/api/media/video/' + tile._id}
          width='100%' height='inherit' style=
            {{maxHeight: '100%'}}/>
      </Link>
      <GridListTileBar title={<> <Link
        to={"/media/" + tile._id}> {tile.title} </Link>
        subtitle={<> <span>
          <span>{tile.views} views</span>
          <span className={classes.tileGenre}>
            <em>{tile.genre}</em>
          </span>
        </span>
      </>
    </GridListTileBar>
  )))
</GridList>
```

This `MediaList` component uses the Material-UI `GridList` components as it iterates through the array of objects sent in the `props` and renders media details for each item in the list. It also includes a `ReactPlayer` component, which renders the video URL without showing any controls. In the view, this gives the visitor a brief overview of each piece of media, as well as a glimpse of the video content.

This component can be added to any view that can provide an array of media objects. In the MERN Mediastream application, we use it to render two different lists of media: a list of popular media and a list of media posted by a specific user. In the next section, we will look at how to retrieve a list of popular media from the database to render it in the frontend.

Listing popular media

To retrieve specific lists of media from the database, we need to set up the relevant APIs on the server. For popular media, we will set up a route that receives a GET request at `/api/media/popular`. The route will be declared as follows.

`mern-mediastream/server/routes/media.routes.js`:

```
| router.route('/api/media/popular')
|   .get(mediaCtrl.listPopular)
```

A GET request to this URL will invoke the `listPopular` method. The `listPopular` controller method will query the Media collection and retrieve nine media documents that have the highest `views` in the whole collection. The `listPopular` method is defined as follows.

`mern-mediastream/server/controllers/media.controller.js`:

```
const listPopular = async (req, res) => {
  try{
    let media = await Media.find({})
      .populate('postedBy', '_id name')
      .sort('-views')
      .limit(9)
      .exec()
    res.json(media)
  } catch(err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The result that's returned by the query to the Media collection is sorted by the number of views in descending order and limited to nine. Each media document in this list will also contain the name and ID of the user who posted it since we are calling `populate` to add these user attributes.

This API can be used in the frontend with a fetch request. You can define a corresponding fetch method in `api-media.js` to make the request, similarly to other API implementations. Then, the fetch method can be called in a React component, such as in the `Home` component for this application. In the `Home` component, we will fetch a list of popular videos in a `useEffect` hook, as shown in the following code.

mern-mediastream/client/core/Home.js:

```
useEffect(() => {
  const abortController = new AbortController()
  const signal = abortController.signal
  listPopular(signal).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      setMedia(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

The list that's fetched from the API in this hook is set in the state so that it can be passed to a `MediaList` component in the view. In the `Home` view, we can add the `MediaList` as follows, with the list provided as a prop.

mern-mediastream/client/core/Home.js:

```
| <MediaList media={media}/>
```

This will render a list of up to nine of the most popular videos from the database on the home page of the MERN Mediastream application. In the next section, we will discuss a similar implementation to retrieve and render a list of media that's been posted by a specific user.

Listing media by users

To be able to retrieve a list of media that's been uploaded by a specific user from the database, we will set up an API with a route that accepts a `GET` request at `'/api/media/by/:userId'`. The route will be declared as follows.

`mern-mediastream/server/routes/media.routes.js`:

```
| router.route('/api/media/by/:userId')
|   .get(mediaCtrl.listByUser)
```

A `GET` request to this route will invoke the `listByUser` method. The `listByUser` controller method will query the Media collection to find media documents that have `postedBy` values matching with the `userId` attached as a parameter in the URL. The `listByUser` controller method is defined as follows.

`mern-mediastream/server/controllers/media.controller.js`:

```
const listByUser = async (req, res) => {
  try{
    let media = await Media.find({postedBy: req.profile._id})
      .populate('postedBy', '_id name')
      .sort('-created')
      .exec()
    res.json(media)
  } catch(err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The result that's returned from the query to the Media collection is sorted by the date it was created on, with the latest post showing up first. Each media document in this list will also contain the name and ID of the user who posted it since we are calling `populate` to add these user attributes.

This API can be used in the frontend with a fetch request. You can define a corresponding `fetch` method in `api-media.js` to make the request, similar to other API implementations. Then, the `fetch` method can be called in a React component. In our application, we use the `fetch` method in the `Profile` component, similar to the `listPopular` fetch method we used in the home view, to retrieve the list data, set it to the state, and then pass it to a `MediaList` component. This will render a profile page with a list of media that was posted by the corresponding user.

We are able to retrieve and display multiple videos on the application by utilizing APIs that have been implemented in the backend to fetch the list data. We can also utilize a `ReactPlayer` component without controls to give the user a glimpse of each video when we render the list in the frontend views. In the next section, we will discuss the full-stack slices that will display media posts and allow authorized users to update and delete individual media posts in the application.

Displaying, updating, and deleting media

Any visitor to MERN Mediastream will be able to view media details and stream videos, while only registered users will be able to edit the media's details and delete it any time after they post it on the application. In the following sections, we will implement full-stack slices, including backend APIs and frontend views, to display a single media post, update details of a media post, and delete a media post from the application.

Displaying media

Any visitor to MERN Mediastream will be able to browse to a single media view to play a video and read the details associated with it. Every time a specific video is loaded on the application, we will also increment the number of views associated with the media. In the following sections, we will implement the individual media view by adding a read media API to the backend, a way to call this API from the frontend, and the React component that will display the media details in the view.

The read media API

To implement the read media API in the backend, we will start by adding a `GET` route that queries the `Media` collection with an ID and returns the media document in the response. The route is declared as follows.

`mern-mediastream/server/routes/media.routes.js`:

```
| router.route('/api/media/:mediaId')
|   .get( mediaCtrl.incrementViews, mediaCtrl.read)
```

The `mediaId` in the request URL will cause the `mediaByID` controller method to execute and attach the retrieved media document to the request object so that it can be accessed in the next method.

A `GET` request to this API will execute the `incrementViews` controller method next, which will find the matching media record and increment the `views` value by `1`, before saving the updated record to the database. The `incrementViews` method is defined as follows.

`mern-mediastream/server/controllers/media.controller.js`:

```
| const incrementViews = async (req, res, next) => {
|   try {
|     await Media.findByIdAndUpdate(req.media._id,
|       {$inc: {"views": 1}}, {new: true}).exec()
|     next()
|   } catch(err){
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

This method will increment the number of views for a given media by `1` every time this read media API is called. After the media is updated from this `incrementViews` method, the `read` controller method is invoked. The `read` controller method will simply return the retrieved

media document in response to the requesting client, as shown in the following code.

mern-mediastream/server/controllers/media.controller.js:

```
| const read = (req, res) => {
|   return res.json(req.media)
| }
```

To retrieve the media document that was sent in the response, we need to call this read media API in the frontend using a `fetch` method. We will set up a corresponding fetch method in `api-media.js`, as shown in the following code.

mern-mediastream/client/media/api-media.js:

```
| const read = async (params, signal) => {
|   try {
|     let response = await fetch('/api/media/' + params.mediaId, {
|       method: 'GET',
|       signal: signal
|     })
|     return await response.json()
|   } catch(err) {
|     console.log(err)
|   }
| }
```

This method takes the ID of the media to be retrieved and makes a `GET` request to the read API route using a `fetch`.

The read media API can be used to render individual media details in a view or to pre-populate a media edit form. In the next section, we will use this `fetch` method to call the read media API in the React component to render the media details, as well as a `ReactPlayer` that will play the associated video.

The Media component

The `Media` component will render details of an individual media record and stream the video in a basic `ReactPlayer` with default browser controls. The completed single Media view will look as follows:

Goldfinch pair 389 views

Nature



A photograph showing two goldfinches perched on a bird feeder. The birds have bright red faces and yellow wing patches. They are eating seeds from a cylindrical bird feeder hanging from a branch.

▶ 0:07 / 0:17

Thomas Brogan
Published on Tue Feb 20 2018

Edit trash

The goldfinch is a highly colored finch with a bright red face and yellow wing patch. Sociable, often breeding in loose colonies, they have a delightful liquid twittering song and call. Their long fine beaks allow them to extract otherwise inaccessible seeds from thistles and teasels. Increasingly they are visiting bird tables and feeders, as seen in this video eating seeds from the feeder.

The `Media` component can call the read API to fetch the media data itself or receive the data as props from a parent component that makes the call to the read API. In the latter case, the parent component will fetch the media from the server in a `useEffect` hook, set it to state, and add it to the `Media` component, as follows.

mern-mediastream/client/media/PlayMedia.js:

```
| <Media media={media}>
```

In MERN Mediastream, we will add the `Media` component in a `PlayMedia` component that fetches the media content from the server in a `useEffect` hook using the read API and passes it to `Media` as a prop. The composition of the `PlayMedia` component will be discussed in more detail in the next chapter.

The `Media` component will take this data in the props and render it in the view to display the details and load the video in a `ReactPlayer` component. The title, genre, and view count details of the media can be rendered in a Material-UI `CardHeader` component in the `Media` component, as shown in the following code.

mern-mediastream/client/media/Media.js:

```
| <CardHeader  
|   title={props.media.title}  
|   action={<span>  
|     {props.media.views + ' views'}  
|   </span>}  
|   subheader={props.media.genre}  
| />
```

Besides rendering these media details, we will also load the video in the `Media` component. The video URL, which is basically the get video API route we set up in the backend, is loaded in a `ReactPlayer` with default browser controls, as shown in the following code.

mern-mediastream/client/media/Media.js:

```
| const mediaUrl = props.media._id  
|   ? `/api/media/video/${props.media._id}`  
|   : null
```

```

<ReactPlayer ...
  url={mediaUrl}
  controls
  width={'inherit'}
  height={'inherit'}
  style={{maxHeight: '500px'}}
  config={{ attributes:
    { style: { height: '100%', width: '100%' } }
  }}/>

```

This will render a simple player that allows the user to play the video stream.

The `Media` component also renders additional details about the user who posted the video, a description of the video, and the date it was created, as shown in the following code.

mern-mediastream/client/media/Media.js:

```

<ListItem>
  <ListItemIcon>
    <ListItemIconAvatar>
      <Avatar>
        {props.media.postedBy.name &&
         props.media.postedBy.name[0]}
      </Avatar>
    </ListItemIconAvatar>
    <ListItemText primary={props.media.postedBy.name}>
      secondary={"Published on " +
                  (new Date(props.media.created)) +
                  .toDateString()}/>
  </ListItemText>
</ListItemIcon>
<ListItemIcon>
  <ListItemText primary={props.media.description}/>
</ListItemIcon>

```

In the details being displayed in the Material-UI `ListItemText` component, we will also conditionally show edit and delete options if the currently signed-in user is the one who posted the media being displayed. To render these elements conditionally in the view, we will add the following code after the `ListItemText` displaying the date.

mern-mediastream/client/media/Media.js:

```

{ (auth.isAuthenticated().user && auth.isAuthenticated().user._id)
  == props.media.postedBy._id && (<ListItemSecondaryAction>
    <Link to={"/media/edit/" + props.media._id}>
      <IconButton aria-label="Edit" color="secondary">
        <Edit/>
    </IconButton>
  </ListItemSecondaryAction>)}

```

```
    </IconButton>
  </Link>
  <DeleteMedia mediaId={props.media._id} mediaTitle=
{props.media.title}/>
</ListItemSecondaryAction>)
```

This will ensure that the edit and delete options only render when the current user is signed in and is the uploader of the media being displayed. The edit option links to the media edit form, while the delete option opens a dialog box that can initiate the deletion of this particular media document from the database. In the next section, we will implement the functionality of this option to edit details of the uploaded media post.

Updating media details

Registered users will have access to an edit form for each of their media uploads. Updating and submitting this form will save the changes to the given document in the Media collection. To implement this capability, we will have to create a backend API that allows the update operation on a given media after ensuring that the requesting user is authenticated and authorized. Then, this updated API needs to be called from the frontend with the changed details of the media. In the following sections, we will build this backend API and the React component to allow users to make changes to the media they already posted on the application.

The media update API

In the backend, we will need an API that allows us to update existing media in the database if the user making the request is the authorized creator of the given media post. First, we will declare the PUT route, which accepts the update request from the client.

mern-mediastream/server/routes/media.routes.js:

```
router.route('/api/media/:mediaId')
    .put(authCtrl.requireSignin,
        mediaCtrl.isPoster,
        mediaCtrl.update)
```

When a PUT request is received at `'api/media/:mediaId'`, the server will ensure the signed-in user is the original poster of the media content by calling the `isPoster` controller method. The `isPoster` controller method is defined as follows.

mern-mediastream/server/controllers/media.controller.js:

```
const isPoster = (req, res, next) => {
  let isPoster = req.media && req.auth
    && req.media.postedBy._id == req.auth._id
  if(!isPoster){
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}
```

This method ensures the ID of the authenticated user is the same as the user ID referenced in the `postedBy` field of the given media document. If the user is authorized, the `update` controller method will be called `next` in order to update the existing media document with the changes. The `update` controller method is defined as follows.

mern-mediastream/server/controllers/media.controller.js:

```

const update = async (req, res) => {
  try {
    let media = req.media
    media = extend(media, req.body)
    media.updated = Date.now()
    await media.save()
    res.json(media)
  } catch(err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

```

This method extends the existing media document with the changed details that were received in the request body and saves the updated media to the database.

To access the update API in the frontend, we will add a corresponding fetch method in `api-media.js` that takes the necessary user auth credentials and media details as parameters before making the fetch call to this update media API, as shown in the following code.

`mern-mediastream/client/user/api-media.js`:

```

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

```

This fetch method will be used in the media edit form when the user makes updates and submits the form. In the next section, we will discuss the implementation of this media edit form.

The media edit form

The media edit form, which will allow an authorized user to make changes to the details of a media post, will be similar to the new media form. However, it will not have an upload option, and the fields will be pre-populated with the existing values, as shown in the following screenshot:

Edit Video Details

Title
Endeavour Lift-off

Description
Space Shuttle Endeavour s a retired orbiter from NASA's Space Shuttle

Genre
Space Exploration

SUBMIT

The `EditMedia` component containing this form will fetch the existing values of the media by calling the read media API in a `useEffect` hook, as shown in the following code.

`mern-mediastream/client/media/EditMedia.js`:

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

  read({mediaId: match.params.mediaId}).then((data) => {
    if (data.error) {
      setError(data.error)
    } else {
      setMedia(data)
    }
  })
  return function cleanup() {
```

```
    abortController.abort()
  },
  [match.params.mediaId])
```

The retrieved media details are set to state so that the values can be rendered in the text fields. The form field elements will be the same as in the `NewMedia` component. When a user updates any of the values in the form, the changes will be registered in the `media` object in state with a call to the `handleChange` method. The `handleChange` method is defined as follows.

mediastream/client/media/EditMedia.js:

```
const handleChange = name => event => {
  let updatedMedia = {...media}
  updatedMedia[name] = event.target.value
  setMedia(updatedMedia)
}
```

In this method, the specific field that's being updated in the form is reflected in the corresponding attribute in the `media` object in state. When the user is done editing and clicks submit, a call will be made to the update API with the required credentials and the changed media values. This is done by invoking the `clickSubmit` method, which is defined as follows.

mediastream/client/media/EditMedia.js:

```
const clickSubmit = () => {
  const jwt = auth.isAuthenticated()
  update({
    mediaId: media._id
  }, {
    t: jwt.token
  }, media).then((data) => {
    if (data.error) {
      setError(data.error)
    } else {
      setRedirect(true)
    }
  })
}
```

The call to the update media API will update the media details in the corresponding media document in the Media collection, while the

video file associated with the media remains as it is in the database.

This `EditMedia` component can only be accessed by signed-in users and will be rendered at `'/media/edit/:mediaId'`. Due to this, we will add a `PrivateRoute` in the `MainRouter` component, like so.

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

```
|<PrivateRoute path="/media/edit/:mediaId" component={EditMedia}/>
```

This link is added with an edit icon in the `Media` component, allowing the user who posted the media to access the edit page. In the `Media` view, the user can also choose to delete their media post. We will implement this in the next section.

Deleting media

An authorized user can completely delete the media they uploaded to the application, including the media document in the Media collection and the file chunks stored in MongoDB using GridFS. To allow a user to remove the media from the application, in the following sections, we will define a backend API for media deletion from the database and implement a React component that makes use of this API when the user interacts with the frontend to perform this deletion.

The delete media API

To delete media from the database, we will implement a delete media API in the backend, which will accept a DELETE request from a client at `/api/media/:mediaId`. We will add the `DELETE` route for this API as follows, which will allow an authorized user to delete their uploaded media records.

mern-mediastream/server/routes/media.routes.js:

```
router.route('/api/media/:mediaId')
    .delete(authCtrl.requireSignin,
            mediaCtrl.isPoster,
            mediaCtrl.remove)
```

When the server receives a DELETE request at `'/api/media/:mediaId'`, it will make sure the signed-in user is the original poster of the media by invoking the `isPoster` controller method. Then, the `remove` controller method will completely delete the specified media from the database. The `remove` method is defined as follows.

mern-mediastream/server/controllers/media.controller.js:

```
const remove = async (req, res) => {
  try {
    let media = req.media
    let deletedMedia = await media.remove()
    gridfs.delete(req.file._id)
    res.json(deletedMedia)
  } catch(err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

Besides deleting the media record from the Media collection, we are also using `gridfs` to remove the associated file details and chunks stored in the database.

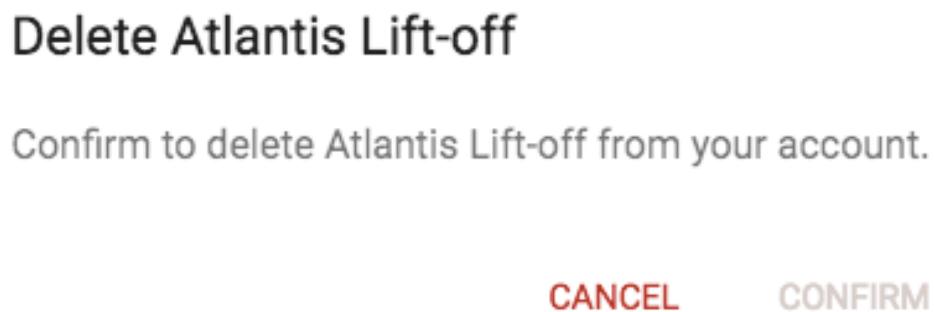
To access this backend API in the frontend, you will also need a fetch method with this route, similar to other API implementations. The fetch method will need to take the media ID and the current user's auth credentials in order to call the delete media API with these values.

The fetch method will be used when the user performs the delete operation by clicking a button in the frontend interface. In the next section, we will discuss a React component called `DeleteMedia`, where by this delete media action will be performed by the user.

The DeleteMedia component

The `DeleteMedia` component is added to the `Media` component and is only visible to the signed-in user who added this specific media.

This component is basically a button that, when clicked, opens a `Dialog` component asking the user to confirm the delete action, as shown in the following screenshot:



This `DeleteMedia` component takes the media ID and title as props when it is added in the `Media` component. Its implementation will be similar to the `DeleteUser` component we discussed in [Chapter 4, Adding a React Frontend to Complete MERN](#). Once the `DeleteMedia` component has been added, the user will be able to remove the posted media completely from the application by confirming their action.

The MERN Mediastream application that we've developed in this chapter is a complete media streaming application with the capability to upload video files to the database, stream stored videos back to the viewers, support CRUD operations such as media create, update, read, and delete, and support options for listing media by uploader or popularity.

Summary

In this chapter, we developed a media streaming application by extending the MERN skeleton application and leveraging MongoDB GridFS.

Besides adding basic add, update, delete, and listing features for media uploads, we looked into how MERN-based applications can allow users to upload video files, store these files into MongoDB GridFS as chunks, and stream the video back to the viewer partially or fully as required. We also covered using `ReactPlayer` with default browser controls to stream the video file. You can apply these streaming capabilities to any full-stack application that may require storing and retrieving large files from the database.

In the next chapter, we will learn how to customize `ReactPlayer` with our own controls and functionality so that users have more options, such as playing the next video in a list. In addition, we will discuss how to improve the SEO of the media details by implementing server-side rendering with data for the media view.

Customizing the Media Player and Improving SEO

Users visit a media-streaming application mainly to play media and explore other related media. This makes the media player—and the view that renders the related media details—crucial to a streaming application.

In this chapter, we will focus on developing the play media page for the MERN Mediastream application that we started building in the previous chapter, [Chapter 11, Building a Media Streaming Application](#). We will address the following topics to bolster the media-playing functionalities and to help boost the presence of the media content across the web so that it reaches more users:

- Customizing player controls on `ReactPlayer`
- Playing the next video from a list of related videos
- Autoplaying a list of related media
- **Server-side rendering (SSR)** of the `PlayMedia` view with data to improve **search engine optimization (SEO)**

After completing these topics, you will be more adept at designing complex interactions between React components in a frontend user interface, and also at improving SEO across your full-stack React applications.

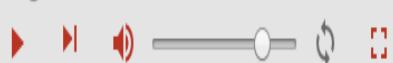
Adding a custom media player to MERN Mediastream

The MERN Mediastream application developed in the previous chapter implemented a simple media player with default browser controls that played one video at a time. In this chapter, we will update the view that plays media with a customized `ReactPlayer` and a related media list that can be set to play automatically when the current video ends. The updated view with the custom player and related playlist will resemble the following screenshot:

Endeavour Final Lift-off

Space Exploration

657 views



Cecil McQueen

Published on Fri Mar 22 2013

Space Shuttle Endeavour is a retired orbiter from NASA's Space Shuttle program and the fifth and final operational shuttle built. Endeavour completed 25 missions, spent 299 days in orbit, and orbited Earth 4,671 times while traveling 122,883,151 miles.

Up Next

 Autoplay ON

Atlantis Landing

Space Exploration

Wed Apr 03 2013

402 views



Atlantis Lift-off

Space Exploration

Thu Jun 21 2012

324 views



Discovery Lift-off

Space Exploration

Tue Nov 05 2002

132 views



Aries Lift-off

Space Exploration

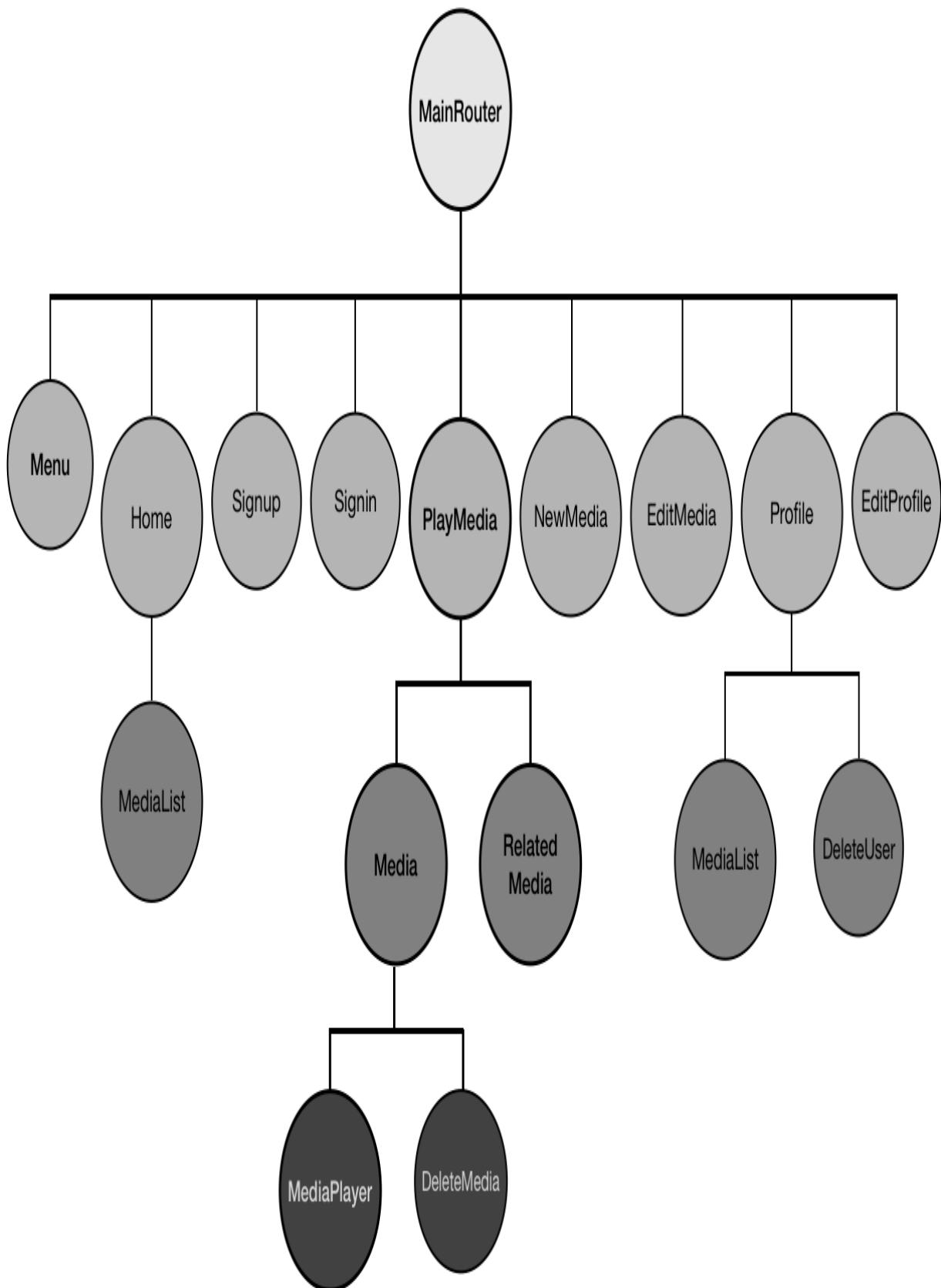
Tue May 02 2000

66 views



The code for the complete MERN Mediastream application is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter11%20and%2012/mern-mediastream>. You can clone this code and run the application as you go through the code explanations for the rest of this chapter.

The following component tree diagram shows all the custom components that make up the MERN Mediastream frontend, including the components that will be improved or added in this chapter:



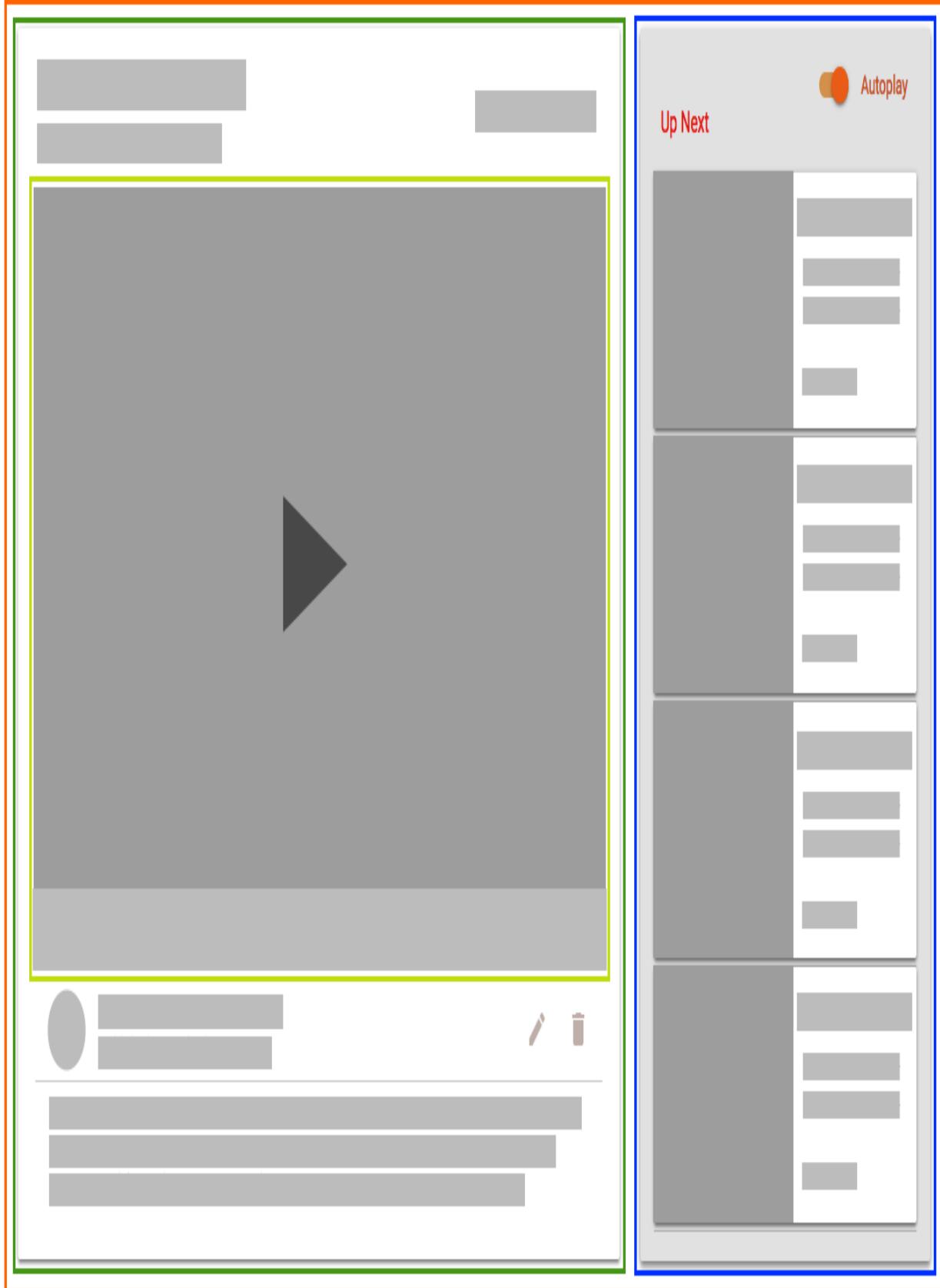
Modified components and new components added in this chapter include the `PlayMedia` component, which houses all the media player functionalities; the `MediaPlayer` component, which adds a `ReactPlayer` with custom controls; and a `RelatedMedia` component, which contains a list of related videos. In the following section, we will discuss the play media page structure, and how it will accommodate all the media viewing and interaction features to be extended in the MERN Mediastream application throughout this chapter.

The play media page

When visitors want to view specific media on MERN Mediastream, they will be taken to the play media page, which will contain the media details, a media player to stream the video, and a list of related media that can be played next. We will implement this `PlayMedia` view in a React component named `PlayMedia`. In the following section, we discuss how this component will be structured to enable these functionalities.

The component structure

We will compose the component structure in the play media page in a way that allows the media data to trickle down to the inner components from the parent component. In this case, the `PlayMedia` component will be the parent component, containing the `RelatedMedia` component and also the `Media` component, which will have a nested `MediaPlayer` component, as sectioned and highlighted in the following screenshot:



When individual media links are accessed in the frontend of the application, the `PlayMedia` component will retrieve and load the corresponding media data and list of related media from the server. Then, the relevant details will be passed as props to the `Media` and `RelatedMedia` child components.

The `RelatedMedia` component will list and link other related media, and clicking any media in this list will re-render the `PlayMedia` component and its inner components with the new data.

We will update the `Media` component we developed in [Chapter 11, Building a Media-Streaming Application](#), to add a customized media player as a child component. This customized `MediaPlayer` component will also utilize the data passed from `PlayMedia` to stream the current video and link to the next video in the related media list.

In the `PlayMedia` component, we will add an autoplay toggle that will let users choose to autoplay the videos in the related media list, one after the other. The autoplay state will be managed from the `PlayMedia` component, but this feature will require the data available in the parent component's state to re-render when a video ends in the `MediaPlayer` nested child component, so it can be ensured that the next video starts playing automatically while keeping track of the related list.

To achieve this, the `PlayMedia` component will need to provide a state updating method as a prop that will be used in the `MediaPlayer` component to update the shared and interdependent state values across these components.

Taking this component structure into consideration, we will extend and update the MERN Mediastream application to implement a functional play media page. In the next section, we will start by adding the feature that provides a list of related media to the user in this `PlayMedia` view.

Listing related media

When a user is viewing an individual media on the application, they will also see a list of related media on the same page. The related media list will consist of other media records that belong to the same genre as the given video and is sorted by the highest number of views. For this feature, we will need to integrate a full-stack slice that retrieves the relevant list from the Media collection in the backend and renders it in the frontend. In the following sections, we will add a related media list API in the backend, along with a way to fetch this API in the frontend, and a React component that renders the list of media retrieved by this API.

The related media list API

We will implement an API endpoint in the backend to retrieve the list of related media from the database. The API will receive a `GET` request at `'/api/media/related/:mediaId'`, and the route will be declared with the other media routes, as follows:

```
mern-mediastream/server/routes/media.routes.js
```

```
| router.route('/api/media/related/:mediaId')
|   .get(mediaCtrl.listRelated)
```

The `:mediaId` parameter in the route path will be processed by the `mediaByID` method implemented in *The video API* section of [Chapter 11, Building a Media Streaming Application](#). It retrieves the media corresponding to this ID from the database and attaches it to the `request` object, so it can be accessed in the next method.

The `listRelated` controller method is the next method invoked for the `GET` request at this API route. This method will query the Media collection to find records with the same genre as the media provided, and also exclude this given media record from the results returned. The `listRelated` controller method is defined as shown in the following code:

```
mern-mediastream/server/controllers/media.controller.js
```

```
const listRelated = async (req, res) => {
  try {
    let media = await Media.find({ "_id": { "$ne": req.media },
      "genre": req.media.genre})
      .limit(4)
      .sort('-views')
      .populate('postedBy', '_id name')
      .exec()
    res.json(media)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

```
| } }
```

The results returned from the query will be sorted by the highest number of views and limited to the top four media records.

Each `media` object in the returned results will also contain the name and ID of the user who posted the media, as specified in the `populate` method.

On the client side, we will set up a corresponding `fetch` method that will be used in the `PlayMedia` component to retrieve the related list of media using this API. This method will be defined as follows:

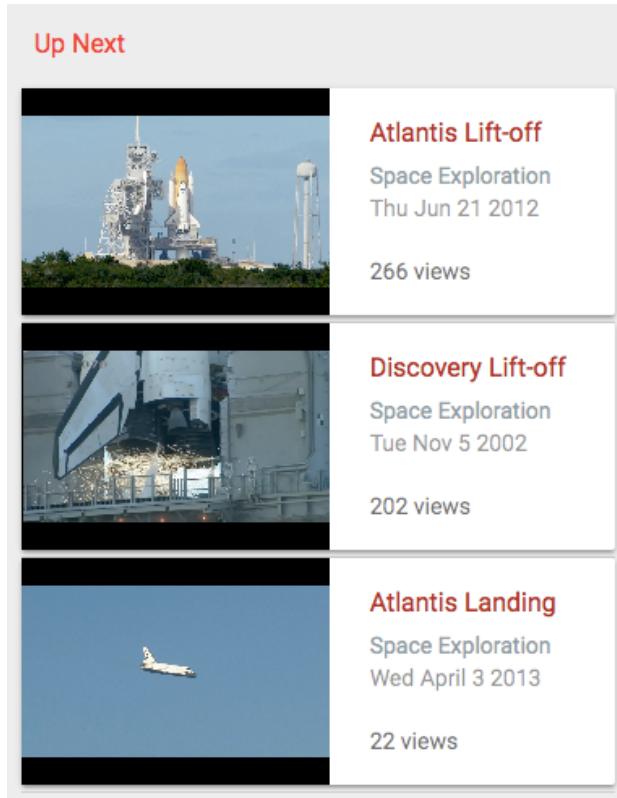
```
mern-mediastream/client/media/api-media.js
```

```
const listRelated = async (params, signal) => {
  try {
    let response = await fetch('/api/media/related/' + params.mediaId, {
      method: 'GET',
      signal: signal,
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      }
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

This `listRelated` fetch method will take a media ID and make a `GET` request to the related media list API in the backend. We will use this method in the `PlayMedia` component to retrieve a list of media related to the current media loaded in the media player. Then this list will be rendered in the `RelatedMedia` component. In the next section, we will look at the implementation of this `RelatedMedia` component.

The RelatedMedia component

In the play media page, beside the media loaded in the player, we will load a list of related media in the `RelatedMedia` component. The `RelatedMedia` component will take the list of related media as a prop from the `PlayMedia` component and render the details along with a video snapshot of each video in the list, as pictured in the following screenshot:



In the implementation of the `RelatedMedia` view, we iterate through the media array received in the props using the `map` function and render each media item's details and video snapshot, as shown in the following code structure:

`mern-mediastream/client/media/RelatedMedia.js`

```
{props.media.map((item, i) => {
  return
    <span key={i}>... video snapshot ... | ... media details ...</span>
  )
})
```

In this structure, to render the video snapshot for each media item, we will use a basic `ReactPlayer` without the controls, as follows:

mern-mediastream/client/media/RelatedMedia.js

```
<Link to={"/media/" + item._id}>
  <ReactPlayer url={'/api/media/video/' + item._id}
    width='160px'
    height='140px' />
</Link>
```

We wrap the `ReactPlayer` with a link to the individual view of this media. So, clicking on the given video snapshot will re-render the `PlayMedia` view to load the linked media's details. Beside the snapshot, we will display the details of each video including title, genre, created date, and the number of views, with the following code:

mern-mediastream/client/media/RelatedMedia.js

```
<Typography type="title" color="primary">{item.title}</Typography>
<Typography type="subheading"> {item.genre} </Typography>
<Typography component="p">
  {(new Date(item.created)).toDateString()}
</Typography>
<Typography type="subheading">{item.views} views</Typography>
```

This will render the details next to the video snapshot for each media in the related media list that is received in the props.

To render this `RelatedMedia` component in the play media page, we have to add it to the `PlayMedia` component. The `PlayMedia` component will use the related media list API implemented earlier in this section to retrieve the related media from the backend, and then pass it in the props to the `RelatedMedia` component. In the next section, we will discuss the implementation of this `PlayMedia` component.

The PlayMedia component

The `PlayMedia` component will render the play media page. This component consists of the `Media` and `RelatedMedia` child components along with an autoplay toggle, and it provides data to these components when it loads in the view.

To render the `PlayMedia` component when individual media links are accessed by the user, we will add a `Route` in `MainRouter` and mount `PlayMedia` at `'/media/:mediaId'`, as follows:

mern-mediastream/client/MainRouter.js

```
| <Route path="/media/:mediaId" component={PlayMedia}/>
```

When the `PlayMedia` component mounts, it will fetch the media data and the related media list from the server with `useEffect` hooks based on the `mediaId` parameter in the route link.

In one `useEffect` hook, it will fetch the media to be loaded in the media player, as shown in the following code:

mern-mediastream/client/media/PlayMedia.js

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

  read({mediaId: props.match.params.mediaId}, signal).then((data) => {
    if (data && data.error) {
      console.log(data.error)
    } else {
      setMedia(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [props.match.params.mediaId])
```

The media ID from the route path is accessed in the `props.match` received from the React Router components. It is used in the call to the `read` API fetch method to retrieve the media details from the server. The received `media` object is set in the state so that it can be rendered in the `Media` component.

In another `useEffect` hook, we use the same media ID to call the `listRelated` API fetch method, as shown in the following code.

`mern-mediastream/client/media/PlayMedia.js`

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

  listRelated({
    mediaId: props.match.params.mediaId}, signal).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      setRelatedMedia(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [props.match.params.mediaId])
```

The `listRelated` API fetch method retrieves the related media list from the server and sets the values to the state so that it can be rendered in the `RelatedMedia` component.

The media and related media list values stored in the state are used to pass relevant props to these child components that are added in the view. For example, in the following code, the `RelatedMedia` component is only rendered if the list of related media contains any media, and the list is passed to it as a prop:

`mern-mediastream/client/media/PlayMedia.js`

```
{relatedMedia.length > 0 &&
  (<RelatedMedia media={relatedMedia}/>) }
```

Later in the chapter, in the *Autoplaying related media* section, we will add the `Autoplay` toggle component above the `RelatedMedia` component only if the length of the related media list is greater than `0`. We will also discuss the implementation of the `handleAutoPlay` method that will be passed as a prop to the `Media` component. It will also receive the `media` detail object, and the video URL for the first item in the related media list, which will be treated as the next URL to play. The `Media` component is added to `PlayMedia`, along with these props, as shown in the following code:

`mern-mediastream/client/media/PlayMedia.js`

```
const nextUrl = relatedMedia.length > 0
    ? `/media/${relatedMedia[0]._id}`
    : ''
<Media media={media}
      nextUrl={nextUrl}
      handleAutoplay={handleAutoplay}/>
```

This `Media` component renders the media details on the play media page, and also a customized media player that allows viewers to control the streaming of the video. In the next section, we will discuss the implementation of this customized media player and complete this core feature of the play media page.

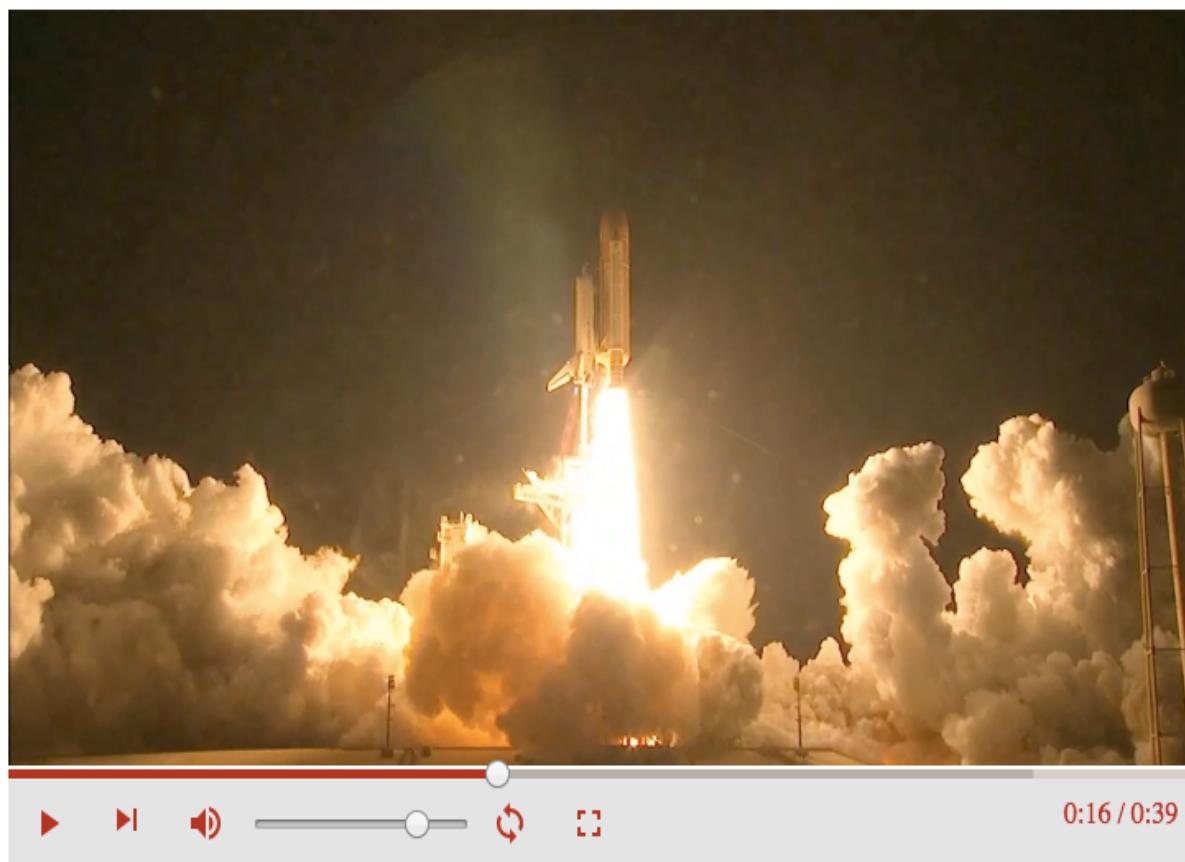
Customizing the media player

In MERN Mediastream, we want to provide users with a media player with more controls than those available in the default browser options, and with a look that matches the rest of the application. We will customize the player controls on `ReactPlayer` to replace these default controls with a custom look and functionality, as seen in the following screenshot:

Endeavour Lift-off

Space Exploration

527 views



Cecil McQueen

Published on Wed Feb 21 2018

Space Shuttle Endeavour is a retired orbiter from NASA's Space Shuttle program and the fifth and final operational shuttle built. Endeavour completed 25 missions, spent 299 days in orbit, and orbited Earth 4,671 times while traveling 122,883,151 miles.

The controls will be added below the video and will include the progress seeking bar; the play, pause, next, volume, loop, and fullscreen options; and will also display full duration of the video and the amount that's been played. In the following sections, we will first update the `Media` component discussed in the previous chapter, [Chapter 11, Building a Media Streaming Application](#), to accommodate the new player features. Then, we will initialize a `MediaPlayer` component that will

contain the new player, before implementing functionality for the custom media controls in this player.

Updating the Media component

The existing `Media` component contains a basic `ReactPlayer` with default browser controls for playing a given video. We will replace this `ReactPlayer` with a new `MediaPlayer` component that we will begin implementing in the next section. The `MediaPlayer` component will contain a customized `ReactPlayer`, and it will be added to the `Media` component code as follows:

`mern-mediastream/client/media/Media.js`

```
const mediaUrl = props.media._id  
  ? `/api/media/video/${props.media._id}`  
  : null  
...  
<MediaPlayer srcUrl={mediaUrl}  
            nextUrl={props.nextUrl}  
            handleAutoplay={props.handleAutoplay}/>
```

While adding this `MediaPlayer` component to the `Media` component, it will be passed the current video's source URL, the next video's source URL, and the `handleAutoPlay` method, which are received as `props` in the `Media` component from the `PlayMedia` component. These URL values and the autoplay handling method will be used in the `MediaPlayer` component to add various video-playing options. In the next section, we will begin implementing this `MediaPlayer` component by initializing the different values needed for adding functional controls to the custom media player.

Initializing the media player

We will implement the customized media player in the `MediaPlayer` component. This player will render the video streamed from the backend and provide the user with different control options. We will incorporate this media-playing functionality and the custom control options in the `MediaPlayer` using a `ReactPlayer` component. The `ReactPlayer` component, as discussed in the previous chapter, provides a range of customizations that we will leverage for the media player features to be added in this application.

While defining the `MediaPlayer` component, we will begin by initializing the `ReactPlayer` component with starting values for the controls, before we add the custom functionalities and corresponding user-action handling code for each control.



The control values we customize will correspond to the props allowed in the `ReactPlayer` component. To see a list of available props and an explanation of each, visit github.com/CookPet/e/react-player#props.

First, we need to set the initial control values in the component's state. We will start with control values that correspond to the following:

- The playing state of the media
- The volume of the audio
- The muted state
- The duration of the video
- The seeking state
- The playback rate of the video
- The loop value
- The fullscreen value
- Video errors
- The played, loaded, and ended states of the video getting streamed

The code to initialize these values in the component will be added as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
const [playing, setPlaying] = useState(false)
const [volume, setVolume] = useState(0.8)
const [muted, setMuted] = useState(false)
const [duration, setDuration] = useState(0)
const [seeking, setSeeking] = useState(false)
const [playbackRate, setPlaybackRate] = useState(1.0)
const [loop, setLoop] = useState(false)
const [fullscreen, setFullscreen] = useState(false)
const [videoError, setVideoError] = useState(false)
const [values, setValues] = useState({
  played: 0, loaded: 0, ended: false
})
```

These values set in the state will allow us to customize the functionalities of the corresponding controls in the `ReactPlayer` component, which we discuss in detail in the next section.

In the `MediaPlayer` component's view code, we will add this `ReactPlayer` with these control values and source URL, using the prop sent from the `Media` component, as shown in the following code:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
<ReactPlayer
  ref={ref}
  width={fullscreen ? '100%':'inherit'}
  height={fullscreen ? '100%':'inherit'}
  style={fullscreen ? {position:'relative'} : {maxHeight: '500px'}}
  config={{ attributes: { style: { height: '100%', width: '100%' } } }}
  url={props.srcUrl}
  playing={playing}
  loop={loop}
  playbackRate={playbackRate}
  volume={volume}
  muted={muted}
  onEnded={onEnded}
  onError={showVideoError}
  onProgress={onProgress}
  onDuration={onDuration}/>
```

Besides setting the control values, we will also add styling to the player, depending on whether it is in fullscreen mode. We also need

to get a reference to this player element rendered in the browser so that it can be used in the change-handling code for the custom controls. We will use the `useRef` React hook to initialize the reference to `null` and then set it to the corresponding player element using the `ref` method, as defined in the following code:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
let playerRef = useRef(null)
const ref = player => {
    playerRef = player
}
```

The value in `playerRef` will give access to the player element rendered in the browser. We will use this reference to manipulate the player as required, to make the custom controls functional.

As a final step for initializing the media player, we will add code for handling errors thrown by the player if the specified video source cannot be loaded for any reason. We will define a `showVideoError` method that will be invoked when a video error occurs.

The `showVideoError` method will be defined as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
const showVideoError = e => {
    console.log(e)
    setVideoError(true)
}
```

This method will render an error message in the view above the media player. We can show this error message conditionally by adding the following code in the view above the `ReactPlayer`:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
{videoError && <p className={classes.videoError}>Video Error. Try again later.</p>}
```

This will render the video error message when an error occurs. As we will allow users to play another video in the player from the

related media list, we will reset the error message if a new video is loaded. We can hide the error message when a new video loads with a `useEffect` hook, by ensuring the `useEffect` only runs when the video source URL changes, as shown in the following code:

`mern-mediastream/client/media/MediaPlayer.js`

```
| useEffect(() => {  
|   setVideoError(false)  
| }, [props.srcUrl])
```

This will ensure the error message isn't shown when a new video is loaded and streaming correctly.

With these initial control values set and the `ReactPlayer` added to the component, in the next section, we can begin customizing how these controls will appear and function in our application.

Custom media controls

We will add custom player control elements below the video rendered in the `MediaPlayer` component and manipulate their functionality using the options and events provided by the `ReactPlayer` library. In the following sections, we will implement the play, pause, and replay controls; the play next control; the loop functionality; volume control options; progress control options; fullscreen option, and also display full duration of the video and the amount that's been played.

Play, pause, and replay

Users will be able to play, pause, and replay the current video. We will implement these three options using [Material-UI](#) components bound to `ReactPlayer` attributes and events. The play, pause, and replay options will render as shown in the following screenshot:



To implement the play, pause, and replay functionality, we will add a play, pause, or replay icon button conditionally depending on whether the video is playing, is paused, or has ended, as shown in the following code:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
<IconButton color="primary" onClick={playPause}>
  <Icon>{playing ? 'pause' : (ended ? 'replay' : 'play_arrow')}</Icon>
</IconButton>
```

The play, pause, or replay icons are rendered in this `IconButton` based on the outcome of the ternary operator.

When the user clicks the button, we will update the `playing` value in the state, so the `ReactPlayer` is also updated. We achieve this by invoking the `playPause` method when this button is clicked. The `playPause` method is defined as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

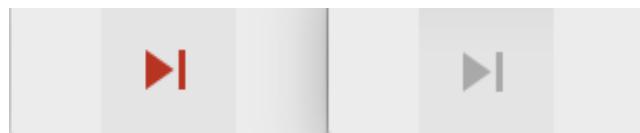
```
const playPause = () => {
  setPlaying(!playing)
}
```

The updated value of `playing` in the state will play or pause the video in the `ReactPlayer` component accordingly. In the next section, we will

see how we can add a control option that will allow us to play the next video from the list of related media.

Play next

Users will be able to play the next video in the related media list using a play next button, which will render depending on whether the next video is available or not. The two versions of this play next button will display as shown in the following screenshot:



The play next button will be disabled if the related list does not contain any media. The play next icon will basically link to the next URL value passed in as a prop from `PlayMedia`. This play next button will be added to the `MediaPlayer` view, as follows:

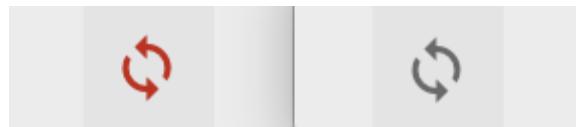
`mern-mediastream/client/media/MediaPlayer.js`

```
| <IconButton disabled={ !props.nextUrl} color="primary">
|   <Link to={props.nextUrl}>
|     <Icon>skip_next</Icon>
|   </Link>
| </IconButton>
```

Clicking on this play next button will reload the `PlayMedia` component with the new media details and start playing the video. In the next section, we will add a control option that will allow the current video to be played in a loop.

Loop when a video ends

Users will be able to set the current video to keep playing in a loop, using a loop button. The loop button will render in two states, set and unset, as shown in the following screenshot:



This loop icon button will display in a different color to indicate whether it has been set or unset by the user. The code for rendering this loop button will be added to the `MediaPlayer`, as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
<IconButton color={loop ? 'primary' : 'default'}  
          onClick={onLoop}>  
  <Icon>loop</Icon>  
</IconButton>
```

The loop icon color will change based on the value of `loop` in the state. When this loop icon button is clicked, we will update the `loop` value in the state by invoking the `onLoop` method, which is defined as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
const onLoop = () => {  
  setLoop(!loop)  
}
```

The video will play on loop when this `loop` value is set to `true`. We will need to catch the `onEnded` event, to check whether `loop` has been set to `true`, so the `playing` value can be updated accordingly. When a video reaches the end, the `onEnded` method will be invoked. This `onEnded` method will be defined as follows:

mern-mediastream/client/media/MediaPlayer.js

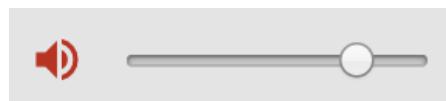
```
const onEnded = () => {
  if(loop){
    setPlaying(true)
  } else{
    setValues({...values, ended: true})
    setPlaying(false)
  }
}
```

So, if the `loop` value is set to `true`, when the video ends it will start playing again; otherwise, it will stop playing and render the replay button. In the next section, we will add controls for setting the volume of the video.

Volume control

In order to control the volume of the video being played, users will have the option to increase or decrease the volume, as well as to mute or unmute. The rendered volume controls will be updated based on the user action and current value of the volume. The different states of the volume controls will be as follows:

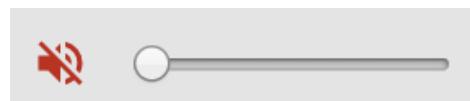
- A volume-up icon will be rendered if the volume is raised, as shown in the following screenshot:



- A volume-off icon will be rendered if the user decreases the volume to zero, as pictured next:



- A volume-mute icon button will be shown if the user clicks the icon to mute the volume, as shown next:



To implement this, we will conditionally render the different icons in an `IconButton`, based on the `volume`, `muted`, `volume_up`, and `volume_off` values, as shown in the following code:

```
<IconButton color="primary" onClick={toggleMuted}>
  <Icon> {volume > 0 && !muted && 'volume_up' ||
            muted && 'volume_off' ||
            volume==0 && 'volume_mute'} </Icon>
</IconButton>
```

When this `IconButton` is clicked, it will either mute or unmute the volume by invoking the `toggleMuted` method, which is defined as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
| const toggleMuted = () => {
|   setMuted(!muted)
| }
```

The volume will be muted or unmuted, depending on the current value of `muted` in the state. To allow users to increase or decrease the volume, we will add an input element of type `range` that will allow users to set a volume value between `0` and `1`. This input element will be added to the code, as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
| <input type="range"
|   min={0}
|   max={1}
|   step='any'
|   value={muted? 0 : volume}
|   onChange={changeVolume} />
```

Changing the `value` on the input range will set the `volume` value in the state accordingly by invoking the `changeVolume` method. This `changeVolume` method will be defined as follows:

```
mern-mediastream/client/media/MediaPlayer.js
```

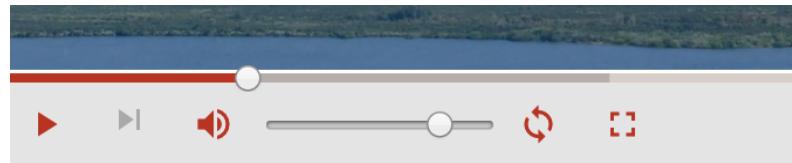
```
| const changeVolume = e => {
|   setVolume(parseFloat(e.target.value))
| }
```

The `volume` value changed in the state will be applied to the `ReactPlayer`, and this will set the volume of the current media being played. In the next section, we will add options to control the progression of the video being played.

Progress control

In the media player, users will see how much of the video has been loaded and played in a progress bar. To implement this feature, we will use a Material-UI `LinearProgress` component to indicate how much of the video has buffered, and how much has been played already. Then, we'll combine this component with an input element of type `range` to give users the ability to move the time slider to a different part of the video and play from there.

This time slider and progress bar will render as shown in the following screenshot:



The `LinearProgress` component will use the `played` and `loaded` values in the state to render these bars. It will take the `played` and `loaded` values to show each in a different color, as shown in the following code:

`mern-mediastream/client/media/MediaPlayer.js`

```
<LinearProgress color="primary" variant="buffer"
    value={values.played*100} valueBuffer={values.loaded*100}
    style={{width: '100%'}}
    classes={{
        colorPrimary: classes.primaryColor,
        dashedColorPrimary : classes.primaryDashed,
        dashed: classes.dashed
    }}
/>
```

The look and color for each progress bar will be determined by the styles you define for the `primaryColor`, `dashedColorPrimary`, and `dashed` classes.

To update the `LinearProgress` component when the video is playing or loading, we will use the `onProgress` event listener to set the current values for `played` and `loaded`. The `onProgress` method will be defined as shown in the following code:

mern-mediastream/client/media/MediaPlayer.js

```
const onProgress = progress => {
  if (!seeking) {
    setValues({...values, played: progress.played, loaded:
  progress.loaded})
  }
}
```

We only want to update the time slider if we are not currently seeking, so we first check the `seeking` value in the state before setting the `played` and `loaded` values.

For time-sliding control, we will add the range input element and define styles, as highlighted in the following code, to place it over the `LinearProgress` component. The current value of the range will update as the `played` value changes, so the range value seems to be moving with the progression of the video. This input element representing the time slider will be added to the media player, as shown in the following code:

mern-mediastream/client/media/MediaPlayer.js

```
<input type="range" min={0} max={1}
      value={values.played} step='any'
      onMouseDown={onSeekMouseDown}
      onChange={onSeekChange}
      onMouseUp={onSeekMouseUp}
      style={{ position: 'absolute',
              width: '100%',
              top: '-7px',
              zIndex: '999',
              '-webkit-appearance': 'none',
              backgroundColor: 'rgba(0,0,0,0)' }}>
/>
```

In the case where the user drags and sets the range picker on their own, we will add code to handle the `onMouseDown`, `onMouseUp`, and `onChange`

events to start the video from the desired position.

When the user starts dragging by holding the mouse down, we will set `seeking` to `true` so that the progress values are not set in `played` and `loaded`. This will be achieved with the `onSeekMouseDown` method, which is defined as follows:

mern-mediastream/client/media/MediaPlayer.js

```
| const onSeekMouseDown = e => {
|   setSeeking(true)
| }
```

As the range value change occurs, we will invoke the `onSeekChange` method to set the `played` value and also the `ended` value, after checking whether the user dragged the time slider to the end of the video. This `onSeekChange` method will be defined as follows:

mern-mediastream/client/media/MediaPlayer.js

```
| const onSeekChange = e => {
|   setValues({...values, played:parseFloat(e.target.value),
|             ended: parseFloat(e.target.value) >= 1})
| }
```

When the user is done dragging and lifts their click on the mouse, we will set `seeking` to `false`, and set the `seekTo` value for the media player to the current value set in the input range. The `onSeekMouseUp` method will be executed when the user is done seeking, and it is defined as follows:

mern-mediastream/client/media/MediaPlayer.js

```
| const onSeekMouseUp = e => {
|   setSeeking(false)
|   playerRef.seekTo(parseFloat(e.target.value))
| }
```

This way, the user will be able to select any part of the video to play from, and also get visual information on the time progress of the

video being streamed. In the next section, we will add a control that will allow the user to view the video in fullscreen mode.

Fullscreen

Users will be able to view the video in fullscreen mode by clicking the fullscreen button in the controls. The fullscreen button for the player will be rendered as shown in the following screenshot:



In order to implement a fullscreen option for the video, we will use the `screenfull` Node module to track when the view is in fullscreen, and `findDOMNode` from `react-dom` to specify which **Document Object Model (DOM)** element will be made fullscreen with `screenfull`.

To set up the `fullscreen` code, we first install `screenfull`, by running the following command from the command line:

```
| yarn add screenfull
```

Then, we will import `screenfull` and `findDOMNode` into the `MediaPlayer` component, as shown in the following code:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
| import screenfull from 'screenfull'  
| import { findDOMNode } from 'react-dom'
```

When the `MediaPlayer` component mounts, we will use a `useEffect` hook to add a `screenfull` change event listener that will update the `fullscreen` value in the state to indicate whether the screen is in fullscreen or not. The `useEffect` hook will be added as follows, with the `screenfull` change listener code:

```
mern-mediastream/client/media/MediaPlayer.js
```

```
useEffect(() => {
  if (screenfull.enabled) {
    screenfull.on('change', () => {
      let fullscreen = screenfull.isFullscreen ? true : false
      setFullscreen(fullscreen)
    })
  }
}, [])
```

This `fullscreen` value set in the state will be updated when the user interacts with the button for rendering the video in fullscreen mode. In the view, we will add an `icon` button for `fullscreen` with the other control buttons, as shown in the following code:

mern-mediastream/client/media/MediaPlayer.js

```
<IconButton color="primary" onClick={onClickFullscreen}>
  <Icon>fullscreen</Icon>
</IconButton>
```

When the user clicks this button, we will use `screenfull` and `findDOMNode` to make the video player fullscreen by invoking the `onClickFullscreen` method, which is defined as follows:

mern-mediastream/client/media/MediaPlayer.js

```
const onClickFullscreen = () => {
  screenfull.request(findDOMNode(playerRef))
}
```

We access the element that renders the media player in the browser by using the `playerRef` reference in `findDOMNode` and make it fullscreen by using `screenfull.request`. The user can then watch the video in fullscreen, where they can press *Esc* at any time to exit fullscreen and get back to the `PlayMedia` view. In the next section, we will implement the final customization in the media player controls to display the total length of the video, and how much of it was already played.

Played duration

In the custom media controls section of the media player, we want to show the time that has already passed and the total duration of the video in a readable time format, as shown in the following screenshot:



To show the time, we can utilize the `HTML time element`, which takes a `datetime` value, and add it to the view code in `MediaPlayer`, as follows:

mern-mediastream/client/media/MediaPlayer.js

```
<time dateTime={`${P${Math.round(duration * played)}S`}>
  {format(duration * played)}
</time> /
<time dateTime={`${P${Math.round(duration)}S`}>
  {format(duration)}
</time>
```

In the `dateTime` attribute for these `time` elements, we provide the total rounded-off seconds that represent the played duration or the total duration of the video. We will get this total `duration` value for a video by using the `onDuration` event and then set it to the state, so it can be rendered in the `time` element. The `onDuration` method is defined as follows:

mern-mediastream/client/media/MediaPlayer.js

```
const onDuration = (duration) => {
  setDuration(duration)
}
```

To make the duration and already played time values readable, we will use the following `format` function:

mern-mediastream/client/media/MediaPlayer.js

```
const format = (seconds) => {
  const date = new Date(seconds * 1000)
  const hh = date.getUTCHours()
  let mm = date.getUTCMinutes()
  const ss = ('0' + date.getUTCSeconds()).slice(-2)
  if (hh) {
    mm = ('0' + date.getUTCMinutes()).slice(-2)
    return `${hh}:${mm}:${ss}`
  }
  return `${mm}:${ss}`
}
```

This `format` function takes the duration value in seconds and converts it to the `hh/mm/ss` format, using methods from the JavaScript Date API.

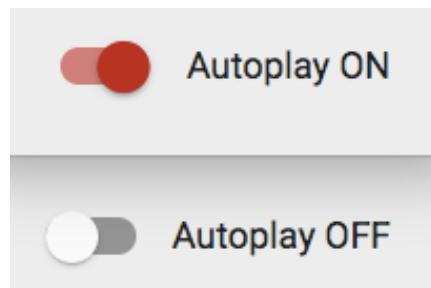
The controls added to this custom media player are all mostly based on some of the available functionality provided in the `ReactPlayer` module, and its examples in the official documentation. While implementing the custom media player for this application, we updated and added the associated playing controls, looping option, volume controls, progress seeking control, fullscreen viewing option, and a display of the video duration. There are more options available for further customizations and extensions in `ReactPlayer` that may be explored depending on specific feature requirements. With the different functionalities of the customized media player implemented, in the next section, we can start discussing the implementation of autoplaying videos in this player from a list of available media.

Autoplaying related media

In the play media page, users will have the option to autoplay one video after the other from the related media list. To make this feature possible, the `PlayMedia` component will manage the autoplay state, which will determine the data and how it will be rendered next in the `MediaPlayer` and `RelatedMedia` components after the current video finishes streaming in the player. In the following sections, we will complete this autoplay functionality by adding a toggle in the `PlayMedia` component and implementing the `handleAutoplay` method, which needs to be called when a video ends in the `MediaPlayer` component.

Toggling autoplay

On the play media page, we will add an autoplay toggle option above the related media list. Besides letting the user set autoplay, the toggle will also indicate whether it is currently set or not, as shown in the following screenshot:



To add the autoplay toggle option, we will use a Material-UI `Switch` component along with a `FormControlLabel`, and add it to the `PlayMedia` component over the `RelatedMedia` component. It will only be rendered when there are media in the related media list. We will add this `Switch` component representing the autoplay toggle as shown in the following code:

```
mern-mediastream/client/media/PlayMedia.js
```

```
<FormControlLabel
  control={
    <Switch
      checked={autoPlay}
      onChange={handleChange}
      color="primary"
    />
  }
  label={autoPlay ? 'Autoplay ON':'Autoplay OFF'}
/>
```

The autoplay toggle label will render according to the current value of `autoPlay` in the state. To handle the change to the toggle when the user interacts with it, and to reflect this change in the state's `autoPlay` value, we will use the following `onChange` handler function:

```
mern-mediastream/client/media/PlayMedia.js
```

```
| const handleChange = (event) => {  
|   setAutoPlay(event.target.checked)  
| }
```

This `autoPlay` value, which represents whether the user chose to autoplay all the media, will determine what happens when the current video finishes streaming. In the next section, we will discuss how the autoplay behavior will be integrated with the child components in `PlayMedia`, depending on the toggled value set for `autoPlay` by the user.

Handling autoplay across components

When a user selects to set the Autoplay toggle to ON, the functionality desired here is that when a video ends, if `autoPlay` is set to `true` and the current related list of media is not empty, `PlayMedia` should load the media details of the first video in the related list.

In turn, the `Media` and `MediaPlayer` components should update with the new media details, start playing the new video, and render the controls on the player appropriately. The list in the `RelatedMedia` component should also update with the current media removed from the list, so only the remaining playlist items are visible.

In order to handle this autoplay behavior across the `PlayMedia` component and its child components, `PlayMedia` passes a `handleAutoPlay` method to the `Media` component as a prop to be used by the `MediaPlayer` component when a video ends. The `handleAutoPlay` method is defined as shown in the following code:

`mern-mediastream/client/media/PlayMedia.js`

```
const handleAutoplay = (updateMediaControls) => {
  let playList = relatedMedia
  let playMedia = playList[0]
  if(!autoPlay || playList.length == 0 )
    return updateMediaControls()

  if(playList.length > 1){
    playList.shift()
    setMedia(playMedia)
    setRelatedMedia(playList)
  }else{
    listRelated({
      mediaId: playMedia._id}).then((data) => {
        if (data.error) {
          console.log(data.error)
        } else {
```

```
        setMedia(playMedia)
        setRelatedMedia(data)
    }
}
}
```

This `handleAutoplay` method takes care of the following when a video ends in the `MediaPlayer` component:

- It takes a callback function from the `onEnded` event listener in the `MediaPlayer` component. This callback will be executed if autoplay is not set or the related media list is empty so that the controls on the `MediaPlayer` are rendered to show that the video has ended.
- If autoplay is set and there are more than one related media in the list, then:
 - The first item in the related media list is set as the current `media` object in the state so it can be rendered.
 - The related media list is updated by removing this first item, which will now start playing in the view.
- If autoplay is set and there is only one item in the related media list, this last item is set to `media` so it can start playing, and the `listRelated` fetch method is called to repopulate the `RelatedMedia` view with the related media for this last item.

With these steps covered within this `handleAutoplay` method, all the aspects of the play media page can be updated accordingly at the end of a video, if autoplay is set to `true`. In the next section, we will see how the `MediaPlayer` component utilizes this `handleAutoplay` method when the current video ends, in order to make the autoplay feature functional.

Updating the state when a video ends in MediaPlayer

The `MediaPlayer` component receives the `handleAutoplay` method as a prop from `PlayMedia`. This method will be utilized when the current video finishes playing in the player. Hence, we will update the listener code for the `onEnded` event to execute this method only when the `loop` is set to `false` for the current video. We don't want to play the next video if the user has decided to loop the current video. The `onEnded` method in `MediaPlayer` will be updated with the highlighted code shown in the following block:

`mern-mediastream/client/media/MediaPlayer.js`

```
const onEnded = () => {
  if(loop){
    setPlaying(true)
  } else{
    props.handleAutoplay(()=>{
      setValues({...values, ended: true})
      setPlaying(false)
    })
  }
}
```

In this code, a callback function is passed to the `handleAutoplay` method, in order to set the `playing` value to `false` and render the replay icon button instead of the play or pause icon button, after it is determined in `PlayMedia` that the autoplay has not been set or that the related media list is empty.

The autoplay functionality will continue playing the related videos one after the other with this implementation. This implementation demonstrates another way to update the state across the components when the values are interdependent.

With this autoplay functionality implemented, we have a complete play media page with a customized media player and a related media list that the user can choose to autoplay through like a playlist. In the next section, we will make this page SEO-friendly by SSR of this view with the media data populated in the backend.

Server-side rendering with data

SEO is important for any web application that delivers content to its users and wants to make the content easy to find. Generally, content on any web page will have a better chance of getting more viewers if the content is easily readable to search engines. When a search-engine bot accesses a web URL, it will get the SSR output. Hence, to make the content discoverable, the content should be part of the SSR output.

In MERN Mediastream, we will use the case of making media details popular across search engine results, to demonstrate how to inject data into an SSR view in a MERN-based application. We will focus on implementing SSR with data injected for the `PlayMedia` component that is returned at the `'/media/:mediaId'` path. The general implementation steps outlined here can be used to implement SSR with data for other views.

In the following sections, we will extend the SSR implementation discussed in [Chapter 4, Adding a React Frontend to Complete MERN](#). We will first define a static route configuration file and use it to update the existing SSR code in the backend to inject the necessary media data from the database. Then, we will update the frontend code to render this server-injected data in the view, and, finally, check if this SSR implementation works as expected.

Adding a route configuration file

In order to load data for the React views when these are rendered on the server, we will need to list the frontend routes in a route configuration file. This file may then be used with the `react-router-config` module, which provides static route configuration helpers for React Router.

We will first install the module by running the following command from the command line:

```
| yarn add react-router-config
```

Next, we will create a route configuration file that will list frontend React Router routes. This configuration will be used on the server to match these routes with incoming request URLs, to check whether data must be injected before the server returns the rendered markup in response to this request.

For the route configuration in MERN Mediastream, we will only list the route that renders the `PlayMedia` component and demonstrate how to server-render a specific component with data injected from the backend. The route configuration will be defined as follows:

```
mern-mediastream/client/routeConfig.js
```

```
import PlayMedia from './media/PlayMedia'
import { read } from './media/api-media.js'
const routes = [
  {
    path: '/media/:mediaId',
    component: PlayMedia,
    loadData: (params) => read(params)
  }
]
export default routes
```

For this frontend route and `PlayMedia` component, we specify the `read` fetch method from `api-media.js` as the `loadData` method. This can then be used to retrieve and inject the data into the `PlayMedia` view when the server generates the markup for this component, after receiving a request at `/media/:mediaId`. In the next section, we will use this route configuration to update the existing SSR code on the backend.

Updating SSR code for the Express server

We will update the existing basic SSR code in `server/express.js` to add the data-loading functionality for the React views that will get rendered server-side. In the following sections, we will first see how to use the route configuration to load the data that needs to be injected when the server renders React components. Then, we will integrate `isomorphic-fetch` so the server is able to make the `read` `fetch` call to retrieve the necessary data, using the same API fetching code from the frontend. Finally, we will inject this retrieved data into the markup generated by the server.

Using route configuration to load data

We will use the routes defined in the route configuration file to look for a matching route when the server receives any request. If a match is found, we will use the corresponding `loadData` method declared for this route in the configuration to retrieve the necessary data, before it is injected into the server-rendered markup representing the React frontend. We will perform these route-matching and data-loading actions in a method called `loadBranchData`, which is defined as follows:

mern-mediastream/server/express.js

```
import { matchRoutes } from 'react-router-config'
import routes from './client/routeConfig'
const loadBranchData = (location) => {
  const branch = matchRoutes(routes, location)
  const promises = branch.map(({ route, match }) => {
    return route.loadData
      ? route.loadData(branch[0].match.params)
      : Promise.resolve(null)
  })
  return Promise.all(promises)
}
```

This method uses `matchRoutes` from `react-router-config`, and the routes defined in the route configuration file, to look for a route matching the incoming request URL, which is passed as the `location` argument. If a matching route is found, then any associated `loadData` method will be executed to return a `Promise` containing the fetched data, or `null` if there were no `loadData` methods. The `loadBranchData` defined here will need to be called whenever the server receives a request, so if any matching route is found, we can fetch the relevant data and inject it into the React components while rendering server side. In the next section, we will ensure the fetch methods defined in the frontend code also

work on the server side, so these same methods also load the corresponding data from the server side.

Isomorphic-fetch

We will ensure that any fetch method we defined for the client code can also be used on the server by using the `isomorphic-fetch` Node module. We will first install the module by running the following command from the command line:

```
| yarn add isomorphic-fetch
```

Then, we will simply import `isomorphic-fetch` in `express.js`, as shown in the following code, to ensure fetch methods now work isomorphically both on the client and the server side:

```
mern-mediastream/server/express.js
```

```
| import 'isomorphic-fetch'
```

This `isomorphic-fetch` integration will make sure that the `read` `fetch` method, or any other fetch method that we defined for the client, can now be used on the server as well. Before this integration becomes functional, we need to ensure the fetch methods use absolute URLs, as discussed in the next section.

Absolute URLs

One issue with using `isomorphic-fetch` is that it currently requires the `fetch` URLs to be absolute. So, we need to update the URL used in the `read` fetch method, defined in `api-media.js`, into an absolute URL.

Instead of hardcoding a server address in the code, we will set a `config` variable in `config.js`, as follows:

```
mern-mediastream/config/config.js
```

```
| serverUrl: process.env.serverUrl || 'http://localhost:3000'
```

This will allow us to define and use separate absolute URLs for the API routes in development and in production.

Then, we will update the `read` method in `api-media.js` to make sure it uses an absolute URL to call the `read` API on the server, as highlighted in the following code:

```
mern-mediastream/client/media/api-media.js
```

```
| import config from '../config/config'  
| const read = (params) => {  
|   return fetch(config.serverUrl + '/api/media/' + params.mediaId, {  
|     method: 'GET'  
|   }).then((response) => { ... })
```

This will make the `read` fetch call compatible with `isomorphic-fetch`, so it can be used without a problem on the server side to retrieve the media data while server-rendering the `PlayMedia` component with data. In the next section, we will discuss how to inject this retrieved data into the server-generated markup representing the rendered React frontend.

Injecting data into the React app

In the existing SSR code in the backend, we use `ReactDOMServer` to convert the React app to markup. We will update this code in `express.js` to inject the retrieved data into the `MainRouter`, as shown in the following code:

mern-mediastream/server/express.js

```
...
loadBranchData(req.url).then(data => {
  const markup = ReactDOMServer.renderToString(
    sheets.collect(
      <StaticRouter location={req.url} context={context}>
        <ThemeProvider theme={theme}>
          <MainRouter data={data}/>
        </ThemeProvider>
      </StaticRouter>
    )
  )
  ...
}) .catch(err => {
  res.status(500).send({ "error": "Could not load React view with
data"})
})
...
...
```

We utilize the `loadBranchData` method to retrieve the relevant data for the requested view, then pass this data as a prop to the `MainRouter` component. For this data to be added correctly in the rendered `PlayMedia` component when the server generates the markup, we need to update the client-side code to consider this server-injected data, as discussed in the next section.

Applying server-injected data to client code

We will update the React code in the frontend to add considerations for the data that may be injected from the server if the view is being rendered server-side. For this MERN Mediastream application, on the client side, we will access the media data passed from the server, and add it to the `PlayMedia` view when the server receives a direct request to render this component. In the following sections, we will see how to pass the data received in the `MainRouter` to the `PlayMedia` component, and render it accordingly.

Passing data props to PlayMedia from MainRouter

While generating markup with `ReactDOMServer.renderToString`, we pass the preloaded data to `MainRouter` as a prop. We can access this data prop in the `MainRouter` component definition, as follows:

```
mern-mediastream/client/MainRouter.js
```

```
| const MainRouter = ({data}) => { ... }
```

To give `PlayMedia` access to this data from the `MainRouter`, we will change the `Route` component added originally to declare the route for `PlayMedia`, and pass this data as a prop, as shown in the following code:

```
mern-mediastream/client/MainRouter.js
```

```
|<Route path="/media/:mediaId"
|       render={ (props) =>
|           <PlayMedia {...props} data={data} />
|       }
|   />
```

The data prop sent to `PlayMedia` will need to be rendered in the view, as discussed next.

Rendering received data in PlayMedia

In the `PlayMedia` component, we will check for data passed from the server and set the values to the state so the media details are rendered in the view when the server is generating the corresponding markup. We will do this checking and assignment as shown in the following code:

```
mern-mediastream/client/media/PlayMedia.js
```

```
| if (props.data && props.data[0] != null) {  
|     media = props.data[0]  
|     relatedMedia = []  
| }
```

If media data is received in the props from the server, we assign it to the `media` value in the state. We also set the `relatedMedia` value to an empty array, as we do not intend to render the related media list in the server-generated version. This implementation will produce server-generated markup with media data injected in the `PlayMedia` view when the corresponding frontend route request is received directly on the server. In the next section, we will see how to ensure this implementation is actually working and successfully rendering server-generated markup with the data populated.

Checking the implementation of SSR with data

For MERN Mediastream, any of the links that render `PlayMedia` should now generate markup on the server side with media details preloaded. We can verify that the implementation for SSR with data is working properly by opening the app URL in a browser, with JavaScript turned off. In the following section, we will look into how to achieve this check in the Chrome browser, and what the resulting view should show to the user and to a search engine.

Testing in Chrome

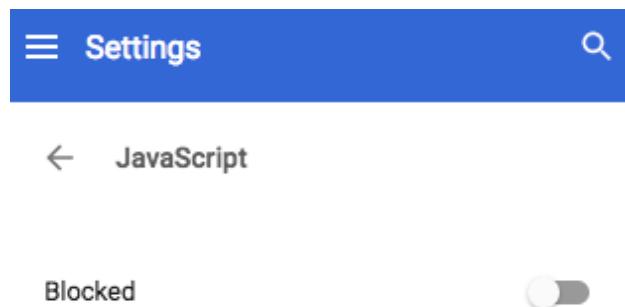
Testing this implementation in Chrome just requires updating the Chrome settings and loading the application in a tab, with JavaScript blocked. In the following sections, we will go over the steps to check whether the `PlayMedia` view renders with data when it is just server-generated markup.

Loading a page with JavaScript enabled

First, open the MERN Mediastream application in Chrome, then browse to any media link and let it render normally with JavaScript enabled. This should show the implemented `PlayMedia` view with the functioning media player and the related media list. Leave this tab open as we move on to the next step, to disable JavaScript in Chrome.

Disabling JavaScript from settings

To test how the server-generated markup is rendered in the view, we need to disable JavaScript on Chrome. For this, you can go to the advanced settings at `chrome://settings/content/javascript`, and use the toggle to block JavaScript, as shown in the following screenshot:



Now, refresh the media link in the MERN Mediastream tab, and there should be an icon next to the address URL, as shown in the following screenshot, indicating that JavaScript is indeed disabled:



The view that will be displayed in the browser at this point will only render the server-generated markup received from the backend. In the next section, we will discuss what the expected view is when JavaScript is blocked on the browser.

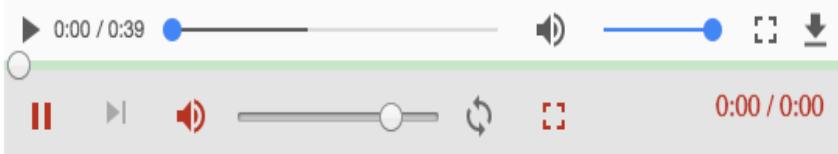
PlayMedia view with JavaScript blocked

When JavaScript is blocked in the browser, the `PlayMedia` view should render with only the media details populated. But the user interface is no longer interactive as JavaScript is blocked, and only the default browser controls are operational, as shown in the following screenshot:

Endeavour Lift-off

566 views

Space Exploration



Cecil McQueen

Published on Wed Feb 21 2018

Space Shuttle Endeavour is a retired orbiter from NASA's Space Shuttle program and the fifth and final operational shuttle built. Endeavour completed 25 missions, spent 299 days in orbit, and orbited Earth 4,671 times while traveling 122,883,151 miles.

This is the markup that a search-engine bot will read for media content, and also what a user will see when no JavaScript loads on the browser. If this implementation for SSR with data was not added to the application, then this view would render without the associated media details in this scenario, and hence the media information would not be read and indexed by search engines.

MERN Mediastream now has fully operational media-playing tools that will allow users to browse and play videos with ease. In addition, the media views that display individual media content items are now search-engine-optimized because of SSR with preloaded data.

Summary

In this chapter, we completely upgraded the play media page on MERN Mediastream. We first added custom media player controls, utilizing options available in the `ReactPlayer` component. Then, we incorporated the autoplay functionality for a related media playlist, after retrieving the related media from the database. Finally, we made the media details search-engine-readable by injecting data from the server when the view is rendered on the server.

You can apply the techniques explored in this chapter to build the play media page, to compose and build your own complex user interface with React components that are interdependent, and to add SSR with data for views that need to be SEO-friendly in your applications.

We have now explored advanced capabilities, such as streaming and SEO, with the MERN stack technologies. In the next two chapters, we will test the potential of this stack further by incorporating **virtual reality (VR)** elements into a full-stack web application using React 360.

Developing a Web-Based VR Game

The advent of **virtual reality (VR)** and **augmented reality (AR)** technologies is transforming how users interact with software and, in turn, the world around them. The possible applications of VR and AR are innumerable, and though the gaming industry has been an early adopter, these rapidly developing technologies have the potential to shift paradigms across multiple disciplines and industries.

In order to demonstrate how the MERN stack paired with React 360 can easily add VR capabilities to any web application, we will discuss and develop a dynamic, web-based VR game in this and the next chapter. In this chapter, we will focus on defining the features of the VR game. Additionally, we will go over the key 3D VR concepts that are relevant to implementing this VR game, before developing the game view using React 360.

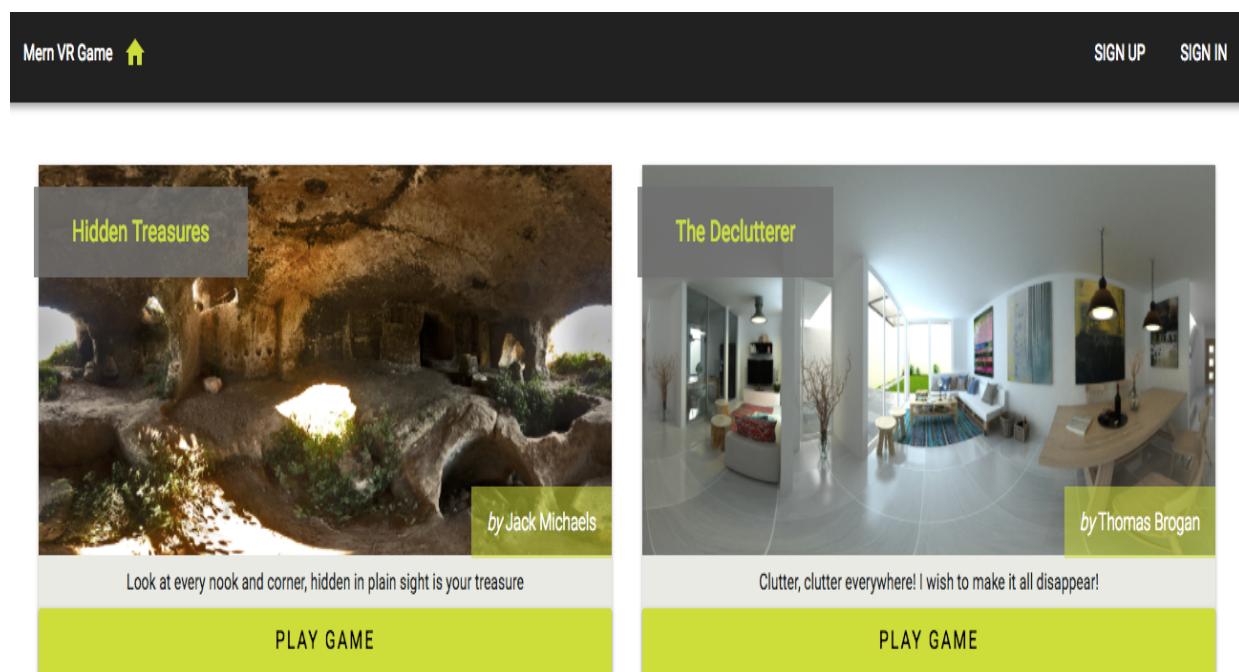
In this chapter, we will build the VR game using React 360 by covering the following topics:

- Introducing the MERN VR Game
- Getting started with React 360
- Key concepts for developing 3D VR applications
- Defining game details
- Building the game view in React 360
- Bundling the React 360 code to integrate with the MERN skeleton

After going over these topics, you will be able to apply 3D VR concepts and use React 360 to start building your own VR-based applications.

Introducing the MERN VR Game

The MERN VR Game web application will be developed by extending the MERN skeleton and integrating VR capabilities using React 360. It will be a dynamic, web-based VR game application, in which registered users can make their own games, and any visitor to the application can play these games. The home page of this application will list the games on the platform, as shown in the following screenshot:



The code to implement features of the VR game using React 360 is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter 13/MERNVR>. You can clone this code and run the application as you go through the code explanations in the rest of this chapter.

The features of the game will be simple enough to reveal the capabilities of introducing VR into a MERN-based application, without delving too deeply into the advanced concepts of React 360 that may be used to implement more complex VR features. In the next section, we will briefly define the features of a game in this application.

Game features

Each game in the MERN VR Game application will essentially be a different VR world, where users can interact with 3D objects placed at different locations in a 360-degree panoramic world.

The gameplay will be similar to that of a scavenger hunt, and to complete each game, users will have to find and collect the 3D objects that are relevant to the clue or description for each game. This means the game world will contain some VR objects that can be collected by the player and some VR objects that cannot be collected, but that may be placed by makers of the game as props or hints. Finally, the game will be won when all of the relevant 3D objects have been collected by the user.

In this chapter, we will build these game features using React 360, focusing primarily on VR and React 360 concepts that will be relevant to implementing the features defined here. Once the game features are ready, we will discuss how the React 360 code can be bundled and prepared for integration with the MERN application code developed in [Chapter 14](#), *Making the VR Game Dynamic Using MERN*. Before diving into the implementation of the game features with React 360, we will first look at setting up and getting started with React 360 in the next section.

Getting started with React 360

React 360 makes it possible to build VR experiences using the same declarative and component-based approach as in React. The underlying technology of React 360 makes use of the Three.js JavaScript 3D engine to render 3D graphics with WebGL within any compatible web browser and also provides us with access to VR headsets with the Web VR API.

Though React 360 builds on top of React and the apps run in the browser, React 360 has a lot in common with React Native, thus making React 360 apps cross-platform. This also means that some concepts of React Native are also applicable to React 360. Covering all of the React 360 concepts is outside the scope of this book; therefore, we will focus on the concepts that are required to build the game and integrate them with the MERN stack web application. In the following section, we will begin by setting up a React 360 project, which will be extended on later in the chapter in order to build the game features.

Setting up a React 360 project

React 360 provides developer tools that make it easy to start developing a new React 360 project. The steps to get started are detailed in the official React 360 documentation, so we will only summarize the steps here and point out the files that are relevant to developing the game.

Since we already have Node and Yarn installed for the MERN applications, we can start by installing the React 360 CLI tool by running the following command in the command line:

```
| yarn global add react-360-cli
```

Then, use this React 360 CLI tool to create a new application, and install the required dependencies by running the following command from the command line:

```
| react-360 init MERNVR
```

This will add the new React 360 application with all of the necessary files into a folder, named `MERNVR`, in the current directory. Finally, we can go into this folder in the command line, and run the application using the following command:

```
| yarn start
```

This `start` command will initialize the local development server, and the default React 360 application can be viewed in the browser at <http://localhost:8081/index.html>.

To update this starter application and implement our game features, we will modify code mainly in the `index.js` file with some minor updates in the `client.js` file, which can be found in the `MERNVR` project folder.

The default code in `index.js` for the starter application generated by React 360 should be as follows. Note that it renders a Welcome to React 360 text in a 360-degree world in the browser:

```
import React from 'react'
import { AppRegistry, StyleSheet, Text, View } from 'react-360'

export default class MERNVR extends React.Component {
  render() {
    return (
      <View style={styles.panel}>
        <View style={styles.greetingBox}>
          <Text style={styles.greeting}>
            Welcome to React 360
          </Text>
        </View>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  panel: {
    // Fill the entire surface
    width: 1000,
    height: 600,
    backgroundColor: 'rgba(255, 255, 255, 0.4)',
    justifyContent: 'center',
    alignItems: 'center',
  },
  greetingBox: {
    padding: 20,
    backgroundColor: '#000000',
    borderColor: '#639dda',
    borderWidth: 2,
  },
  greeting: {
    fontSize: 30,
  }
})

AppRegistry.registerComponent('MERNVR', () => MERNVR)
```

This `index.js` file contains the application's content and the main code, including the view and style code. The code in `client.js` contains the boilerplate that connects the browser to the React application in `index.js`. The default `client.js` file in the starter project folder should look like this:

```
import {ReactInstance} from 'react-360-web'
```

```
function init(bundle, parent, options = {}) {
  const r360 = new ReactInstance(bundle, parent, {
    // Add custom options here
    fullScreen: true,
    ...options,
  })

  // Render your app content to the default cylinder surface
  r360.renderToSurface(
    r360.createRoot('MERNVR', { /* initial props */ }),
    r360.getDefaultSurface()
  )

  // Load the initial environment
  r360.compositor.setBackground(r360.getAssetURL('360_world.jpg'))
}

window.React360 = {init}
```

This code executes the React code defined in `index.js`, essentially creating a new instance of React 360 and loading the React code by attaching it to the DOM.

With this, the default React 360 starter project is set up and ready for extension. Before modifying this code to implement the game, in the next section, we will first look at some of the key concepts related to developing 3D VR experiences, in the context of how these concepts are applied with React 360.

Key concepts for developing the VR game

Before creating VR content and an interactive 360-degree experience for the game, we will highlight some of the relevant aspects of the virtual world, and how React 360 can be used to work with these VR concepts. Given the wide range of possibilities in the VR space and the various options available with React 360, we need to identify and explore the specific concepts that will enable us to implement the interactive VR features we defined for the game. In the following sections, we will discuss the images that will make up the 360-degree world of the game, the 3D positioning system, along with the React 360 components, APIs, and input events that will be utilized to implement the game.

Equirectangular panoramic images

The VR world for the game will be composed of a panoramic image that is added to the React 360 environment as a background image.

Panorama images are generally 360-degree images or spherical panoramas projected onto a sphere that completely surrounds the viewer. A common and popular format for 360-degree panorama images is the equirectangular format. React 360 currently supports mono and stereo formats for equirectangular images.



To learn more about the 360 images and video support in React 360, refer to the React 360 documentation at facebook.github.io/react-360/docs/setup.html.

The photograph shown here is an example of an equirectangular, 360-degree panoramic image. To set the world background for a game in MERN VR Game, we will use this kind of image:



An equirectangular panoramic image consists of a single image with an aspect ratio of 2:1, where the width is twice the height. These images are created with a special 360-degree camera. An



excellent source of equirectangular images is Flickr; you just need to search for the equirectangular tag.

Creating the game world by setting the background scene using an equirectangular image in a React 360 environment will make the VR experience immersive and transport the user to a virtual location. To enhance this experience and add 3D objects in this VR world effectively, we need to learn more about the layout and coordinate system relevant to the 3D space, which is discussed next.

3D position – coordinates and transforms

We need to understand positioning and orientation in the VR world space, in order to place 3D objects at the desired locations and to make the VR experience feel more real. In the following sections, we will review the 3D coordinate system to help us to determine the location of a virtual object in the 3D space, and the transform capabilities in React 360, which will allow us to position, orient, and scale objects as required.

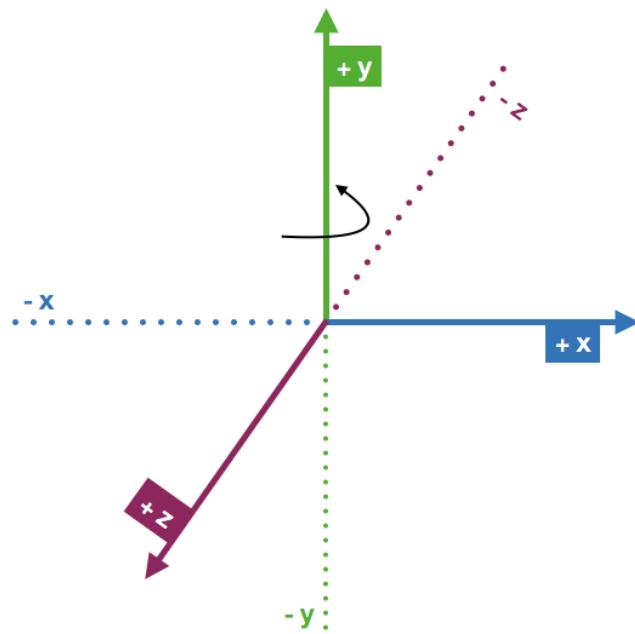
3D coordinate system

For mapping in a 3D space, React 360 uses a three-dimensional meter-based coordinate system that is similar to the OpenGL® 3D coordinate system. This allows individual components to be transformed, moved, or rotated in 3D in relation to the layout in their parent component.



The 3D coordinate system used in React 360 is a right-handed system. This means the positive x axis is to the right, the positive y axis points upward, and the positive z axis is backward. This provides a better mapping with common coordinate systems of the world space in assets and 3D world modeling.

If we try to visualize the 3D space, the user starts out at the center of the **x-y-z** axes pictured in the following diagram:



The **z** axis points forward toward the user and the user looks out at the **-z** axis direction. The **y** axis runs up and down, whereas the **x** axis runs from side to side. The curved arrow in the diagram shows the direction of the positive rotation values.

While deciding where and how to place 3D objects in the 360-degree world, we will have to set values according to this 3D coordinate system. In the next section, we will demonstrate how to place 3D objects using React 360 by setting values in transform properties.

Transforming 3D objects

The position and orientation of a 3D object will be determined by its transform properties, which will have values corresponding to the 3D coordinate system. In the following screenshot, the same 3D book object is placed in two different positions and orientations by changing the `transform` properties in the `style` attribute of a React 360 Entity component that is rendering the 3D object:



This transform feature is based on the transform style used in React, which React 360 extends to be fully 3D, considering the **x-y-z** axes. The `transform` properties are added to the `Entity` components in the `style` attribute as an array of keys and values in the following form:

```
style={ { ...  
        transform: [  
            {TRANSFORM_COMMAND: TRANSFORM_VALUE},  
            ...  
        ]  
    ... } }
```

The transform commands and values that are relevant to the 3D objects to be placed in our games are `translate [x, y, z]`, with values in meters; `rotate [x, y, z]`, with values in degrees; and `scale`, to determine the size of the object across all axes. We can also utilize the `matrix` command instead, which accepts an array of 16 numbers representing the translation, rotation, and scale values.



To learn more about the React 360 3D coordinates and transforms, refer to the React 360 documentation at facebook.github.io/react-360/docs/setup.html.

We will utilize these transform properties to position and orient 3D objects according to the 3D coordinate system while placing objects in the game world built using React 360. In the next section, we will go over the React 360 components that will allow us to build the game world.

React 360 components

React 360 provides a range of components that can be used out of the box to create the VR user interface for the game. This range consists of basic components available from React Native and also VR-specific components that will allow you to incorporate the interactive 3D objects in the VR game. In the following sections, we will summarize the specific components that will be used to build the game view and features, including core components, such as `View` and `Text`, and VR-specific components, such as `Entity` and `VrButton`.

Core components

The core components in React 360 include two of React Native's built-in components – the `Text` and `View` components. In the game, we will use these two components to add content to the game world. In the following sections, we will discuss these two core components.

View

The `View` component is the most fundamental component for building a user interface in React Native, and it maps directly to the native view equivalent on whatever platform React Native is running on. In our case, since the application will render in the browser, it will map to a `<div>` element in the browser. The `View` component can be added as follows:

```
|<View>
|  <Text>Hello</Text>
|</View>
```

The `View` component is typically used as a container for other components; it can be nested inside other views and can have zero-to-many children of any type.

We will use `View` components to hold the game world view and add 3D object entities and text to the game. Next, we will look at the `Text` component, which will allow us to add text to the view.

Text

The `Text` component is a React Native component for displaying text, and we will use it to render strings in a 3D space by placing `Text` components inside `View` components, as shown in the following code:

```
| <View>
|   <Text>Welcome to the MERN VR Game</Text>
| </View>
```

We will compose the game world using these two React Native components, along with other React 360 components to integrate VR features into the game. In the next section, we will go over the React 360 components that will let us add interactive VR objects in the game world.

Components for the 3D VR experience

React 360 provides a set of its own components to create the VR experience. Specifically, we will use the `Entity` component to add 3D objects and a `VrButton` component to capture clicks from the user. We will discuss the `Entity` and `VrButton` components in the following sections.

Entity

In order to add 3D objects to the game world, we will use the `Entity` component, which allows us to render 3D objects in React 360. The `Entity` component can be added in the view as follows:

```
<Entity
  source={{
    obj: {uri: "http://linktoOBJfile.obj"},
    mtl: {uri: "http://linktoMTLfile.obj"}
  }}
/>
```

Files containing the specific 3D object's information are added to the `Entity` component using a `source` attribute. The `source` attribute takes an object of key-value pairs to map resource file types to their locations. React 360 supports the Wavefront OBJ file format, which is a common representation for 3D models. So, in the `source` attribute, the `Entity` component supports the following keys:

- `obj`: The location of an OBJ-formatted model
- `mtl`: The location of an MTL-formatted material (the companion to OBJ)

The values for the `obj` and `mtl` properties point to the location of these files and can be static strings, `asset()` calls, `require()` statements, or URI strings.

 *OBJ (or .OBJ) is a geometry definition file format that was first developed by Wavefront Technologies. It is a simple data format that represents 3D geometry as a list of vertices and texture vertices. OBJ coordinates have no units, but OBJ files can contain scale information in a human-readable comment line. You can learn more about this format at paulbourke.net/dataformats/obj/.*

 *MTL (or .MTL) are material library files that contains one or more material definitions, each of which includes the color, texture, and reflection map of individual materials. These are applied to the surfaces and vertices of objects. You can learn more about this format at paulbourke.net/dataformats/mtl/.*

The `Entity` component also takes `transform` property values in the `style` attribute, so the objects can be placed at the desired positions and

orientations in the 3D world space.

In our MERN VR Game application, makers will add URLs pointing to the VR object files (both `.obj` and `.mtl`) for each of their `Entity` objects in a game, and also specify the `transform` property values to indicate where and how the 3D objects should be placed in the game world.



A good source of 3D objects is <https://clara.io/>, with multiple file formats available for download and use.

The `Entity` component will render 3D objects in the 3D world space. In order to make these objects interactive, we need to use the `VrButton` component, which is discussed in the next section.

VrButton

The `VrButton` component in React 360 will help us to implement a simple, button-style `onClick` behavior for the objects and `Text` buttons that will be added to the game. A `VrButton` component is not visible in the view by default and will only act as a wrapper to capture events, but it can be styled in the same way as a `View` component, as shown in the following code:

```
<VrButton onClick={this.clickHandler}>
  <View>
    <Text>Click me to make something happen!</Text>
  </View>
</VrButton>
```

This component is a helper for managing click-type interactions from the user across different input devices. Input events that will trigger the click event include a spacebar press on the keyboard, a left-click on the mouse, and a touch on the screen.

The `Entity` and `VrButton` components from React 360 will allow us to render interactive 3D objects in the game world. To integrate other VR functionalities such as setting the background scene and playing audio in the game world, we will explore relevant options from the React 360 API in the next section.

The React 360 API

Besides the React 360 components discussed in the previous section, we will utilize the APIs provided by React 360 to implement functionality such as setting the background scene, playing audio, dealing with external links, adding styles, capturing the current orientation of the user's view, and using static asset files. In the following sections, we will explore the `Environment` API, the `Audio` and `Location` native modules, the `StyleSheet` API, the `VrHeadModel` module, and the asset specification options.

Environment

In the game, we will set the world or background scene with equirectangular panoramic images. We will use the `Environment` API from React 360 to change this background scene dynamically from the React code using its `setBackgroundImage` method. This method can be used as follows:

```
| Environment.setBackgroundImage( {uri: 'http://linktopanoramainimage.jpg'} )
```

This method sets the current background image with the resource at the specified URL. When we integrate the React 360 game code with the MERN stack containing the game application backend, we can use this to set the game world image dynamically using image links provided by the user. In the next section, we will explore native modules that will allow us to play audio in this rendered scene in the browser, and provide access to the browser location.

Native modules

Native modules in React 360 provide us with the ability to access functionality that is only available in the main browser environment. In the game, we will use `AudioModule` in `NativeModules`, to play sounds in response to user activity, and the `Location` module, to give us access to `window.location` in the browser to handle external links. These modules can be accessed in `index.js`, as follows:

```
import {  
  ...  
  NativeModules  
} from 'react-360'  
  
const { AudioModule, Location } = NativeModules
```

We can use these imported modules in the code to manipulate the audio and location URL in the browser. In the following sections, we will explore how these modules can be used to implement the features of the game.

AudioModule

When the user interacts with the 3D objects in the game, we will play sounds based on whether the object can be collected or not, and also whether the game has been completed. The `AudioModule` in `NativeModules` allows us to add sound to the VR world as background environmental audio, one-off sound effects, and spatial audio. In our game, we will use environmental audio and one-off sound effects:

- **Environmental audio:** To play audio on loop and set the mood when the game is successfully completed, we will use the `playEnvironmental` method, which takes an audio file path as the `source` attribute, and the `loop` option as a `playback` parameter, as shown in the following code:

```
AudioModule.playEnvironmental({  
    source: asset('happy-bot.mp3'),  
    loop: true  
})
```

- **Sound effects:** To play a single sound once when the user clicks on 3D objects, we will use the `playOneShot` method that takes an audio file path as the `source` attribute, as shown in the following code:

```
AudioModule.playOneShot({  
    source: asset('clog-up.mp3'),  
})
```

The `source` attribute in the options passed to `playEnvironmental` and `playOneShot` takes a resource file location to load the audio. It can be an `asset()` statement or a resource URL declaration in the form of `{uri: 'PATH'}`.

We will call these `AudioModule` methods to play specified audio files as needed from the game implementation code. In the next section, we

will look at how we can use the `Location` module, which is another native module in React 360.

Location

After we integrate the React 360 code containing the game with the MERN stack containing the game application backend, the VR game will be launched from the MERN server at a declared route containing the specific game's ID. Then, once a user completes a game, they will also have the option to leave the VR space and go to a URL containing a list of other games. To handle these incoming and outgoing app links in the React 360 code, we will utilize the `Location` module in `NativeModules`.

The `Location` module is essentially the `Location` object returned by the read-only `window.location` property in the browser. We will use the `replace` method and the `search` property in the `Location` object to implement features related to external links. We will handle outgoing and incoming links as follows:

- **Handling outgoing links:** When we want to direct the user out of the VR application to another link, we can use the `replace` method in `Location`, as shown in the following code:

```
|     Location.replace(url)
```

- **Handling incoming links:** When the React 360 app is launched from an external URL and after the registered component mounts, we can access the URL and retrieve its query string part using the `search` property in `Location`, as shown in the following code.

```
|     componentDidMount = () => {
|       let queryString = Location.search
|       let gameId = queryString.split('?id=')[1]
|     }
```

For the purpose of integrating this React 360 component with MERN VR Game, and dynamically loading game details, we will capture this

initial URL to parse the game ID from a query parameter and then use it to make a read API call to the MERN application server. This implementation is elaborated further in [Chapter 11](#), *Making the VR Game Dynamic Using MERN*.

Besides using these native modules from the React 360 API, we will also use the StyleSheet API to add styling to the components rendered to make the game view. We will demonstrate how to use the StyleSheet API in the next section.

StyleSheet

The StyleSheet API from React Native can also be used in React 360 to define several styles in one place rather than adding styles to individual components. The styles can be defined using StyleSheet, as shown in the following code:

```
const styles = StyleSheet.create({
  subView: {
    width: 10,
    borderColor: '#d6d7da',
  },
  text: {
    fontSize: '1em',
    fontWeight: 'bold',
  }
})
```

These style objects defined using `StyleSheet.create` can be added to components as required, as shown in the following code:

```
<View style={styles.subView}>
  <Text style={styles.text}>hello</Text>
</View>
```

This will apply the CSS styles to the `View` and `Text` components accordingly.



The default distance units for CSS properties, such as width and height, are in meters when mapping to 3D space in React 360, whereas the default distance units are in pixels for 2D interfaces in React Native.

We will use StyleSheet in this way to define styles for the components that will make up the game view. In the next section, we will discuss the `VrHeadModel` module in React 360, which will allow us to figure out where the user is currently looking.

VrHeadModel

`vrHeadModel` is a utility module in React 360 that simplifies the process of obtaining the current orientation of the headset. Since the user is moving around in a VR space, when the desired feature requires that an object or piece of text should be placed in front of or with respect to the user's current orientation, it becomes necessary to know exactly where the user is currently gazing.

In MERN VR Game, we will use this to show the game completed message to the user in front of their view, no matter where they end up turning to from their initial position. For example, the user may be looking up or down when collecting the final object, and the completed message should pop up wherever the user is gazing.

To implement this, we will retrieve the current head matrix as an array of numbers using `getHeadMatrix()` from `VrHeadModel`, and set it as a value for the `transform` property in the `style` attribute of the `view` component containing the game completed message.

This will render the message at the location where the user is currently gazing. We will see the usage of this `getHeadMatrix()` function later in the chapter, in the *Building the game view in React 360* section. In the next section, we will discuss how static assets can be loaded in React 360.

Loading assets

In order to load any static asset files such as image or audio files in the code, we can utilize the `asset` method in React 360. This `asset()` functionality in React 360 allows us to retrieve external resource files, including audio and image files.

For example, we will place the sound audio files for the game in the `static_assets` folder, to be retrieved using `asset()` for each audio added to the game, as shown in the following code:

```
AudioModule.playOneShot({  
    source: asset('collect.mp3'),  
})
```

This will load the audio file to be played in the call to `playOneShot`.

With these different APIs and modules available in React 360, we will integrate different features for the VR game such as setting the background scene, playing audio, adding styles, loading static files, and retrieving the user orientation. In the next section, we will look at some of the input events available in React 360 that will allow us to make the game interactive.

React 360 input events

In order to make the game interface interactive, we will utilize some of the input event handlers exposed in React 360. Input events are collected from mouse, keyboard, touch, and gamepad interactions, and also with the `gaze` button click on a VR headset.

The specific input events we will work with are the `onEnter`, `onExit`, and `onClick` events, as discussed in the following list:

- `onEnter`: This event is fired whenever the platform cursor begins intersecting with a component. We will capture this event for the VR objects in the game, so the objects can start rotating around the `y` axis when the platform cursor enters the specific object.
- `onExit`: This event is fired whenever the platform cursor stops intersecting with a component. It has the same properties as the `onEnter` event and we will use it to stop rotating the VR object just exited.
- `onClick`: The `onClick` event is used with the `VrButton` component, and is fired when there is click interaction with `VrButton`. We will use this to set click event handlers on the VR objects, and also on the game complete message to redirect the user out of the VR application to a link containing a list of games.

These events will allow us to add actions to the game, which happens when the user does something.

While implementing the VR game, we will apply 3D world concepts to determine how to set the game world with equirectangular panoramic images, and position VR objects in this world based on the 3D coordinate system. We will use React 360 components such as `View`, `Text`, `Entity`, and `VrButton` to render the VR game view. We can also use available React 360 APIs to load audio and external URLs for the VR game in the browser environment. Finally, we can utilize

available React 360 events that capture user interactions to make the VR game interactive. With the VR-related concepts, React 360 components, APIs, modules, and events discussed in this section, we are ready to define the specific game data details before we start implementing the complete VR game using these concepts. In the next section, we will go over the game data structure and details.

Defining game details

Each game in MERN VR Game will be defined in a common data structure that the React 360 application will also adhere to when rendering the individual game details. In the following sections, we will discuss the data structure for capturing a game's details, and then highlight the difference between using static game data and dynamically loaded game data.

Game data structure

The game data will consist of details such as the game's name, a URL pointing to the location of the equirectangular image for the game world, and two arrays containing details for each VR object to be added to the game world. The following list indicates the fields corresponding to the game data attributes:

- `name`: A string representing the name of the game
- `world`: A string with the URL pointing to the equirectangular image either hosted on cloud storage, CDNs, or stored on MongoDB
- `answerObjects`: An array of JavaScript objects containing details of the VR objects that can be collected by the player
- `wrongObjects`: An array of JavaScript objects containing details of the other VR objects to be placed in the VR world that cannot be collected by the player

These details will define each game in the MERN VR Game application. The arrays containing the VR object details will store properties of each object to be added to the 3D world in the game. In the following section, we will go over the details representing a VR object in these arrays.

Details of VR objects

The two arrays in the game data structure will store details of the VR objects to be added in the game world. The `answerObjects` array will contain details of the 3D objects that can be collected, and the `wrongObjects` array will contain details of 3D objects that cannot be collected. Each object will contain links to the 3D data resource files and `transform` style property values. In the following list, we will go over these specific details to be stored for each object:

- **OBJ and MTL links:** The 3D data information resources for the VR objects will be added to the `objUrl` and `mtlUrl` attributes. These attributes will contain the following values:
 - `objUrl`: The link to the `.obj` file for the 3D object
 - `mtlUrl`: The link to the accompanying `.mtl` file

The `objUrl` and `mtlUrl` links may point to files either hosted on cloud storage, CDNs, or stored on MongoDB. For MERN VR Game, we will assume that makers will add URLs to their own hosted OBJ, MTL, and equirectangular image files.

- **Translation values:** The position of the VR object in the 3D space will be defined with the `translate` values in the following attributes:
 - `translateX`: The translation value of the object along the x axis
 - `translateY`: The translation value of the object along the y axis
 - `translateZ`: The translation value of the object along the z axis

All translation values are numbers in meters.

- **Rotation values:** The orientation of the 3D object will be defined with the `rotate` values in the following keys:
 - `rotateX`: The rotation value of the object around the x axis; in other words, turning the object up or down

- `rotateY`: The rotation value of the object around the `y` axis that would turn the object left or right
- `rotateZ`: The rotation value of the object around the `z` axis, making the object tilt forward or backward

All rotation values are in numbers or string representations of a number in degrees.

- **Scale value**: The `scale` value will define the relative size and appearance of the 3D object in the 3D environment:
 - `scale`: A number value that defines uniform scale across all axes
- **Color**: If the 3D object's material texture is not provided in an MTL file, a color value can be defined to set the default color of the object in the `color` attribute:
 - `color`: A string value representing color values allowed in CSS

These attributes will define the details of each VR object to be added to the game.

With this game data structure capable of holding the details of the game and its VR objects, we can implement the game in React 360 accordingly with sample data values. In the next section, we will look at sample game data and distinguish between setting game data statically in contrast to loading it dynamically for different games.

Static data versus dynamic data

While integrating the game developed using React 360 with the MERN-based application in the next chapter, we will update the React 360 code to fetch game data dynamically from the backend database. This will render the React 360 game view with different games stored in the database. For now, we will start developing the game features here with dummy game data that is set in component state. The sample game data will be set as follows, using the defined game data structure:

```
game: {
  name: 'Space Exploration',
  world: 'https://s3.amazonaws.com/mernbook/vrGame/milkyway.jpg',
  answerObjects: [
    {
      objUrl: 'https://s3.amazonaws.com/mernbook/vrGame/planet.obj',
      mtlUrl: 'https://s3.amazonaws.com/mernbook/vrGame/planet.mtl',
      translateX: -50,
      translateY: 0,
      translateZ: 30,
      rotateX: 0,
      rotateY: 0,
      rotateZ: 0,
      scale: 7,
      color: 'white'
    }
  ],
  wrongObjects: [
    {
      objUrl: 'https://s3.amazonaws.com/mernbook/vrGame/tardis.obj',
      mtlUrl: 'https://s3.amazonaws.com/mernbook/vrGame/tardis.mtl',
      translateX: 0,
      translateY: 0,
      translateZ: 90,
      rotateX: 0,
      rotateY: 20,
      rotateZ: 0,
      scale: 1,
      color: 'white'
    }
  ]
}
```

This game object holds the details of a sample game including the name, a link to the 360 world image, and two object arrays with one

3D object detailed in each array. For initial development purposes, this sample game data can be set in state to be rendered in the game view. Using this game structure and data, in the next section, we will implement the game features in React 360.

Building the game view in React 360

We will apply the React 360 concepts and use the game data structure to implement the game features for each game in the MERN VR Game application. For these implementations, we will update the default starter code generated in the `index.js` and `client.js` files within the initiated React 360 project.

For a working version of the game, we will start with the `MERNVR` component's state that was initialized using the sample game data from the previous section.

The `MERNVR` component is defined in `index.js`, and the code will be updated with the game data in state, as shown in the following code:

/MERNVR/index.js

```
export default class MERNVR extends React.Component {  
  constructor() {  
    super()  
    this.state = {  
      game: sampleGameData  
      ...  
    }  
  }  
  ...  
}
```

This will make the sample game's details available for building the rest of the game features. In the following sections, we will update the code in the `index.js` and `client.js` files to first mount the game world, define the CSS styles, and load the 360-degree environment for the game. Then, we will add the 3D VR objects to the game, make these objects interactive, and implement behavior that indicates the game is completed.

Updating client.js and mounting to Location

The default code in `client.js` attaches the mount point declared in `index.js` to the default `Surface` in the React 360 app, where `Surface` is a cylindrical layer that is used for placing a 2D user interface. In order to use the 3D meter-based coordinate system for a layout in 3D space, we need to mount to a `Location` object instead of a `Surface`. So, we will update `client.js` to replace the `renderToSurface` with a `renderToLocation`, as highlighted in the following code:

/MERNVR/client.js

```
r360.renderToLocation(  
  r360.createRoot('MERNVR', { /* initial props */ }),  
  r360.getDefaultLocation()  
)
```

This will mount our game view to a React 360 `Location`.



You can also customize the initial background scene by updating the `r360.compositor.setBackground(r360.getAssetURL('360_world.jpg'))` code in `client.js` to use your desired image.

With this update added in `client.js`, we can move on to updating the code in `index.js`, which will contain our game functionalities. In the next section, we will start by defining CSS styles for the elements to be rendered in the game view.

Defining styles with StyleSheet

In `index.js`, we will update the default styles generated in the initial React 360 project to add our own CSS rules. In the `StyleSheet.create` call, we will define style objects to be used with the components in the game, as shown in the following code:

/MERNVR/index.js

```
const styles = StyleSheet.create({
    completeMessage: {
        margin: 0.1,
        height: 1.5,
        backgroundColor: 'green',
        transform: [ {translate: [0, 0, -5]} ]
    },
    congratsText: {
        fontSize: 0.5,
        textAlign: 'center',
        marginTop: 0.2
    },
    collectedText: {
        fontSize: 0.2,
        textAlign: 'center'
    },
    button: {
        margin: 0.1,
        height: 0.5,
        backgroundColor: 'blue',
        transform: [ { translate: [0, 0, -5] } ]
    },
    buttonText: {
        fontSize: 0.3,
        textAlign: 'center'
    }
})
```

For the game features implemented in this book, we are keeping the styling simple with CSS declared for only the text and button to be displayed when the game is completed. In the next section, we will look at how to load the 360 panoramic image that will represent the 3D world for each game.

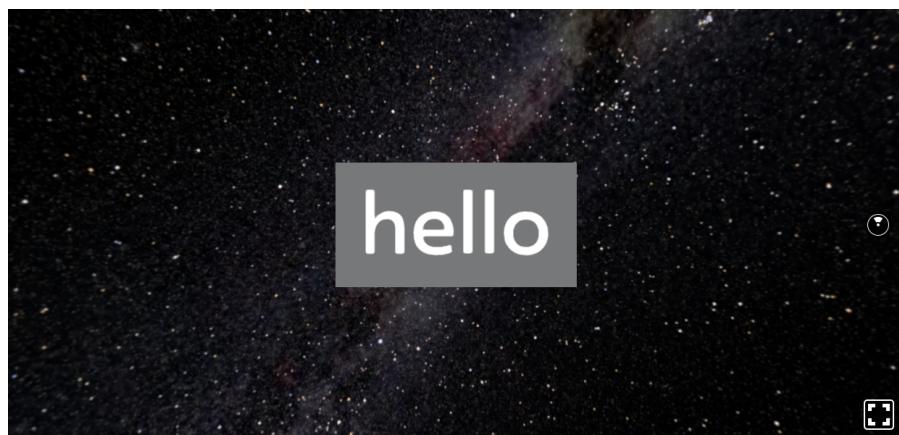
World background

In order to set the game's 360-degree world background, we will update the current background scene using the `setBackgroundImage` method from the `Environment` API. We will call this inside the `componentDidMount` of the `MERNVR` component defined in `index.js`, as shown in the following code:

/MERNVR/index.js

```
| componentDidMount = () => {
|   Environment.setBackgroundImage(
|     {uri: this.state.game.world}
|   )
| }
```

This will replace the default 360-degree background in the starter React 360 project with our sample game's world image fetched from cloud storage. If you are editing the default React 360 application and have it running, refreshing the <http://localhost:8081/index.html> link in the browser should show an outer space background, which you can pan across using the mouse:



To generate the preceding screenshot, the `View` and `Text` components in the default code were also updated with custom CSS rules to show this hello text on the screen.

With this, we will have a 360-degree game world that can be explored by the user. In the next section, we will explore how to place 3D objects in this world.

Adding 3D VR objects

We will add 3D objects to the game world using `Entity` components from React 360, along with the sample object details in the `answerObjects` and `wrongObjects` arrays that were defined for the game.

First, we will concatenate the `answerObjects` and `wrongObjects` arrays in `componentDidMount` to form a single array containing all of the VR objects, as shown in the following code:

/MERNVR/index.js

```
componentDidMount = () => {
  let vrObjects =
    this.state.game.answerObjects.concat(this.state.game.wrongObjects)
  this.setState({vrObjects: vrObjects})
  ...
}
```

This will give us a single array containing all of the VR objects for the game. Then, in the main view, we will iterate over this merged `vrObjects` array to render the `Entity` components with details of each object. The iteration code will be added using `map`, as shown in the following code:

/MERNVR/index.js

```
{this.state.vrObjects.map((vrObject, i) => {
  return (
    <Entity key={i} style={this.setModelStyles(vrObject, i)}
      source={{
        obj: {uri: vrObject.objUrl},
        mtl: {uri: vrObject.mtlUrl}
      }>
    )
  )
})}
```

The `obj` and `mtl` file links are added to the `source` prop in `Entity`, and the `transform` style details are applied in the `Entity` component's styles with

the call to `setModelStyles`. The `setModelStyles` method constructs the styles for the specific VR object to be rendered, using values defined in the VR object's details.

The `setModelStyles` method is implemented as follows:

/MERNVR/index.js

```
setModelStyles = (vrObject, index) => {
  return {
    display: this.state.collectedList[index] ? 'none' : 'flex',
    color: vrObject.color,
    transform: [
      {
        translateX: vrObject.translateX
      },
      {
        translateY: vrObject.translateY
      },
      {
        translateZ: vrObject.translateZ
      },
      {
        scale: vrObject.scale
      },
      {
        rotateY: vrObject.rotateY
      },
      {
        rotateX: vrObject.rotateX
      },
      {
        rotateZ: vrObject.rotateZ
      }
    ]
  }
}
```

The `display` property will allow us to show or hide an object based on whether it has already been collected by the player or not. The `translate` and `rotate` values will render the 3D objects in the desired positions and orientations across the VR world. Next, we will update the `Entity` code further to enable user interactions with these 3D objects.

Interacting with VR objects

In order to make the VR game objects interactive, we will use the React 360 event handlers, such as `onEnter` and `onExit` with `Entity`, and `onClick` with `VrButton`, to add rotation animation and gameplay behavior. In the following sections, we will add the implementations for rotating a VR object when a user focuses on it, and for adding click behavior on the objects to allow a user to collect the correct objects in the game.

Rotating a VR object

We want to add a feature that starts rotating a 3D object around its y axis whenever a player focuses on the 3D object, that is, when the platform cursor begins intersecting with the `Entity` component rendering the specific 3D object.

We will update the `Entity` component from the previous section to add the `onEnter` and `onExit` handlers, as shown in the following code:

/MERNVR/index.js

```
| <Entity  
|   ...  
|   onEnter={this.rotate(i)}  
|   onExit={this.stopRotate}  
| />
```

The object rendered with this `Entity` component will start rotating on a cursor entry or focus on the object, and it will stop when the platform cursor exits the object and is no longer in the player's focus. In the following section, we will discuss the implementation of this rotation animation.

Animation with requestAnimationFrame

The rotation behavior for each 3D object is implemented in the event handlers added to the `Entity` component, which is rendering the 3D object. Specifically, in the `rotate(index)` and `stopRotate()` handler methods that are called when the `onEnter` and `onExit` events occur, we will implement rotation animation behavior using `requestAnimationFrame` for smooth animations in the browser.



The `window.requestAnimationFrame()` method asks the browser to call a specified callback function to update an animation before the next repaint. With `requestAnimationFrame`, the browser optimizes the animations to make them smoother and more resource-efficient.

Using the `rotate` method, we will update the `rotateY` transform value of the given object at a steady rate on a set time interval with `requestAnimationFrame`, as shown in the following code:

/MERNVR/index.js

```
this.lastUpdate = Date.now()
rotate = index => event => {
  const now = Date.now()
  const diff = now - this.lastUpdate
  const vrObjects = this.state.vrObjects
  vrObjects[index].rotateY = vrObjects[index].rotateY + diff / 200
  this.lastUpdate = now
  this.setState({vrObjects: vrObjects})
  this.requestID = requestAnimationFrame(this.rotate(index))
}
```

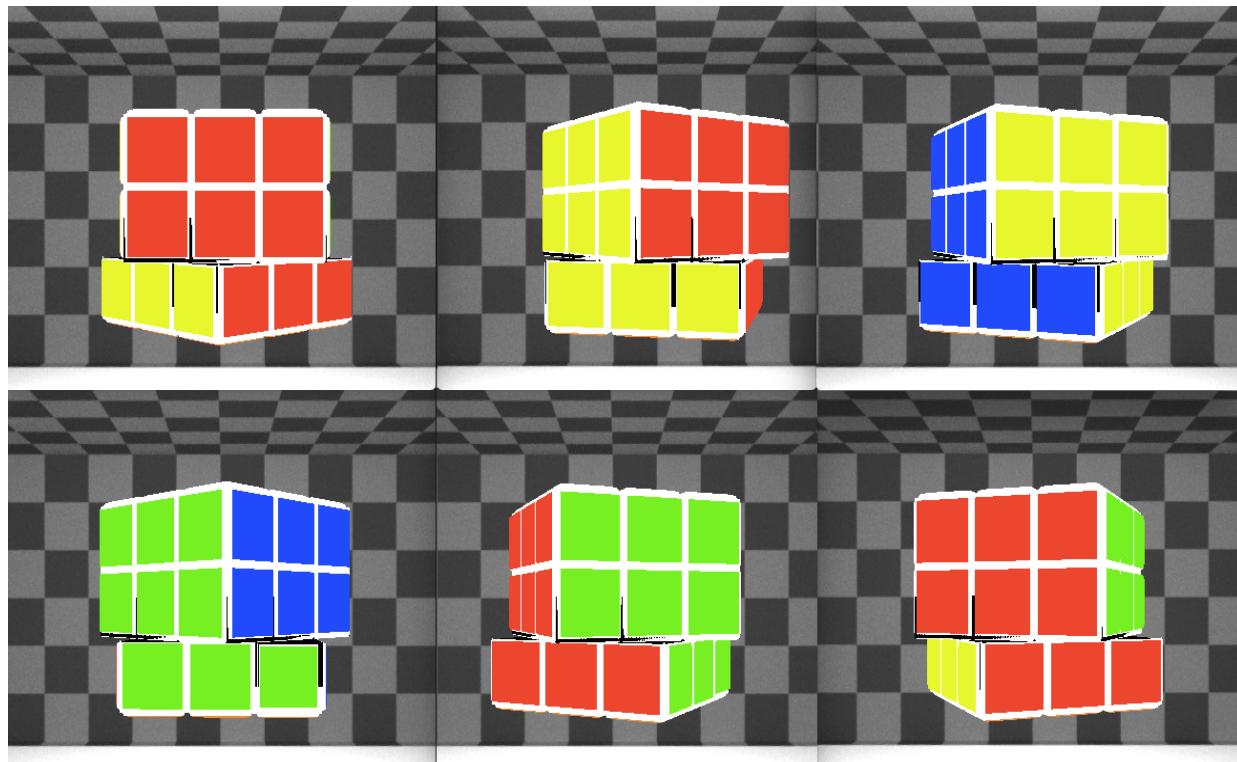
The `requestAnimationFrame` method will take the `rotate` method as a recursive callback function, then execute it to redraw each frame of the rotation animation with the new values, and, in turn, update the animation on the screen.

The `requestAnimationFrame` method returns a `requestID`, which we will use in the call to `stopRotate`, so the animation gets canceled in the `stopRotate` method. This `stopRotate` method is defined as follows:

/MERNVR/index.js

```
stopRotate = () => {
  if (this.requestID) {
    cancelAnimationFrame(this.requestID)
    this.requestID = null
  }
}
```

This will implement the functionality of animating the 3D object only when it is in the viewer's focus. As shown in the following screenshot, the 3D Rubik's cube rotates clockwise around its y axis while it is in focus:



Though not covered here, it is also worth exploring the React 360 Animated library, which can be used to compose different types of animations. Core components can be animated natively with this library, and it is possible to animate other components using `createAnimatedComponent()`. This library was originally implemented from React Native; to learn more, you can refer to the React Native documentation.

Now the users playing the game will observe motion when they focus on any of the VR objects placed in the game world. In the next section, we will add the functionality for capturing user clicks on these objects.

Clicking on the 3D objects

In order to register the click behavior on each 3D object added to the game, we need to wrap the `Entity` component with a `VrButton` component that can call the `onClick` handler.

We will update the `Entity` component added inside the `vrObjects` array iteration code, to wrap it with the `VrButton` component, as shown in the following code:

/MERNVR/index.js

```
| <VrButton onClick={this.collectItem(vrObject)} key={i}>
|   <Entity ... />
| </VrButton>
```

The `VrButton` component will call the `collectItem` method when clicked on, and pass it the current object's details.

When a 3D object is clicked on by a user, we need the `collectItem` method to perform the following actions with respect to the game features:

- Check whether the clicked object is an `answerObject` or a `wrongObject`.
- Based on the object type, play the associated sound.
- If the object is an `answerObject`, it should be collected and removed from view, then added to a list of collected objects.
- Check whether all instances of `answerObject` were successfully collected with this click:
 - If yes, show the game completed message to the player and play the sound for game completed.

We will implement these actions in the `collectItem` method with the following structure and steps:

```
| collectItem = vrObject => event => {
|   if (vrObject is an answerObject) {
```

```
    ... update collected list ...
    ... play sound for correct object collected ...
    if (all answer objects collected) {
        ... show game completed message in front of user ...
        ... play sound for game completed ...
    }
} else {
    ... play sound for wrong object clicked ...
}
}
```

Any time a VR object is clicked on by the user, in this method, we will first check the type of the object before taking the related actions. We will discuss the implementation of these steps and actions in detail in the following section.

Collecting the correct object on click

When a user clicks on a 3D object, we need to first check whether the clicked object is an answer object. If it is, this object will be *collected* and hidden from view, and a list of collected objects will be updated along with the total number to keep track of the user's progress in the game.

To check whether the clicked VR object is an `answerObject`, we will use the `indexOf` method to find a match in the `answerObjects` array, as shown in the following code:

```
| let match = this.state.game.answerObjects.indexOf(vrObject)
```

If the `vrObject` is an `answerObject`, `indexOf` will return the array index of the matched object; otherwise, it will return `-1` if no match is found.

To keep track of collected objects in the game, we will also maintain an array of Boolean values in `collectedList` at corresponding indices, and the total number of objects collected so far in `collectedNum`, as shown in the following code:

```
let updateCollectedList = this.state.collectedList
let updateCollectedNum = this.state.collectedNum + 1
updateCollectedList[match] = true
this.setState({collectedList: updateCollectedList,
              collectedNum: updateCollectedNum})
```

Using the `collectedList` array, we will also determine which `Entity` component should be hidden from the view because the associated object was collected. The `display` style property of the relevant `Entity` component will be set based on the Boolean value of the corresponding index in the `collectedList` array. We set this in the `style` for the `Entity` component using the `setModelStyles` method, as shown

earlier in the *Adding 3D VR objects* section. This display style value is set conditionally with the following line of code:

```
| display: this.state.collectedList[index] ? 'none' : 'flex'
```

Depending on whether the array index of the rendered VR object is set to true in the collected list of objects, we hide or show the `Entity` component in the view.

For example, in the following screenshot, the treasure chest can be clicked on to be collected as it is an `answerObject`, whereas the flower pot cannot be collected because it is a `wrongObject`:



When the treasure chest is clicked on, it disappears from the view as the `collectedList` is updated, and we also play the sound effect for collection using `AudioModule.playOneShot` with the following code:

```
    | AudioModule.playOneShot({  
    |     source: asset('collect.mp3'),  
    | })
```

However, when the flower pot is clicked on, and it is identified as a wrong object, we play another sound effect indicating it cannot be collected, as shown in the following code:

```
| AudioModule.playOneShot({  
|   source: asset('clog-up.mp3'),  
| })
```

As the flower pot was identified to be a wrong object, the `collectedList` was not updated and it remains on the screen, whereas the treasure chest is gone, as shown in the following screenshot:



The complete code in the `collectItem` method that executes all of these steps when an object is clicked on will be as follows.

/MERNVR/index.js:

```
collectItem = vrObject => event => {  
  let match = this.state.game.answerObjects.indexOf(vrObject)  
  if (match != -1) {  
    let updateCollectedList = this.state.collectedList  
    let updateCollectedNum = this.state.collectedNum + 1  
    updateCollectedList[match] = true  
    this.checkGameCompleteStatus(updateCollectedNum)  
    AudioModule.playOneShot({  
      source: asset('collect.mp3'),  
    })  
    this.setState({collectedList: updateCollectedList,  
      collectedNum: updateCollectedNum})  
  } else {  
    AudioModule.playOneShot({  
      source: asset('clog-up.mp3'),  
    })  
  }  
}
```

After a clicked object is collected using this method, we will also check whether all of the `answerObjects` have been collected and whether the game is complete with a call to the `checkGameCompleteStatus` method. We will take a look at the implementation of this method and the game completed functionality in the next section.

Game completed state

Every time an `answerObject` is collected, we will check whether the total number of collected items is equal to the total number of objects in the `answerObjects` array to determine whether the game is complete. We will achieve this by calling the `checkGameCompleteStatus` method, which will perform this check, as shown in the following code:

/MERNVR/index.js

```
checkGameCompleteStatus = (collectedTotal) => {
  if (collectedTotal == this.state.game.answerObjects.length) {
    AudioModule.playEnvironmental({
      source: asset('happy-bot.mp3'),
      loop: true
    })
    this.setState({hide: 'flex', hmMatrix: VrHeadModel.getHeadMatrix()})
  }
}
```

In this method, we first confirm that the game is indeed complete, and then we perform the following actions:

- Play the audio for game completed, using `AudioModule.playEnvironmental`.
- Fetch the current `headMatrix` value using `VrHeadModel` so that it can be set as the transform matrix value for the `view` component containing the game completion message.
- Set the `display` style property of the `view` message to `flex`, so the message renders to the viewer.

The `view` component containing the message congratulating the player for completing the game will be added to the parent `view` component as follows:

/MERNVR/index.js

```
<View style={this.setGameCompletedStyle}>
  <View style={this.styles.completeMessage}>
    <Text style={this.styles.congratsText}>Congratulations!</Text>
```

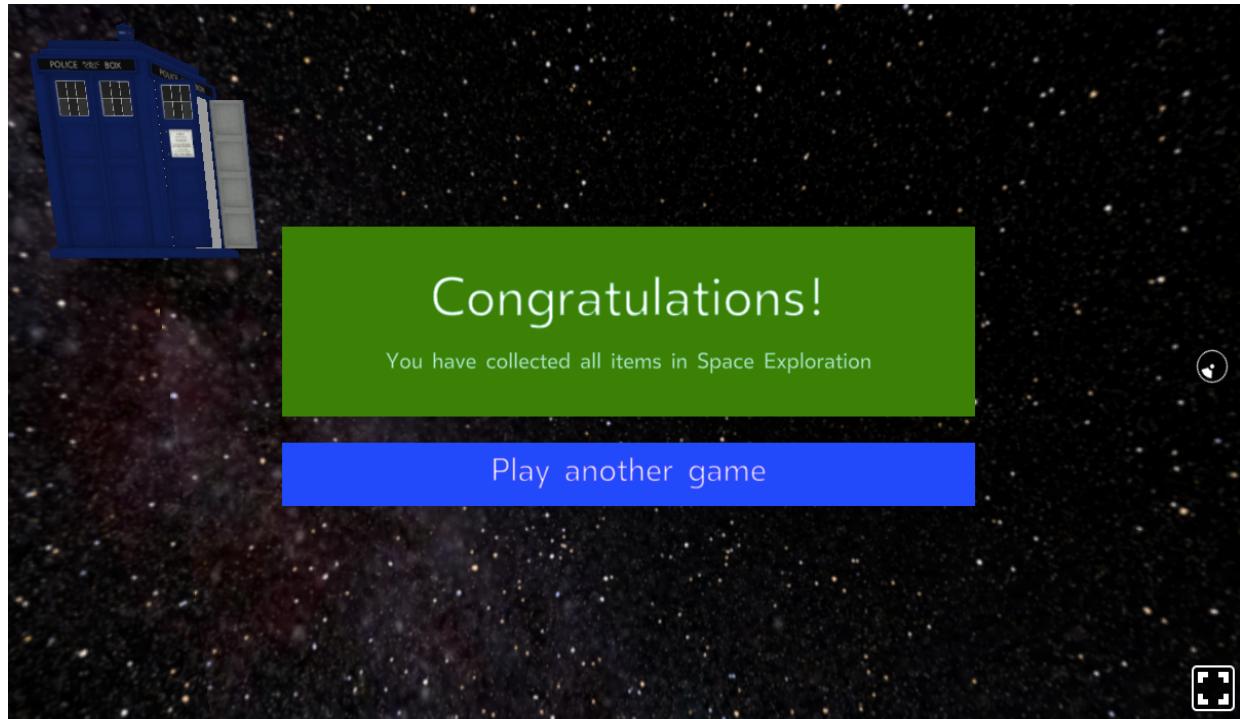
```
<Text style={this.styles.collectedText}>
    You have collected all items in {this.state.game.name}
</Text>
</View>
<VrButton onClick={this.exitGame}>
    <View style={this.styles.button}>
        <Text style={this.styles.buttonText}>Play another game</Text>
    </View>
</VrButton>
</View>
```

The call to the `setGameCompletedStyle()` method will set the styles for the `view` message with the updated `display` value and the `transform` matrix value. The `setGameCompletedStyle` method is defined as follows:

/MERNVR/index.js

```
setGameCompletedStyle = () => {
    return {
        position: 'absolute',
        display: this.state.hide,
        layoutOrigin: [0.5, 0.5],
        width: 6,
        transform: [{translate: [0, 0, 0]}, {matrix: this.state.hmMatrix}]
    }
}
```

These style values will render the `view` component with the completion message at the center of the user's current view, regardless of whether they are looking up, down, behind, or forward in the 360-degree VR world, as shown in the following screenshot:



The final text in the `view` message will act as a button, as we wrapped this `view` in a `VrButton` component that calls the `exitGame` method when clicked. The `exitGame` method is defined as follows:

/MERNVR/index.js

```
| exitGame = () => {
|   Location.replace('/')
| }
```

The `exitGame` method will use the `Location.replace` method to redirect the user to an external URL that may contain a list of games.

The `replace` method can be passed any valid URL, and once this React 360 game code is integrated with the MERN VR Game application in [Chapter 14, Making the VR Game Dynamic Using MERN](#), `replace('/')` will take the user to the home page of the whole application.

The VR game functionalities are complete with these updates to the React 360 project. It is now possible to set a 360-degree panoramic background as the game world and add interactive VR objects to this world. These 3D objects will rotate in place and can be collected

based on user interaction if the game rules allow it. In the next section, we will demonstrate how to bundle this React 360 code so that the game can be integrated with a MERN-based web application.

Bundling for production and integration with MERN

Now that we have features of the VR game implemented and are functional with the sample game data, we can prepare it for production and add it to our MERN-based application to see how VR can be added to an existing web application. In the following sections, we will look at how to bundle the React 360 code, integrate it with a MERN application, and test the integration by running the game from the application.

Bundling React 360 files

React 360 tools provide a script to bundle all of the React 360 application code into a few files that we can just place on the MERN web server and serve as content at a specified route. To create the bundled files, we can run the following command in the React 360 project directory:

```
| yarn bundle
```

This generates compiled versions of the React 360 application files in a folder called `build`. The compiled bundle files are `client.bundle.js` and `index.bundle.js`. These two files, in addition to the `index.html` file and the `static-assets/` folder, make up the production version of the whole React 360 application that was developed. The final production code will be in the following folder and files:

```
-- static_assets/  
-- index.html  
-- index.bundle.js  
-- client.bundle.js
```

We will have to take these folders and files over to a MERN project directory to integrate the game with the MERN application, as discussed in the next section.

Integrating with a MERN application

In order to integrate the game developed in React 360 with a MERN-based web application, we will first bring the React 360 production files discussed in the previous section to our MERN application project. Then, we will update the bundle file references in the generated `index.html` code to point to the new location of the bundle files, before loading the `index.html` code at a specified route in the Express app.

Adding the React 360 production files

With consideration to the folder structure in the existing MERN skeleton application, we will add the `static_assets` folder and the bundle files from the React 360 production files to the `dist/` folder. This will keep our MERN code organized with all the bundles in the same location. The `index.html` file will be placed in a new folder, named `vr` in the `server` folder, as highlighted in the following folder structure:

```
-- ...
-- client/
-- dist/
    --- static_assets/
    --- ...
    --- client.bundle.js
    --- index.bundle.js
-- ...
-- server/
    --- ...
    --- vr/
        ---- index.html
-- ...
```

This will bring the React 360 code over to the MERN application. However, to make it functional, we need to update the file references in the `index.html` code, as discussed in the next section.

Updating references in index.html

The `index.html` file, which was generated after bundling the React 360 project, references the bundle files expecting these files to be in the same folder, as shown in the following code:

```
<html>
  <head>
    <title>MERNVR</title>
    <style>body { margin: 0 }</style>
    <meta name="viewport" content="width=device-width, initial-scale=1,
user-scalable=no">
  </head>
  <body>
    <!-- Attachment point for your app -->
    <div id="container"></div>
    <script src="./client.bundle.js"></script>
    <script>
      // Initialize the React 360 application
      React360.init(
        'index.bundle.js',
        document.getElementById('container'),
        {
          assetRoot: 'static_assets/',
        }
      )
    </script>
  </body>
</html>
```

We need to update this `index.html` code to refer to the correct location of the `client.bundle.js`, `index.bundle.js`, and `static_assets` folders.

First, update the reference to `client.bundle.js` as follows, to point to the file we placed in the `dist` folder:

```
|<script src="/dist/client.bundle.js" type="text/javascript"></script>
```

Then, update the `React360.init` call with the correct reference to `index.bundle.js`, and `assetRoot` set to the correct location of the `static_assets` folder, as shown in the following code:

```
React360.init(  
    './.../dist/index.bundle.js',  
    document.getElementById('container'),  
    { assetRoot: '/dist/static_assets/' }  
)
```

The `assetRoot` specifies where to look for asset files when we use `asset()` to set resources in the React 360 components.

The game view implemented with React 360 is now available in the MERN application. In the next section, we will try out this integration by setting up a route to load the game from the web application.

Trying out the integration

If we set up an Express route in the MERN application to return the `index.html` file in the response, then visiting the route in the browser will render the React 360 game. To test out this integration, we can set up an example route, as follows:

```
router.route('/game/play')
  .get((req, res) => {
    res.sendFile(process.cwd() + '/server/vr/index.html')
  })
```

This declares a `GET` route in the `'/game/play'` path, which will simply return the `index.html` file that we placed in the `vr` folder with the server code, in response to the requesting client.

Then, we can run the MERN server and open this route in the browser at `localhost:3000/game/play`. This should render the complete React 360 game implemented in this chapter from within the MERN-based web application.

Summary

In this chapter, we used React 360 to develop a web-based VR game that can be easily integrated into MERN applications.

We began by defining simple VR features for the gameplay. Then, we set up React 360 for development and looked at key VR concepts, such as equirectangular panoramic images, 3D positions, and coordinate systems in the 360-degree VR world. We explored the React 360 components and APIs required to implement the game features, including components such as `View`, `Text`, `Entity`, and `VrButton`, along with the `Environment`, `VrHeadModel`, and `NativeModules` APIs.

Finally, we updated the code in the starter React 360 project to implement the game with sample game data, then we bundled the code files and discussed how to add these compiled files to an existing MERN application.

With these steps covered, you will now be able to build your own VR interfaces with React 360, which can be easily integrated with any MERN-based web application.

In the next chapter, we will develop the MERN VR Game application, complete with a game database and backend APIs. This is so that we can make the game developed in this chapter dynamic by fetching data from a game collection stored in MongoDB.

Making the VR Game Dynamic using MERN

In this chapter, we will extend the **MongoDB, Express.js, React.js, and Node.js (MERN)** skeleton application to build the MERN VR Game application, and use it to convert the static React 360 game developed in the previous chapter into a dynamic game. We will achieve this by replacing the sample game data with game details fetched directly from the database. We will use the MERN stack technologies to implement a game model and **Create, Read, Update, and Delete (CRUD) application programming interfaces (APIs)** in the backend, which will allow storage and retrieval of games, and frontend views, which will allow users to make their own games besides playing any of the games on the platform in their browser. We will update and integrate the game developed with React 360 into the game platform developed with MERN technologies. After completing these implementations and integration, you will be able to design and build your own full-stack web applications with dynamic VR features.

To make MERN VR Game a complete and dynamic game application, we will implement the following:

- A game model schema to store game details in MongoDB
- APIs for game CRUD operations
- React views for creating, editing, listing, and deleting games
- Updating the React 360 game to fetch data with the API
- Loading the VR game with dynamic game data

Introducing the dynamic MERN VR Game application

Throughout this chapter, we will develop the MERN VR Game application with MERN-stack technologies. On this platform, registered users will be able to make and modify their own games by providing an equirectangular image for the game world, and the VR object resources, including transform property values for each object to be placed in the game world. Any visitor to the application will be able to browse through all the games added by the makers and play any game, to find and collect the 3D objects in the game world that are relevant to the clue or description of each game. When a registered user signs into the application, they will see a home page with all the games listed and an option to make their own game, as pictured in the following screenshot:

Space Exploration

by Jamie Woolcraft

If I could, then I surely would, cruise the Milky Way, and collect the celestial

PLAY GAME

Hidden Treasures

by Jack Michaels

Look at every nook and corner, hidden in plain sight is your treasure

PLAY GAME

Woodland Critters

by Antonia Thatcher

The woods awake, the trees rustle, and the little critters scurry and scamper all day

PLAY GAME

The Declutterer

by Thomas Brogan

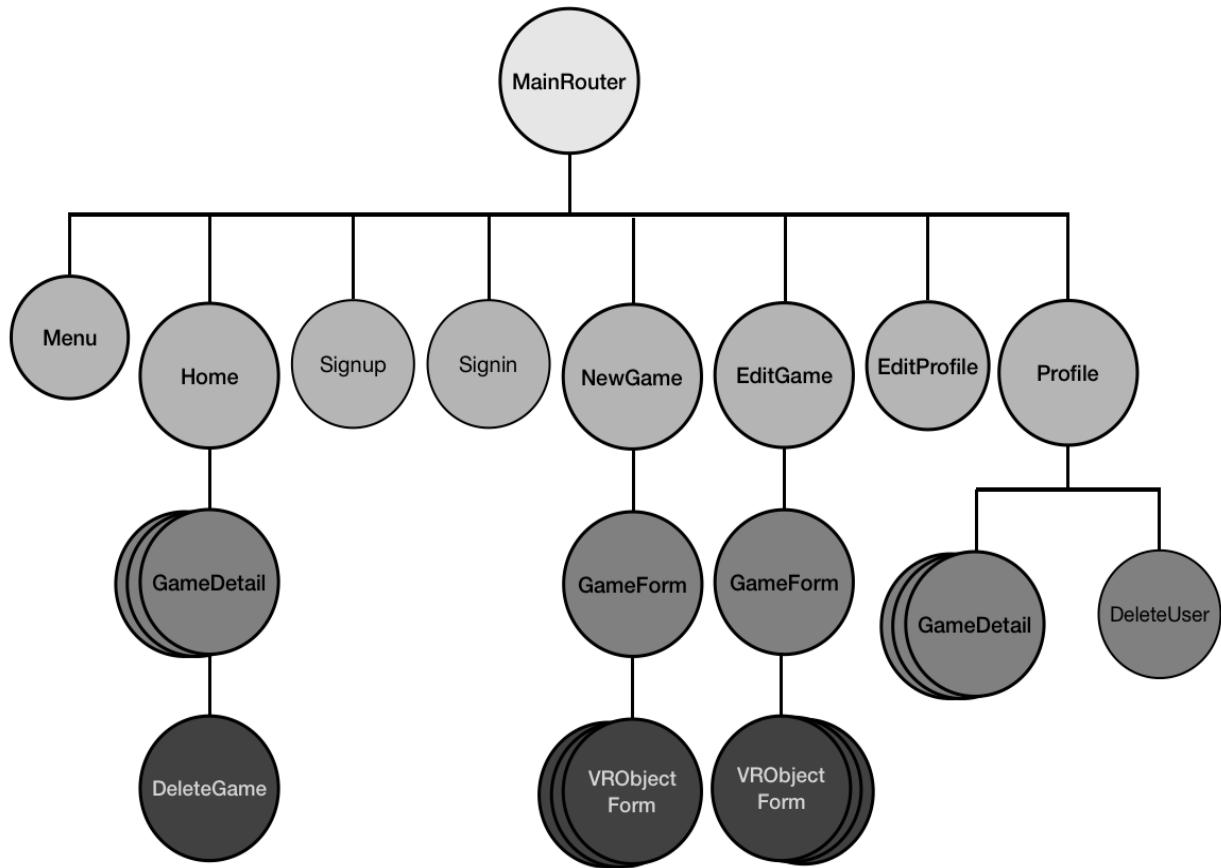
Clutter, clutter everywhere! I wish to make it all disappear!

PLAY GAME

The code for the complete MERN VR Game application is available on GitHub at <https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter14/mern-vr-game>. You can clone this code and run the application as you go through the code explanations for the rest of this chapter.

The views needed for the MERN VR Game application will be developed by extending and modifying the existing React components in the MERN skeleton application. The component tree

pictured in the following screenshot shows all the custom React components that make up the MERN VR Game frontend developed in this chapter:



We will add new React components related to creating, editing, and listing VR games, and will also modify existing components such as the `Profile`, `Menu`, and `Home` components as we build out the features of the MERN VR Game application in the rest of the chapter. The core features in this game platform depend on the capability to store specific details of each game. In the next section, we will begin implementing the MERN VR Game application by defining a game model for storing details of each game.

Defining a Game model

In order to store details of each game in the platform, we will implement a Mongoose model to define a Game model, and the implementation will be similar to other Mongoose model implementations covered in previous chapters, such as the Course model defined in [Chapter 6](#), *Building a Web-Based Classroom Application*. In [Chapter 13](#), *Developing a Web-Based VR Game*, the *Game data structure* section laid out the details needed for each game in order to implement the scavenger-hunt features defined for the gameplay.

We will design the game schema based on these specific details about the game, its VR objects, and also a reference to the game maker. In the following sections, we will discuss the specifics of the game schema, the sub-schema for storing individual VR objects that will be a part of the game, and the validation check to ensure a minimum number of VR objects are placed in the game.

Exploring the game schema

The game schema, which defines the game model with a structure for the game data, will specify the fields to store details about each game. These details will include a game name; a link for the game world image file, text description, or clue; arrays containing details of 3D objects in the game, timestamps indicating when the game was created or updated; and a reference to the user who created the game. The schema for the game model will be defined in `server/models/game.model.js`, and the code defining these game fields is given in the following list, with explanations:

- **Game name:** The `name` field will store a title for the game. It is declared to be a `String` type and will be a required field:

```
name: {  
  type: String,  
  trim: true,  
  required: 'Name is required'  
},
```

- **World image URL:** The `world` field will contain the URL pointing to the equirectangular image that makes up the 3D world of the game. It is declared to be a `String` type and will be a required field:

```
world: {  
  type: String, trim: true,  
  required: 'World image is required'  
},
```

- **Clue text:** The `clue` field will store text of `String` type to give a description of the game or clues about how to complete the game:

```
clue: {  
  type: String,  
  trim: true  
},
```

- **Collectable and other VR objects:** The `answerObjects` field will be an array containing details of the VR objects to be added to the game as collectable objects, whereas the `wrongObjects` field will be an array with VR objects that cannot be collected in the game. Objects in these arrays will be defined in a separate VR object schema, as discussed in the next section:

```
|   answerObjects: [VRObjectSchema],  
|   wrongObjects: [VRObjectSchema],
```

- **Created at and updated at times:** The `created` and `updated` fields will be `Date` types, with `created` generated when a new game is added, and `updated` changed when any game details are modified:

```
|     updated: Date,  
|     created: {  
|       type: Date,  
|       default: Date.now  
|     },
```

- **Game maker:** The `maker` field will be a reference to the user who made the game:

```
|   maker: {type: mongoose.Schema.ObjectId, ref: 'User'}
```

These fields added in the game schema definition will capture details of each game on the platform and allow us to implement the game-related features in the MERN VR Game application. The VR objects to be stored in the `answerObjects` and `wrongObjects` arrays in the game schema will hold details of each VR object to be placed in the game world. In the next section, we will explore the schema defining the details to be stored for each VR object.

Specifying the VR object schema

The `answerObjects` and `wrongObjects` fields already defined in the game schema will both be arrays of VR object documents. These documents will represent the VR objects that are a part of the game. We will define the VR object Mongoose schema for these documents separately, with fields for storing the URLs of the **OBJ** file and **Material Template Library (MTL)** file, along with the React 360 transform values, the scale value, and color value for each VR object.

The schema for the VR object will also be defined in `server/models/game.model.js`, and the code defining these fields is given in the following list, with explanations:

- **OBJ and MTL file URLs:** The `objUrl` and `mtlUrl` fields will store the links to the OBJ and MTL files representing the 3D object data. These fields will be of `String` type and are required fields for storing a VR object:

```
    objUrl: {
      type: String, trim: true,
      required: 'OBJ file is required'
    },
    mtlUrl: {
      type: String, trim: true,
      required: 'MTL file is required'
    },
```

- **Translation transform values:** The `translateX`, `translateY`, and `translateZ` fields will hold the position values of the VR object in 3D space. These fields will be of `Number` type, and the default value for each will be `0`:

```
    translateX: {type: Number, default: 0},
    translateY: {type: Number, default: 0},
    translateZ: {type: Number, default: 0},
```

- **Rotation transform values:** The `rotateX`, `rotateY`, and `rotateZ` fields will hold the orientation values of the VR object in 3D space. These fields will be of `Number` type, and the default value for each will be `0`:

```
|   rotateX: {type: Number, default: 0},  
|   rotateY: {type: Number, default: 0},  
|   rotateZ: {type: Number, default: 0},
```

- **Scale:** The `scale` field will represent the relative size appearance of the VR object. This field will be of `Number` type, and the default value will be `1`:

```
|   scale: {type: Number, default: 1},
```

- **Color:** The `color` field will specify the default color of the object if it is not provided in the MTL file. This field will be of `String` type, and the default value will be `white`:

```
|   color: {type: String, default: 'white'}
```

These fields in the VR object schema represent a VR object to be added to the game world. When a new game document is saved to the database, the `answerObjects` and `wrongObjects` arrays will be populated with `VRObject` documents that adhere to this schema definition. When a user is creating a new game using this Game model with the defined game and VR object schemas, we want to ensure the user adds at least one VR object to each array in the game data. In the next section, we will take a look at how to add this validation check to the Game model.

Validating array length in the game schema

In the game schema defining the Game model, we have two arrays for adding VR objects to the game. These `answerObjects` and `wrongObjects` arrays in a game document must contain at least one VR object in each array when a game is being saved in the game collection. To add this validation for a minimum array length to the game schema, we will add the following custom validation checks to the `answerObjects` and `wrongObjects` paths in the `GameSchema` defined with Mongoose.

We will use `validate` to add the array length validation for the `answerObjects` field, as shown in the following code:

mern-vrgame/server/models/game.model.js:

```
GameSchema.path('answerObjects').validate(function(v) {
  if (v.length == 0) {
    this.invalidate('answerObjects',
      'Must add atleast one VR object to collect')
  }
}, null)
```

In this validation check, if the array length is found to be `0`, we throw a validation error message indicating that at least one object must be added to the array, before saving the game document in the database.

The same validation code is also added for the `wrongObjects` field, as shown in the following code:

mern-vrgame/server/models/game.model.js:

```
GameSchema.path('wrongObjects').validate(function(v) {
  if (v.length == 0) {
    this.invalidate('wrongObjects',
      'Must add atleast one other VR object')
```

```
| },  
| }, null)
```

These checks run every time a game is to be saved in the database and help ensure the game is made with at least two VR objects, including one object that can be collected and another object that cannot be collected. These schema definitions and validations used for defining the Game model will allow a game database for the application to be maintained. This game collection will cater to all the requirements for developing a dynamic VR game according to the specifications of the MERN VR Game application. In order to allow users to access the game collection, for both making their own games and retrieving games made by others, we need to implement corresponding CRUD APIs in the backend. In the next section, we will implement these CRUD APIs that will allow users to create, read, list, update, and delete games from the application.

Implementing game CRUD APIs

In order to build a game platform that allows VR games to be made, managed, and accessed, we need to extend the backend to accept requests that enable game data manipulation in the database. To make these features possible, the backend in the MERN VR Game application will expose a set of CRUD APIs for creating, editing, reading, listing, and deleting games from the database, which can be used in the frontend of the application with fetch calls, including in the React 360 game implementation. In the following sections, we will implement these CRUD API endpoints in the backend, along with the corresponding `fetch` methods that will be deployed in the frontend to use these APIs.

Creating a new game

A user who is signed in to the application will be able to create new games in the database with the create game API endpoint. For the implementation of this API in the backend, we will first declare a `POST` route at `/api/games/by/:userId`, as shown in the following code:

mern-vrgame/server/routes/game.routes.js:

```
| router.route('/api/games/by/:userId')
|   .post(authCtrl.requireSignin, gameCtrl.create)
```

A `POST` request to this route will process the `:userId` param, verify that the current user is signed in, and then create a new game with the game data passed in the request.

The `game.routes.js` file containing this route declaration will be very similar to the `user.routes` file, and to load these new routes in the Express app, we need to mount the game routes in `express.js`, just as we did for the auth and user routes. The game routes can be mounted in the Express app by adding the following line of code:

mern-vrgame/server/express.js:

```
| app.use('/', gameRoutes)
```

This will make the declared game routes available for receiving requests when the server is running.

After a request is received by this create game API, to process the `:userId` param and retrieve the associated user from the database we will utilize the `userByID` method from the user controller. We will also add the following code to the game routes, so the user is available in the `request` object:

mern-vrgame/server/routes/game.routes.js:

```
| router.param('userId', userCtrl.userByID)
```

Once the user authentication is verified after receiving the `POST` request containing the game data in the body, the `create` controller method is invoked next, to add the new game to the database. This `create` controller method is defined as shown in the following code:

mern-vrgame/server/controllers/game.controller.js

```
const create = async (req, res, next) => {
  const game = new Game(req.body)
  game.maker = req.profile
  try{
    let result = await game.save()
    res.status(200).json(result)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

In this `create` method, a new game document is created according to the game model and the data passed in the request body from the client side. Then, this document is saved in the Game collection after the user reference is set as the game maker.

On the frontend, we will add a corresponding `fetch` method in `api-game.js` to make a `POST` request to the create game API by passing the form data collected from the signed-in user. This `fetch` method is defined as shown in the following code:

mern-vrgame/client/game/api-game.js

```
const create = async (params, credentials, game) => {
  try {
    let response = await fetch('/api/games/by/' + params.userId, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      },
      body: JSON.stringify(game)
    })
  }
```

```
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}
```

This `fetch` method will be used in the frontend and provided the new game data with the user credentials needed to make the `POST` request to the create game API. The response from the `fetch` method will tell the user if the game was created successfully.

This create game API endpoint is ready to be used in a form view that can collect the new game details from the user, so new games can be added to the database. In the next section, we will implement an API endpoint that will retrieve the games already added to the database.

Listing all games

In the MERN VR Game application, it will be possible to retrieve a list of all the games in the Game collection from the database using a list games API in the backend. We will implement this API endpoint in the backend by adding a `GET` route to the game routes, as shown in the following code:

```
mern-vrgame/server/routes/game.routes.js:
```

```
| router.route('/api/games')
|   .get(gameCtrl.list)
```

A `GET` request to `/api/games` will execute the `list` controller method, which will query the Game collection in the database, to return all the games in the response to the client.

This `list` controller method will be defined as follows:

```
mern-vrgame/server/controllers/game.controller.js:
```

```
| const list = async (req, res) => {
|   try {
|     let games = await Game.find({}).populate('maker', '_id name').sort('-created').exec()
|     res.json(games)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

In this method, the results retrieved by the query to the Game collection are sorted by the date of creation, with the latest games listed first. Each game in the list will also populate the name and ID of the user who created it. The resulting list of sorted games is returned in the response to the requesting client.

In the frontend, to fetch the games using this list API, we will set up a corresponding `fetch` method in `api-game.js`, as shown in the following code:

mern-vrgame/client/game/api-game.js:

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

This `fetch` method can be used in any frontend interface to make a call to the list games API. The `fetch` will make a `GET` request to the API and receive the list of games in the response, which can be rendered in the interface. In the next section, we will implement another listing API that will only return the games made by a specific user.

Listing games by the maker

In the MERN VR Game application, it will also be possible to retrieve a list of games made by a specific user. To implement this, we will add another API endpoint in the backend that accepts a `GET` request at the `/api/games/by/:userId` route. This route will be declared with the other game routes, as shown in the following code:

```
mern-vrgame/server/routes/game.routes.js:
```

```
| router.route('/api/games/by/:userId')
|   .get(gameCtrl.listByMaker)
```

A `GET` request received at this route will invoke the `listByMaker` controller method, which will query the Game collection in the database to get the matching games. The `listByMaker` controller method will be defined as follows:

```
mern-vrgame/server/controllers/game.controller.js:
```

```
| const listByMaker = async (req, res) => {
|   try {
|     let games = await Game.find({maker:
|       req.profile._id}).populate('maker', '_id name')
|     res.json(games)
|   } catch (err) {
|     return res.status(400).json({
|       error: errorHandler.getErrorMessage(err)
|     })
|   }
| }
```

In the query to the Game collection in this method, we find all the games where the `maker` field matches the user specified in the `userId` route parameter. The retrieved games will contain the maker name and ID and will be returned in the response to the requesting client.

In the frontend, to fetch the games for a specific user with this list by the maker API, we will add a corresponding `fetch` method in `api-game.js`, as shown in the following code:

mern-vrgame/client/game/api-game.js:

```
const listByMaker = async (params, signal) => {
  try {
    let response = await fetch('/api/games/by/' + params.userId, {
      method: 'GET',
      signal: signal,
    })
    return await response.json()
  } catch (err) {
    console.log(err)
  }
}
```

This `fetch` method can be invoked in the frontend interface with the user ID to make a call to the list games by the maker API.

The `fetch` method will make a `GET` request to the API and receive the list of games made by the user specified in the URL. In the next section, we will implement a similar `GET` API to retrieve details of an individual game.

Loading a game

In the backend of the MERN VR Game application, we will expose an API that will retrieve the details of an individual game, specified by its ID in the game collection. To achieve this, we can add a `GET` API that queries the Game collection with an ID and returns the corresponding game document in the response. We will start implementing this API to fetch a single game by declaring a route that accepts a `GET` request at `'/api/game/:gameId'`, as shown in the following code:

```
mern-vrgame/server/routes/game.routes.js:
```

```
| router.route('/api/game/:gameId')
|   .get(gameCtrl.read)
```

When a request is received at this route, the `:gameId` param in the route URL will be processed first to retrieve the individual game from the database. So, we will also add the following to the game routes:

```
| router.param('gameId', gameCtrl.gameByID)
```

The presence of the `:gameId` param in the route will invoke the `gameByID` controller method, which is similar to the `userByID` controller method. It will retrieve the game from the database and attach it to the `request` object to be used in the `next` method. This `gameByID` controller method is defined as shown in the following code:

```
mern-vrgame/server/controllers/game.controller.js:
```

```
| const gameByID = async (req, res, next, id) => {
|   try {
|     let game = await Game.findById(id).populate('maker', '_id
| name').exec()
|     if (!game)
|       return res.status('400').json({
|         error: "Game not found"
|       })
|     req.game = game
```

```
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve game"
    })
  }
}
```

The game queried from the database will also contain the name and ID details of the maker, as specified in the `populate()` method. The `next` method—in this case, the `read` controller method—simply returns this retrieved game in response to the client. This `read` controller method is defined as follows:

mern-vrgame/server/controllers/game.controller.js:

```
const read = (req, res) => {
  return res.json(req.game)
}
```

This API to read a single game's details will be used to load a game in the React 360 implementation of the game world. We can call this API in the frontend code using a `fetch` method, to retrieve the details of an individual game according to its ID. A corresponding `fetch` method can be defined to call this game API, as shown in the following code:

mern-vrgame/client/game/api-game.js:

```
const read = async (params) => {
  try {
    let response = await fetch('/api/game/' + params gameId, {
      method: 'GET'
    })
    return await response.json()
  } catch (err) {
    console.log(err)
  }
}
```

This `read` method will take the game ID in the `params` and make a `GET` request to the API, using a `fetch` method.

This API for loading a single game will be used for the React views fetching a game detail and also the React 360 game view, which will render the game interface in the MERN VR Game application. In the next section, we will implement the API that will allow makers to update the games they already created on the platform.

Editing a game

Authorized users who are signed in—and also the maker of a specific game—will be able to edit the details of that game in the database. To enable this feature, we will implement an edit game API in the backend. We will add a `PUT` route that allows an authorized user to edit one of their games. The route will be declared as follows:

```
mern-vrgame/server/routes/game.routes.js:
```

```
| router.route('/api/games/:gameId')
|   .put(authCtrl.requireSignin, gameCtrl.isMaker, gameCtrl.update)
```

A `PUT` request to `'/api/games/:gameId'` will first execute the `gameByID` controller method to retrieve the specific game's details. The `requireSignin` auth controller method will also be called to ensure the current user is signed in. Then, the `isMaker` controller method will determine whether the current user is the maker of this specific game, before finally running the `game update` controller method to modify the game in the database.

The `isMaker` controller method ensures that the signed-in user is actually the maker of the game being edited, and it is defined as shown in the following code:

```
mern-vrgame/server/controllers/game.controller.js:
```

```
| const isMaker = (req, res, next) => {
|   let isMaker = req.game && req.auth && req.game.maker._id ==
|     req.auth._id
|   if(!isMaker){
|     return res.status('403').json({
|       error: "User is not authorized"
|     })
|   }
|   next()
| }
```

If the `isMaker` condition is not met, that means the currently signed-in user is not the maker of the game being edited, and an authorization error is returned in the response. But if the condition is met, the `next` method is invoked instead. In this case, the update controller method is the `next` method, and it saves the changes to the game in the database. This update method is defined as shown in the following code:

mern-vrgame/server/controllers/game.controller.js:

```
const update = async (req, res) => {
  try {
    let game = req.game
    game = extend(game, req.body)
    game.updated = Date.now()
    await game.save()
    res.json(game)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This `update` method will take the existing game details and the form data received in the request body to merge the changes and save the updated game to the Game collection in the database.

This edit game API can be called in the frontend view using a `fetch` method that takes the changes as form data and sends it with the request to the backend, along with user credentials. The corresponding `fetch` method is defined as shown in the following code:

mern-vrgame/client/game/api-game.js:

```
const update = async (params, credentials, game) => {
  try {
    let response = await fetch('/api/games/' + params gameId, {
      method: 'PUT',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      },
    })
```

```
    body: JSON.stringify(game)
  })
  return await response.json()
} catch(err) {
  console.log(err)
}
}
```

This method makes the `PUT` request to the edit game API, providing the changes to the game in the request body, the current user's credentials in the request header, and the ID of the game to be edited in the route URL. This method can be used in the frontend, which renders a form allowing users to update the game details. In the next section, we will implement another API in the backend that will allow authorized users to delete the games that they created on the platform.

Deleting a game

An authenticated and authorized user will be able to delete any of the games they created on the application. To enable this feature, we will implement a delete game API in the backend. We will start by adding a `DELETE` route that allows an authorized maker to delete one of their own games, as shown in the following code:

```
mern-vrgame/server/routes/game.routes.js:
```

```
| router.route('/api/games/:gameId')
|   .delete(authCtrl.requireSignin, gameCtrl.isMaker, gameCtrl.remove)
```

The flow of the controller method execution on the server, after receiving the `DELETE` request at `/api/games/:gameId`, will be similar to the edit game API, with the final call made to the `remove` controller method instead of `update`.

The `remove` controller method deletes the specified game from the database when a `DELETE` request is received at `/api/games/:gameId`, and it has been verified that the current user is the original maker of the given game. The `remove` controller method is defined as shown in the following code:

```
mern-vrgame/server/controllers/game.controller.js:
```

```
const remove = async (req, res) => {
  try {
    let game = req.game
    let deletedGame = await game.remove()
    res.json(deletedGame)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This `remove` method permanently deletes the specified game from the game collection in the database.

To use this API from the frontend, we will add a corresponding `remove` method in `api-game.js` to make a fetch request to the delete game API. This `fetch` method is defined as follows:

mern-vrgame/client/game/api-game.js:

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

This method uses `fetch` to make a `DELETE` request to the delete game API. It takes the game ID in the `params` and the user credentials that are needed by the API endpoint in the backend to check if this current user is the authorized maker of the specified game. If the request is successful and the corresponding game is removed from the database, a success message is returned in the response.

With these game CRUD APIs functional in the backend, we are ready to implement the frontend that will use these APIs to allow users to create new games, list the games, modify existing games, and load a single game in the React 360 game view. We can start building out this frontend in the next section, starting with the React views for creating and editing games in the application.

Adding a form for creating and editing games

Users registered on the MERN VR Game application will be able to make new games and modify these games from views on the application. To implement these views, we will add React components that allow users to compose and modify the game details and VR object details for each game. As the form for creating new and editing existing games will have similar form fields for composing game details and VR object details, we will make reusable components that can be used both for creating and editing purposes. In the following sections, we will discuss the form views for creating a new game and editing an existing game, and the implementation of the common form components in these views.

Making a new game

When any user signs into the application, they will be given the option to make their own VR game. They will see a MAKE GAME link on the menu that will navigate them to a form where they can fill in the game details to create a new game on the platform. In the following sections, we will update the frontend code to add this link on the menu and implement the `NewGame` component, which will contain the form to create a new game.

Updating the menu

We will update the navigation menu in the application to add the MAKE GAME button, which will appear conditionally based on whether the user is signed in, and redirect the user to a view containing the form to create a new game. The MAKE GAME button will render on the menu, as shown in the following screenshot:



To add this button to the `Menu` component, we will use a `Link` component with the route for the `NewGame` component containing the form. To make it render conditionally, we will place it right before the MY PROFILE link shown in the preceding screenshot, in the section that renders only when the user is authenticated. The button code will be added as shown in the following code:

mern-vrgame/client/core/Menu.js:

```
|<Link to="/game/new">
|  <Button style={isActive(history, "/game/new")}>
|    <AddBoxIcon color="secondary"/> Make Game
|  </Button>
|</Link>
```

This will show the MAKE GAME option to signed-in users, and they can click on it to be redirected to the `/game/new` route containing the form view for making a new game on the platform. In the next section, we will look at the component that will render this form.

The NewGame component

We will implement the form view for creating a new game in the `NewGame` React component. This form view will allow users to fill out the fields for a single game. The `NewGame` component will render these form elements corresponding to the game details, including VR object details, as shown in the following screenshot:

New Game

Game World Equirectangular Image (URL)

Name

Clue Text

VR Objects to collect

^

[+] ADD OBJECT

Other VR objects

^

[+] ADD OBJECT

SUBMIT

CANCEL

The `NewGame` component will use the `GameForm` component, which will contain all the rendered form fields, to compose this new game form. The `GameForm` component will be a reusable component that we will use in both the create and edit forms.

When added to the `NewGame` component, it takes an `onSubmit` method as a prop, along with any server-returned error messages, as shown in the following code:

mern-vrgame/client/game/NewGame.js:

```
| <GameForm onSubmit={clickSubmit} errorMsg={error} />
```

The method passed in the `onSubmit` prop will be executed when the user submits the form. The `clickSubmit` method passed in this case is defined in the `NewGame` component. It uses the create game `fetch` method from `api-game.js` to make a `POST` request to the create game API with the game form data and user details.

This `clickSubmit` method is defined as shown in the following code:

mern-vrgame/client/game/NewGame.js:

```
const clickSubmit = game => event => {
  const jwt = auth.isAuthenticated()
  create({
    userId: jwt.user._id
  }, {
    t: jwt.token
  }, game).then((data) => {
    if (data.error) {
      setError(data.error)
    } else {
      setError('')
      setRedirect(true)
    }
  })
}
```

If the user makes an error while entering the game details in the form, the backend sends back an error message when this `clickSubmit` method is called on form submission. If there are no errors and the

game is successfully created in the database, the user is redirected to another view.

To load this `NewGame` component at a specified URL and only for authenticated users, we will add a `PrivateRoute` in `MainRouter`, as shown in the following code:

mern-vrgame/client/MainRouter.js:

```
| <PrivateRoute path="/game/new" component={NewGame} />
```

This will make the `NewGame` component load in the browser at the `/game/new` path when an authenticated user is accessing it. In the next section, we will see a similar implementation for rendering the same form to edit an existing game from the database.

Editing the game

Users will be able to edit the games they made on the platform using a form similar to the form for creating new games. We will implement this edit game view in the `EditGame` component, which will render the game form fields pre-populated with the existing game's details. We will look at the implementation of this `EditGame` component in the following section.

The EditGame component

Just as in the `NewGame` component, the `EditGame` component will also use the `GameForm` component to render the form elements. But in this form, the fields will load the current values of the game to be edited, and users will be able to update these values, as pictured in the following screenshot:

Edit Game



Game World Equirectangular Image (URL)

<https://s3.amazonaws.com/mernbook/vrObjects/26179>

Name

The Declutterer

Clue Text

Clutter, clutter everywhere! I wish to make it all disappear!

VR Objects to collect

.obj url

<https://s3.amazonaws.com/mernbook/vr>

.mtl url

<https://s3.amazonaws.com/mernbook/vr>

TranslateX TranslateY TranslateZ

25 -15 25

RotateX RotateY RotateZ

90 270 0

Scale

Color

12 white DELETE



Other VR objects

SUBMIT

CANCEL

In the case of this `EditGame` component, the `GameForm` will take the given game's ID as a prop so that it can fetch the game details, in addition to the `onSubmit` method and server-generated error message, if any. The `GameForm` component will be added to the `EditGame` component with these props, as follows:

mern-vrgame/client/game/EditGame.js:

```
| <GameForm gameId={params.gameId} onSubmit={clickSubmit} errorMsg={error}/>
```

The `clickSubmit` method for the edit form will use the update game `fetch` method in `api-game.js` to make a `PUT` request to the edit game API with the form data and user details. The `clickSubmit` method for this edit form submission will be defined as shown in the following code:

mern-vrgame/client/game/EditGame.js:

```
const clickSubmit = game => event => {
  const jwt = auth.isAuthenticated()
  update({
    gameId: match.params.gameId
  }, {
    t: jwt.token
  }, game).then((data) => {
    if (data.error) {
      setError(data.error)
    } else {
      setError('')
      setRedirect(true)
    }
  })
}
```

If the user makes an error while modifying the game details in the form, the backend sends back an error message when this `clickSubmit` method is called on form submission. If there are no errors and the game is successfully updated in the database, the user is redirected to another view.

To load this `EditGame` component at a specified URL and only for authenticated users, we will add a `PrivateRoute` in `MainRouter`, as shown in the following code:

mern-vrgame/client/MainRouter.js:

```
| <PrivateRoute path="/game/edit/: gameId" component={EditGame} />
```

The `EditGame` component will load in the browser at the `/game/edit/: gameId` path when an authenticated user is accessing it. Both this `EditGame` component and the `NewGame` component use the `GameForm` component to render the form elements that allow users to add the details of a game. In the next section, we will discuss the implementation of this reusable `GameForm` component.

Implementing the GameForm component

The `GameForm` component is used in both the `NewGame` and `EditGame` components, and it contains the elements that allow users to enter game details and VR object details for a single game. It may start with a blank game object or load an existing game. To begin the implementation of this component, we will first initialize a blank game object in the component state, as shown in the following code:

mern-vrgame/client/game/GameForm.js:

```
const [game, setGame] = useState({ name: '',
                                  clue: '',
                                  world: '',
                                  answerObjects: [],
                                  wrongObjects: []
                                })
```

If the `GameForm` component receives a `gameId` prop from the parent component—such as from the `EditGame` component—then it will use the load game API to retrieve the game's details and set it to the state, to be rendered in the form view. We will make this API call in an `useEffect` hook, as shown in the following code:

mern-vrgame/client/game/GameForm.js:

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

    read({gameId: props.gameId}, signal).then((data) => {
      if (data.error) {
        setReadError(data.error)
      } else {
        setGame(data)
      }
    })
    return function cleanup() {
      abortController.abort()
    }
  }
})
```

```
|   }
| },
| , [])
```

In the `useEffect` hook, we first check if the props received from the parent component contain a `gameId` prop, and then use the value to make the load game API call. If the API call returns an error, we set the error to the state; otherwise, we set the retrieved game to the state. With this code, we will have the initial values for the game details initialized accordingly, to be used in the form view.

The form view part in the `GameForm` component will essentially have two parts: one part that takes simple game details—such as name, world image link, and clue text—as input, and a second part that allows users to add a variable number of VR objects to either the answer objects array or the wrong objects array. In the following sections, we will look at the implementations of these two parts that will make up the game details form view.

Inputting simple game details

While creating or editing a game, users will first see the form elements for the simpler details of the game, such as name, world image URL, and the clue text. This form section with the simple game details will mostly be text input elements added using the Material-UI `TextField` component, with a change handling method passed to the `onChange` handler. We will build out this section in the `GameForm` component, which is implemented in `mern-vrgame/client/game/GameForm.js`, with the following elements, as shown in the associated code:

- **Form title:** The form title will be either `New Game` or `Edit Game`, depending on whether an existing game ID is passed as a prop to `GameForm` from the parent component to which it is added, as shown in the following code:

```
<Typography type="headline" component="h2">
  {props.gameId? 'Edit': 'New'} Game
</Typography>
```

- **Game world image input:** We will render the background image URL in an `img` element at the very top of the form to show users the image they added as the game world image URL. The image URL input will be taken in a `TextField` component below the rendered image, as shown in the following code:

```
<img src={game.world}/>
<TextField id="world" label="Game World Equirectangular Image
(URL)" value={game.world} onChange={handleChange('world')}/>
```

- **Game name:** The game name will be added in a single `TextField` of the default `text` type, as shown in the following code:

```
<TextField id="name" label="Name" value={game.name} onChange=
{handleChange('name')}/>
```

- **Clue text:** The clue text will be added to a multiline `TextField` component, as shown in the following code:

```
<TextField id="multiline-flexible" label="Clue Text" multiline  
rows="2"  
value={game.clue} onChange={handleChange('clue')}/>
```

In these form elements added to the `GameForm` component, the input fields also take an `onChange` handler function, which is defined as `handleChange`. This `handleChange` method will update the game values in the state whenever a user changes a value in an input element. The `handleChange` method is defined as follows:

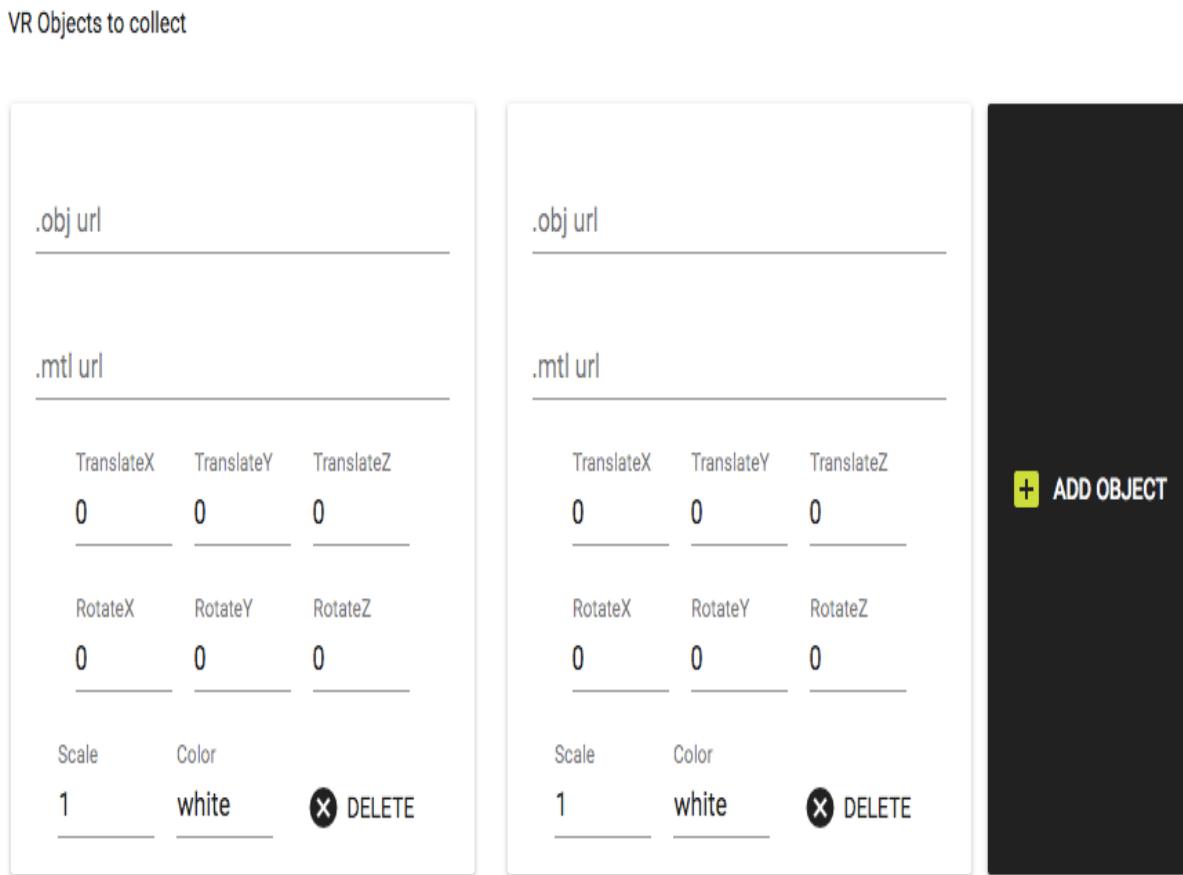
mern-vrgame/client/game/GameForm.js:

```
const handleChange = name => event => {  
  const newGame = {...game}  
  newGame[name] = event.target.value  
  setGame(newGame)  
}
```

In this method, based on the specific field value being changed, we update the corresponding attribute in the game object in the state. This captures the values entered by the user as simple details for their VR game. The form will also give the option to define arrays of VR objects that will also be a part of the game. In the next section, we will look at the form implementation that will allow users to manipulate arrays of VR objects.

Modifying arrays of VR objects

Users will be able to define a dynamic number of VR objects in two different arrays for each game. In order to allow users to modify these arrays of `answerObjects` and `wrongObjects` that they wish to add to their VR game, `GameForm` will iterate through each array and render a VR object form component for each object. With this, it will become possible to add, remove, and modify VR objects from the `GameForm` component, as pictured in the following screenshot:



In the following sections, we will add these array manipulation functionalities in the `GameForm` component. We will start by rendering each item in the VR object arrays and incorporate an option to add a

new item or remove an existing item from an array. Then, since each item in an array will essentially be a form to enter VR object details, we will also discuss how to handle the input changes made within each item from the `GameForm` component.

Iterating and rendering the object details form

We will add the form interface seen in the previous section with Material-UI `ExpansionPanel` components to create a modifiable list of VR objects for each type of VR object array in the given game.

Inside the nested `ExpansionPanelDetails` component, we will iterate through the `answerObjects` array or the `wrongObjects` array to render a `VRObjectForm` component for each VR object, as shown in the following code:

mern-vrgame/client/game/GameForm.js:

```
<ExpansionPanel>
  <ExpansionPanelSummary expandIcon={<ExpandMoreIcon />}>
    <Typography>VR Objects to collect</Typography>
  </ExpansionPanelSummary>
  <ExpansionPanelDetails> {
    game.answerObjects.map((item, i) => {
      return <div key={i}>
        <VRObjectForm index={i} type={'answerObjects'}>
          handleUpdate={handleObjectChange}
          vrObject={item}
          removeObject={removeObject}/>
        </div>
    })
  }
  ...
</ExpansionPanelDetails>
</ExpansionPanel>
```

To render each object in the array, we use a `VRObjectForm` component. We will look at the specific implementation of the `VRObjectForm` component later in the chapter. While adding `VRObjectForm` in this code, we pass the single `vrObject` item as a prop, along with the current `index` in the array, the type of the array, and two methods for updating the state in `GameForm` when the array details are modified by changing details or deleting an object from within the `VRObjectForm`.

component. This will render a form for each VR object in the arrays associated with the game in the `GameForm` component. In the next section, we will see the implementation for including an option to add new objects to these arrays.

Adding a new object to the array

For each array rendered in the game form, we will add a button that will let users push new VR objects to the given array. This button to add an object will render a new `VRObjectForm` component to take the details of a new VR object. We will add this button to the `ExpansionPanelDetails` component after the iteration code, as shown in the following code:

mern-vrgame/client/game/GameForm.js:

```
<ExpansionPanelDetails>
...
    <Button color="primary" variant="contained"
            onClick={addObject('answerObjects')}>
        <AddBoxIcon color="secondary"/>
        Add Object
    </Button>
</ExpansionPanelDetails>
```

This ADD OBJECT button will render at the end of each list of VR object forms. When clicked on, it will add a new blank VR object form by invoking the `addObject` method. This `addObject` method will be defined as follows:

mern-vrgame/client/game/GameForm.js:

```
const addObject = name => event => {
  const newGame = {...game}
  newGame[name].push({})
  setGame(newGame)
}
```

The `addObject` method is passed the array type so we know which array the user wants to add the new object to. In this method, we will just add an empty object to the array being iterated, so an empty form is rendered in its place, which users can fill out to enter new

object details. In the next section, we will see how to let users remove one of these items from a list of VR object forms.

Removing an object from the array

Each of the items rendered in the list of VR object forms can also be removed from the list by the user. The `vRObjectForm` component displaying an item will contain a delete option, which will remove the object from the given array.

To implement the remove item functionality for this DELETE button, we will pass a `removeObject` method as a prop to the `vRObjectForm` component from the parent `GameForm` component. This method will allow the array to be updated in the parent component's state when a user clicks DELETE on a specific `vRObjectForm`.

This `removeObject` method will be defined as shown in the following code:

mern-vrgame/client/game/GameForm.js:

```
const removeObject = (type, index) => event => {
  const newGame = {...game}
  newGame[type].splice(index, 1)
  setGame(newGame)
}
```

In this method, the VR object corresponding to the item clicked will be removed by slicing at the given `index` from the array with the specified array `type`. This updated object array in the game will be reflected in the view when it is set in the state, with the deleted VR object removed from the form view. In the next section, we will look at how to handle changes to the details of a VR object when the user updates values in a VR object form, which is rendered according to items in the VR object arrays.

Handling the object detail change

The details of any VR object in the game will be updated when the user changes input values in any of the fields in the corresponding VR object form. To register this update, the `GameForm` that houses the forms for the VR objects will pass the `handleObjectChange` method to the `VRObjectForm` component, which will render the VR object form.

This `handleObjectChange` method will be defined as follows:

mern-vrgame/client/game/GameForm.js:

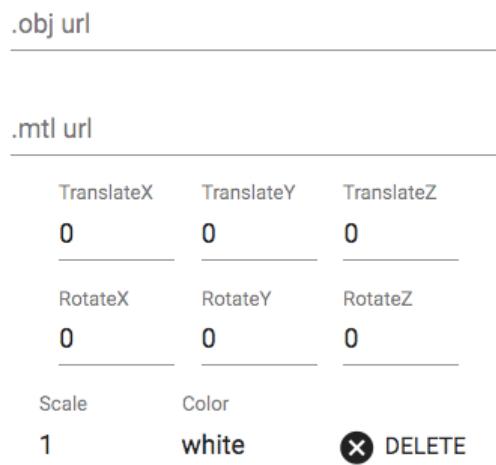
```
const handleObjectChange = (index, type, name, val) => {
  var newGame = {...game}
  newGame[type][index][name] = val
  setGame(newGame)
}
```

This `handleObjectChange` method will be used in the `VRObjectFrom` component to capture the changed input's value and update the corresponding field of the VR object at the specified `index` in the array of the given `type`, so it is reflected in the game object stored in the state in `GameForm`.

The `GameForm` component renders the form elements for modifying the details of a game, including the lists of VR objects. Using this form, users can add, modify, and delete VR objects in the lists. The lists render each item in a VR object form that the users can use to compose the details of the object. In the next section, we will implement the React component that renders this VR object form for each VR object in the game.

The VRObjectForm component

We will implement the `VRObjectForm` component to render the input fields for modifying an individual VR object's details, which are added to the `answerObjects` and `wrongObjects` arrays of the game in the `GameForm` component. The `VRObjectForm` component will render a form, as pictured in the following screenshot:



To begin implementation of this `VRObjectForm` component containing a VR object form, we will start by initializing the blank details of a VR object in the component's state with a `useState` hook, as shown in the following code:

mern-vrgame/client/game/VRObjectForm.js:

```
const [values, setValues] = useState({  
  objUrl: '',  
  mtlUrl: '',  
  translateX: 0,  
  translateY: 0,  
  translateZ: 0,  
  rotateX: 0,  
  rotateY: 0,  
  rotateZ: 0,  
  scale: 1,
```

```
|   color:'white'  
| })
```

These details correspond to the schema defined for storing a VR object. When a `VRObjectForm` component is added to the `GameForm` component, it may receive an empty VR object or a VR object populated with details, depending on whether an empty form or a form with details of an existing object is being rendered. In the case that an existing VR object is passed as a prop, we will set the details of this object in the component state using an `useEffect` hook, as shown in the following code:

mern-vrgame/client/game/VRObjectForm.js:

```
useEffect(() => {  
  if(props.vrObject && Object.keys(props.vrObject).length != 0){  
    const vrObject = props.vrObject  
    setValues({...values,  
      objUrl: vrObject.objUrl,  
      mtlUrl: vrObject.mtlUrl,  
      translateX: Number(vrObject.translateX),  
      translateY: Number(vrObject.translateY),  
      translateZ: Number(vrObject.translateZ),  
      rotateX: Number(vrObject.rotateX),  
      rotateY: Number(vrObject.rotateY),  
      rotateZ: Number(vrObject.rotateZ),  
      scale: Number(vrObject.scale),  
      color:vrObject.color  
    })  
  }  
, [])
```

In this `useEffect` hook, if the `vrObject` value passed in the prop is not an empty object, we set the details of the received VR object in the state. These values will be used in the input fields that make up the VR object form. We will add the input fields corresponding to a VR object's details, in the view of `VRObjectForm` using Material-UI `TextField` components, as shown in the code explained with the following list:

- **3D object file input:** The OBJ and MTL file links will be collected for each VR object as text input using the `TextField` components, as shown in the following code:

```
<TextField label=".obj url" value={values.objUrl}  
  onChange={handleChange('objUrl')} />
```

```
<TextField label=".mtl url" value={values.mtlUrl}  
onChange={handleChange('mtlUrl')} />
```

- **Translate value input:** The translate values of the VR object across the x, y, and z axes will be input in the `TextField` components of the `number` type, as shown in the following code:

```
<TextField type="number" value={values.translateX}  
label="TranslateX" onChange={handleChange('translateX')} />  
<TextField type="number" value={values.translateY}  
label="TranslateY" onChange={handleChange('translateY')} />  
<TextField type="number" value={values.translateZ}  
label="TranslateZ" onChange={handleChange('translateZ')} />
```

- **Rotate value input:** The rotation values of the VR object around the x, y, and z axes will be input in the `TextField` components of the `number` type, as shown in the following code:

```
<TextField type="number" value={values.rotateX}  
label="RotateX" onChange={handleChange('rotateX')} />  
<TextField type="number" value={values.rotateY}  
label="RotateY" onChange={handleChange('rotateY')} />  
<TextField type="number" value={values.rotateZ}  
label="RotateZ" onChange={handleChange('rotateZ')} />
```

- **Scale value input:** The scale value for the VR object will be input in a `TextField` component of the `number` type, as shown in the following code:

```
<TextField type="number" value={values.scale}  
label="Scale" onChange={handleChange('scale')} />
```

- **Object color input:** The color value for the VR object will be input in a `TextField` component of the `text` type, as shown in the following code:

```
<TextField value={values.color} label="Color"  
onChange={handleChange('color')} />
```

These input fields will allow the user to set the details of a VR object in a game. When any of these VR object details are changed in these input fields by the user, the `handleChange` method will be invoked. This `handleChange` method will be defined as shown in the following code:

mern-vrgame/client/game/VRObjectForm.js:

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

This `handleChange` method will update the corresponding value in the state of the `VRObjectForm` component, and use the `handleUpdate` method passed as a prop from `GameForm` to update the VR object in the `GameForm` state with the changed value for the specific object detail.

The `VRObjectForm` will also contain a DELETE button that will execute the `removeObject` method received in the `GameForm` as a prop, which will allow the given object to be removed from the list in the game. This delete button will be added to the view with the following code:

mern-vrgame/client/game/VRObjectForm.js:

```
<Button onClick={props.removeObject(props.type, props.index)}>
  <Icon style={{marginRight: '5px'}}>cancel</Icon> Delete
</Button>
```

The `removeObject` method will take the value of the object array type and the array index position, to remove the given object from the relevant VR object array in the `GameForm` component's state.

With these implementations, the forms for creating and editing games are in place, complete with VR object input forms for arrays of varying sizes. We used reusable components to compose the form elements needed for creating and editing games, along with adding the capability for modifying arrays of VR objects in a game. Any registered user can use these forms to add and edit game details on the MERN VR Game application. In the next section, we will discuss the implementation of the views that will render different lists of games on the platform.

Adding the game list views

Visitors to MERN VR Game will access the games on the application from lists rendered on the home page and individual user profiles. The home page will list all the games on the application, and the games by a specific maker will be listed on their user profile page. These list views will iterate through game data fetched using the backend APIs for listing games, and render details of each game in a reusable React component.

In the following sections, we will discuss the implementation for rendering all games and games only by a specific maker, using a reusable component for rendering each game on the list.

Rendering lists of games

We will render all the games available on the platform on the home page of the application. To implement this feature, the `Home` component will first fetch the list of all the games from the game collection in the database using the list game API. We will achieve this in an `useEffect` hook in the `Home` component, as shown in the following code:

mern-vrgame/client/core/Home.js:

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

  list(signal).then((data) => {
    if (data.error) {
      console.log(data.error)
    } else {
      setGames(data)
    }
  })
  return function cleanup() {
    abortController.abort()
  }
}, [])
```

The list of games retrieved from the server in this `useEffect` hook will be set to the state and iterated over to render a `GameDetail` component for each game in the list, as shown in the following code:

mern-vrgame/client/core/Home.js:

```
{games.map((game, i) => {
  return <GameDetail key={i} game={game} updateGames={updateGames}/>
})}
```

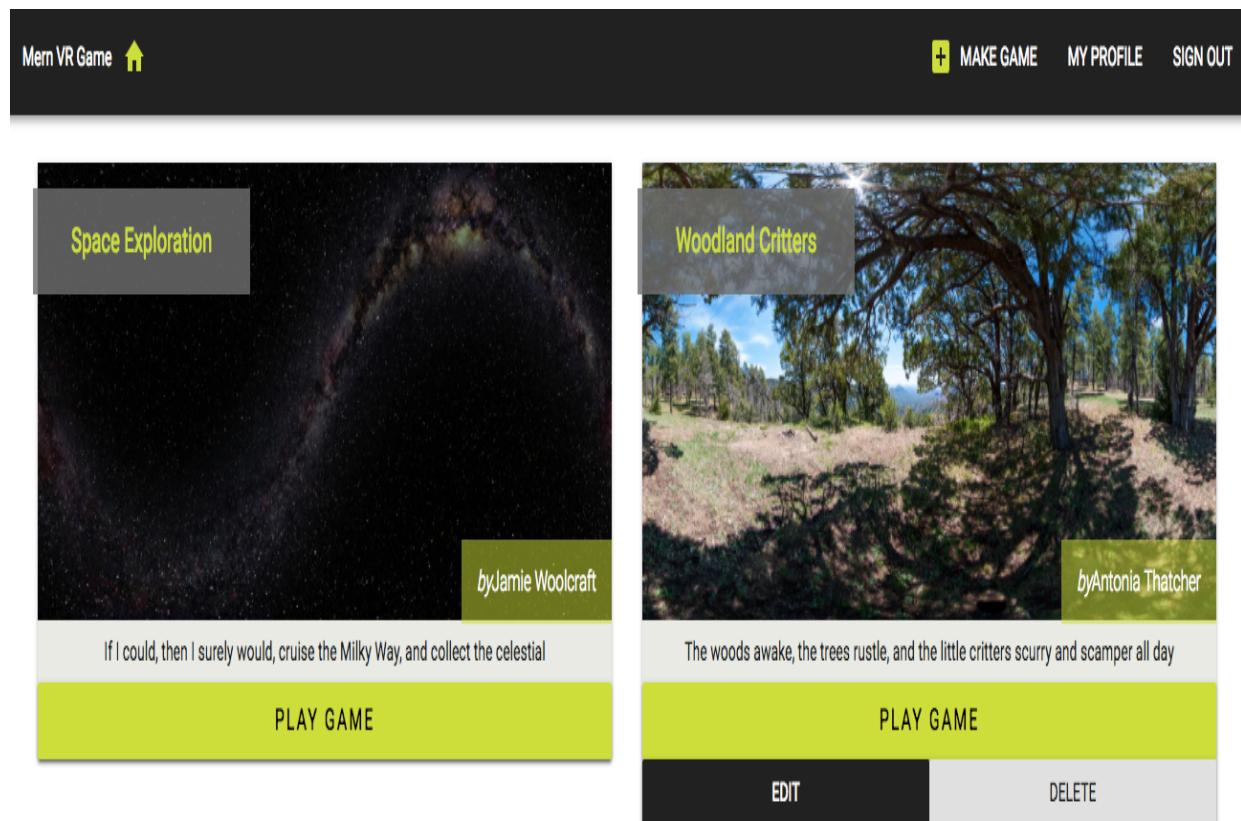
The `GameDetail` component, which will be implemented as a reusable component that renders details of a single game, will be passed the game details and a `updateGames` method. The `updateGames` method will allow the game list in the `Home` component to be updated if any of the games

on the list are deleted by the maker. The `updateGames` method is defined as shown in the following code:

mern-vrgame/client/core/Home.js:

```
const updateGames = (game) => {
  const updatedGames = [...games]
  const index = updatedGames.indexOf(game)
  updatedGames.splice(index, 1)
  setGames(updatedGames)
}
```

The `updateGames` method will update the list rendered in the `Home` component by slicing the specified game from the array of games. This method will be invoked when a user deletes their game using the EDIT and DELETE options rendered conditionally in the `GameDetail` component for the maker of the game, as pictured in the following screenshot of games listed in the home page of the application:



We can render a similar list view in the user profile page, showing only the games made by the corresponding user, as pictured in the following screenshot:

Maker Profile

Jack Michaels
jack@fantastic.info

Joined: Mon Sep 04 2017

Jack's Games

Hidden Treasures

by Jack Michaels

Look at every nook and corner, hidden in plain sight is your treasure

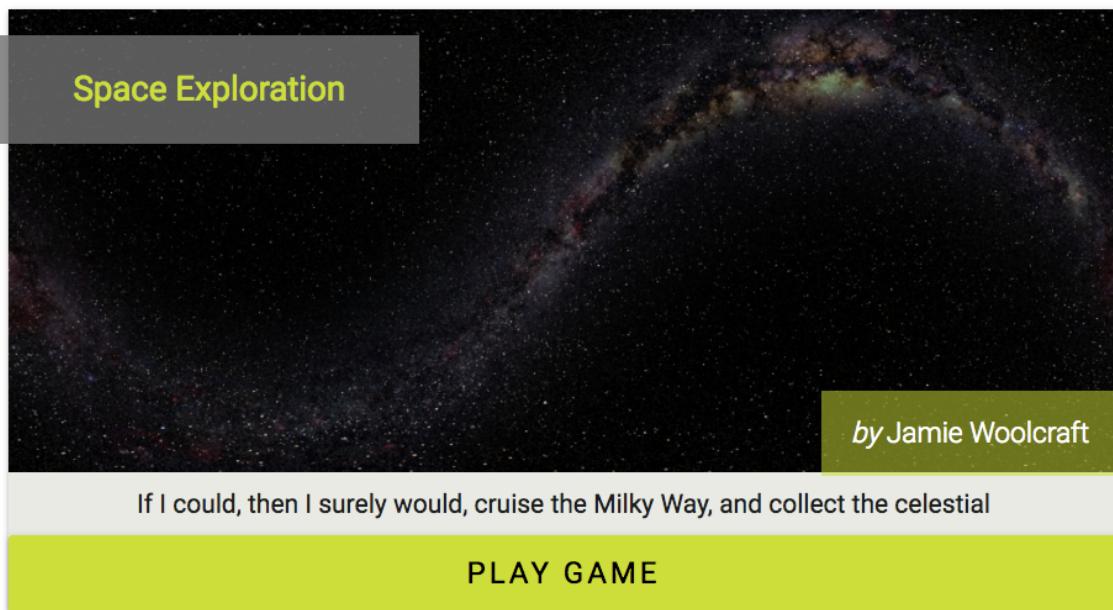
PLAY GAME

EDIT DELETE

Similar to the implementation steps in the `Home` component, in this `Profile` component, we can fetch the list of games by the given user with a call to the associated list games by the maker API in an `useEffect` hook. With the retrieved list of games set in the state, we can iterate over it to render each game in a `GameDetail` component, as discussed earlier, for rendering all games on the home page. In the next section, we will discuss the implementation of this `GameDetail` component that will render the details of a single game.

The GameDetail component

We will implement the `GameDetail` component to render individual games in any game list view in the application. This `GameDetail` component takes the game object as a prop, and renders the details of the game, along with a PLAY GAME button that links to the VR game view, as pictured in the following screenshot:



This component will also conditionally render EDIT and DELETE buttons if the current user is the maker of the game.

In the view code of the `GameDetail` component, we will first add the game details—such as the name, world image, clue text, and maker name—to give users an overview of the game. We will use Material-UI components to compose the interface with these details, as shown in the following code:

`mern-vrgame/client/game/GameDetail.js`:

```

<Typography type="headline" component="h2">
  {props.game.name}
</Typography>
<CardMedia image={props.game.world}
           title={props.game.name}/>
<Typography type="subheading" component="h4">
  <em>by</em>
  {props.game.maker.name}
</Typography>
<CardContent>
  <Typography type="body1" component="p">
    {props.game.clue}
  </Typography>
</CardContent>

```

This code will render the game world image, game name, maker name, and clue text for the game passed in the props.

The PLAY GAME button rendered in the `GameDetail` component will simply be a button wrapped in an HTML link element that points to the route that opens the React 360-generated `index.html` file (implementation for this route on the server is discussed in the *Playing the VR game* section). This PLAY GAME link is added to the `GameDetail` component, as follows:

`mern-vrgame/client/game/GameDetail.js`:

```

<a href={"/game/play?id=" + props.game._id} target='_self'>
  <Button variant="contained" color="secondary"
          className={classes.button}>
    Play Game
  </Button>
</a>

```

The route to the game view takes the game ID as a query parameter. We set `target='_self'` on the link so React Router skips transitioning to the next state and lets the browser handle this link. What this will do is allow the browser to directly make the request to the server at this route when the link is clicked, and render the `index.html` file sent by the server in response to this request, allowing the user to start playing the rendered VR game immediately.

In the final section of the `GameDetail` component, we will conditionally show EDIT and DELETE options only if the currently signed-in user

is also the maker of the game being rendered. We will add these options with the following code:

mern-vrgame/client/game/GameDetail.js:

```
{auth.isAuthenticated().user
  && auth.isAuthenticated().user._id == props.game.maker._id &&
  (<div>
    <Link to={"/game/edit/" + props.game._id}>
      <Button variant="raised" color="primary"
        className={classes.editbutton}>
        Edit
      </Button>
    </Link>
    <DeleteGame game={props.game}
      removeGame={props.updateGames} />
  </div>)}
```

After ensuring the current user is actually authenticated, we check if the user ID of the signed-in user matches the maker ID in the game. Then, accordingly, we render the EDIT button linking to the edit form view, and the DELETE option with a `DeleteGame` component.

The implementation of this `DeleteGame` component is similar to the `DeleteShop` component discussed in [Chapter 7](#), *Exercising MERN Skills with an Online Marketplace*. Instead of a shop, the `DeleteGame` component will take the game to be deleted and the `updateGames` function definition received from the parent component as props. After this implementation is integrated, the maker of a game will be able to remove the game from the platform.

Users visiting the MERN VR Game application can browse through the list of games rendered in these views and select to play a game by clicking the PLAY GAME link rendered in the corresponding `GameDetail` component. In the next section, we will see how to update the server to handle a request to play a game.

Playing the VR game

Users on the MERN VR Game application will be able to open and play any of the VR games from within the application. To enable this, we will add an API on the server that renders the `index.html` file, which was generated with React 360, as discussed in the previous chapter, [chapter 13](#), *Developing a Web-Based VR Game*. This API on the backend will receive a `GET` request at the following path:

```
| /game/play?id=<game ID>
```

This path takes a `game ID` value as a query parameter. The `game ID` in this URL will be used in the React 360 code, as elaborated on later in the chapter, to fetch the game's details using the load game API. In the following section, we will look at the implementation of the backend API that will handle this `GET` request to start playing a game when the user clicks on the PLAY GAME button.

Implementing the API to render the VR game view

In order to implement the API that will render the VR game in the browser, we will add a route in the backend that will receive a `GET` request and open the `index.html` page from React 360.

This route will be declared in `game.routes.js` with the other game routes, as follows:

```
mern-vrgame/server/routes/game.routes.js:
```

```
router.route('/game/play')
  .get(gameCtrl.playGame)
```

A `GET` request received at this route will execute the `playGame` controller method, which will return the `index.html` page in response to the incoming request. The `playGame` controller method will be defined as shown in the following code:

```
mern-vrgame/server/controllers/game.controller.js:
```

```
const playGame = (req, res) => {
  res.sendFile(process.cwd() + '/server/vr/index.html')
}
```

The `playGame` controller method will simply send the `index.html` page placed in the `/server/vr/` folder to the requesting client.

In the browser, this will render the React 360 game code, which needs to fetch the game details from the database using the load game API and render the game world, along with the VR objects that the user can interact with. In the next section, we will see how the game view we built previously with React 360 needs to be updated to load these game details dynamically.

Updating the game code in React 360

With the game backend all set up in the MERN application, we can update the React 360 project code we developed in [Chapter 13](#), *Developing a Web-Based VR Game*, to make it render games directly from the game collection in the database.

We will use the game ID in the link that opens the React 360 application to fetch game details, using the load game API from within the React 360 code. Then, we will set this retrieved game data to the state so that the game loads details from the database instead of the static sample data we used in [Chapter 13](#), *Developing a Web-Based VR Game*. Once the code is updated, we can bundle it again and place the compiled files in the MERN application before trying out the integration, as discussed in the following sections.

Getting the game ID from a link

In order to render the VR game based on the game the user chose to play from the MERN VR Game application, we need to retrieve the corresponding game ID from the link that loads the VR game view. In the `index.js` file of the React 360 project folder, we will update the `componentDidMount` method to first retrieve the game ID from the incoming URL, and then make a fetch call to the load game API, as shown in the following code:

/MERNVR/index.js:

```
componentDidMount = () => {
  let gameId = Location.search.split('?id=')[1]
  read({
    gameId: gameId
  }).then((data) => {
    if (data.error) {
      this.setState({error: data.error});
    } else {
      this.setState({
        vrObjects: data.answerObjects.concat(data.wrongObjects),
        game: data
      });
      Environment.setBackgroundImage(
        {uri: data.world}
      )
    }
  })
}
```

`Location.search` gives us access to the query string in the incoming URL that loads `index.html`. The retrieved query string is `split` to get the `gameId` value from the `id` query parameter attached in the URL. We use this `gameId` value to fetch the game details with the load game API on the backend and set it to the state for the game and `vrObjects` values. To be able to use the load game API in the React 360 project, we will define a corresponding `fetch` method in the project, as discussed in the next section.

Fetching the game data with the load game API

We want to fetch the game data from within the React 360 code. In the React 360 project folder, we will add an `api-game.js` file that will contain a `read` fetch method that makes a call to the load game API on the server using the provided game ID. This `fetch` method will be defined as follows:

/MERNVR/api-game.js:

```
const read = (params) => {
  return fetch('/api/game/' + params.gameId, {
    method: 'GET'
  }).then((response) => {
    return response.json()
  }).catch((err) => console.log(err))
}
export {
  read
}
```

This `fetch` method receives the game ID in the `params` and makes the API call to retrieve the corresponding game from the database. It is used in `componentDidMount` of the React 360 entry component, which is defined in the `index.js` file, to retrieve the game details, as discussed in the previous section.



This updated React 360 code is available in the branch named `dynamic-game-second-edition` on the GitHub repository at github.com/shamahoque/MERNVR/tree/dynamic-game-second-edition.

With the React 360 code updated and capable of retrieving and rendering game details based on the game ID specified in the incoming URL, we can bundle and integrate this updated code with the MERN VR Game application, as discussed in the next section.

Bundling and integrating the updated code

With the React 360 code updated to fetch and render game details dynamically from the server, we can bundle this code using the provided bundle script and place the newly compiled files in the `dist` folder of the MERN VR Game project directory.

To bundle the React 360 code from the command line, go to the React 360 MERNVR project folder and run the following code:

```
| yarn bundle
```

This will generate the `client.bundle.js` and `index.bundle.js` bundle files in the `build/` folder with the updated React 360 code. These files, along with the `index.html` file and `static_assets` folders, need to be added to the MERN VR Game application code, as discussed in [Chapter 13, "Developing a Web-Based VR Game"](#), to integrate the latest VR game code.

With this integration completed, if we run the MERN VR Game application and click the PLAY GAME link on any of the games, it should open up the game view with the details of the specific game rendered in the VR scene, and allow interaction with the VR objects, as specified in the gameplay.

Summary

In this chapter, we integrated the capabilities of the MERN stack technologies with React 360 to develop a dynamic VR game application for the web.

We extended the MERN skeleton application to build a working backend that stores VR game details and allows us to make API calls to manipulate these details. We added React views that let users modify games and browse through the games, with the option to launch and play the VR game at a specified route rendered directly by the server.

Finally, we updated the React 360 project code to pass data between the MERN application and the VR game view, by retrieving query parameters from the incoming URL, and using fetch to retrieve data with the game API.

This integration of the React 360 code with the MERN stack application produced a fully functioning and dynamic web-based VR game application, demonstrating how MERN stack technologies can be used and extended to create unique user experiences. You can apply the capabilities revealed here to build your own VR-infused full-stack web applications.

In the next chapter, we will reflect on the full-stack MERN applications built in this book, discussing not just the best practices that were followed but also the scope for improvements and further development.

Going Forward with MERN

In this part, we wrap up the lessons covered in the book with additional concepts that can further improve MERN stack application development.

This section comprises the following chapter:

[chapter 15](#), *Following Best Practices and Developing MERN Further*

Following Best Practices and Developing MERN Further

In this chapter, we will elaborate on some of the best practices to apply when building the six MERN applications in this book. Additionally, we will explore other practices that we have not applied in this book but that should be considered for real-world applications to ensure reliability and scalability as complexity grows. We will review the decisions behind organizing the project code in modules, the approaches to applying frontend styling, server-side rendering with data only for selective views, and how React interfaces may be composed to manage state across components. We will also look at ways to improve security, add testing to the projects, and optimize bundling with webpack. Finally, we will wrap up with suggestions for enhancing, and steps for extending, the applications built. With these insights, you will be better equipped to prepare your full-stack MERN projects for the real world.

The topics covered in this chapter include the following:

- Separation of concerns with modularity in the application structure
- Considering the options for CSS styling solutions
- Server-side rendering with data for selected views
- Using ES6 classes for stateful versus purely functional components
- Deciding on whether to use Redux or Flux
- Security enhancements for storing user credentials
- Writing test code
- Optimizing bundle sizes
- How to add new features to existing applications

Separation of concerns with modularity

While building the MERN stack applications in this book, we followed a common folder structure across each application. We employed a modular approach by dividing and grouping the code based on relevance and common functionality. The idea behind creating these smaller and distinct sections in the code is to make sure each section addresses a separate concern, so individual sections can be reused, as well as developed and updated independently. In the following section, we will review this structure and its benefits.

Revisiting the application folder structure

In the application folder structure, we kept the client-side and server-side code separate with further subdivisions within these two sections. This gave us some freedom to design and build the frontend and backend of the application independently. At the project root level, the `client` and `server` folders were the main divisions, as shown in the following structure:

```
| mern_application/  
| -- client/  
| -- server/
```

In these `client` and `server` folders, we divided the code further into subfolders that mapped to unique functionalities. We did this by dividing models, controllers, and routes in the server for specific features, and grouping all components related to a feature in one place on the client side. In the following sections, we will review the divisions within the `server` and `client` folders.

Server-side code

On the server side, we divided the code according to functionality, by separating code that defines business models from code implementing routing logic, and controller code that responds to client requests at these routes. Within the `server` folder, we maintained three main sections, as shown in the following structure:

```
| -- server/  
|   --- controllers/  
|   --- models/  
|   --- routes/
```

In this structure, each folder contains code with a specific purpose:

- `models`: This folder is meant to contain all of the Mongoose schema model definitions in separate files, with each file representing a single model.
- `routes`: This folder contains all routes that allow the client to interact with the server, with routes placed in separate files that may be associated with a model in the `models` folder.
- `controllers`: This folder contains all of the controller functions that define logic to respond to incoming requests at the defined routes. These controllers are divided into separate files corresponding to the relevant model and route files.

As demonstrated throughout the book, these specific separations of concerns for the code on the server side allowed us to extend the server developed for the skeleton application by just adding the required model, route, and controller files. In the next section, we will go over the divisions in the client-side code structure.

Client-side code

The client-side code for the MERN applications consists primarily of React components. In order to organize the component code and related helper code in a reasonable and understandable manner, we separated the code into folders related to a feature entity or unique functionality, as shown in the following structure:

```
| -- client/  
|   --- auth/  
|   --- core/  
|   --- post/  
|   --- user/  
|   --- componentFolderN/
```

In the preceding structure, we placed all of the auth-related components and helper code in the `auth` folder; common and basic components, such as the `Home` and `Menu` components, in the `core` folder; and then we made `post` and `user` folders for all of the post-related or user-related components in the respective folders.

This separation and grouping of components based on features allowed us to extend the frontend views in the skeleton application for each application that followed, by adding a new feature-related component code folder, as required, to the `client` folder.

Separating the client and server code, and also modularizing the code within these divisions, made it easier to extend the different applications we developed throughout the book. In the final section of this chapter, we will further demonstrate the advantages of this modularized approach of separating the application code, as we outline the general workflow that can be followed to add a new feature to any of the existing applications developed in this book. In the next section, we will explore the different options available for defining and applying styling to the frontend React components, which will be a necessary decision for every full-stack MERN project.

Adding CSS styles

When discussing **user interface (UI)** implementations for the applications in this book, we chose not to focus on the details of the CSS styling code applied and instead relied mostly on the default Material-UI stylings. However, given that implementing any UI requires us to consider styling solutions, we will briefly look at some of the options that are available.

When it comes to adding CSS styles to the frontend, there are a number of options, each with pros and cons. In the following sections, we will discuss the two most common options, which are external style sheets and inline styles, along with the relatively newer approach of writing CSS in JavaScript, or, more specifically, JSS, which is used in Material-UI components and hence also for the applications in this book.

External style sheets

External style sheets allow us to define CSS rules in separate files, which can be injected into the necessary view. Placing CSS styles in external style sheets this way was once considered the best practice because it enforced the separation of style and content, allowing reusability and also maintaining modularity if a separate CSS file was created for each component.

However, as web development technologies continue evolving, the demands of better CSS organization and performance are no longer met by this approach. For example, using external style sheets while developing frontend views with React components limits our control over updating styles based on the component state. Moreover, loading external CSS for React applications requires additional webpack configurations with `css-loader` and `style-loader`.

When applications grow and share multiple style sheets, it also becomes impossible to avoid selector conflicts because CSS has a single global namespace. Hence, though external style sheets may be enough for simple and trivial applications, as an application grows, other options for using CSS become more relevant. In the next section, we will look at the option of adding styles directly inline.

Inline styles

Inline CSS is a style defined and applied directly to individual elements in the view. Although this takes care of some of the problems faced when using external style sheets, such as eliminating the issue of selector conflicts and allowing state-dependent styles, it takes away reusability and introduces a few problems of its own, such as limiting the CSS features that can be applied.

Using only inline CSS for a React-based frontend has important limitations for growing applications, such as poor performance because all of the inline styles are recomputed at each render, and inline styles are slower than class names, to begin with.

Inline CSS may seem like an easy fix in some cases, but it does not serve as a good option for overall usage. In the next section, we will explore the option to add CSS styles using JavaScript, which addresses some of the issues of using inline and external styles.

JavaScript Style Sheets (JSS)

JSS allows us to write CSS styles using JavaScript in a declarative way. This also means that all the features of JavaScript are now available for writing CSS, making it possible to write reusable and maintainable styling code.

JSS works as a JS to CSS compiler that takes JS objects, where keys represent class names and values represent corresponding CSS rules, and then generates the CSS along with scoped class names.

In this way, JSS generates unique class names by default when it compiles JSON representations to CSS, eliminating the chances of selector conflicts that could be faced with external style sheets. Moreover, unlike inline styles, the CSS rules that are defined with JSS can be shared across multiple elements and all CSS features can be used in the definitions.

Material-UI uses JSS to style its components, and, as a result, we used JSS to apply Material-UI themes and also custom CSS to the components developed for the frontend views in all of the applications. Based on the utility of each approach, you can choose to use one or a combination of external style sheets, inline styles, or JSS for styling the frontend of your full-stack application. In the next section, we will review the approaches to and relevance of incorporating server-side rendering of the React frontend in a full-stack MERN application.

Selective server-side rendering with data

When we developed the frontend of the base skeleton application in [Chapter 4, Adding a React Frontend to Complete MERN](#), we integrated basic server-side rendering in order to load client-side routes directly from the browser address bar when the request went to the server. In this server-side rendering implementation, while rendering the React component's server-side, we did not consider loading the data from the database for the components that displayed data. The data only loads in these components when the client-side JavaScript takes over after the initial load of the server side-rendered markup.

We did update this implementation to add server-side rendering with data for the individual media detail pages in the MERN Mediastream application, which was discussed in [Chapter 12, Customizing the Media Player and Improving the SEO](#). In this case, we decided to render this specific view with data by injecting data into the server side-generated markup of the React frontend. The reasoning behind this selective server-side rendering with data only for specific views can be based on certain desired behaviors for the view in question, as discussed in the following section.

When is server-side rendering with data relevant?

Implementing server-side rendering with data for all of the React views in an application can get complicated, and will be additional work if we need to consider views with client-side authentication or views consisting of multiple data sources. In many cases, it may be unnecessary to tackle these complexities if the view does not require server-side rendering with data. In order to judge whether a view needs to be server-rendered with data, answer the following questions for the specific view to make your decision:

- Is it important for the data to be displayed in the initial load of the view when JavaScript may not be available in the browser?
- Do the view and its data need to be SEO-friendly?

Loading data in the initial load of the page may be relevant from a usability perspective, so it really depends on the use case for the specific view. For SEO, server-side rendering with data will give search engines easier access to the data content in the view; so, if this is crucial for the view in question, then adding server-side rendering with data is a good idea. In the next section, we will go over the varied approaches of composing the React frontend in a full-stack application.

Using stateful versus pure functional components

While building a UI with React components, composing the views with more stateless functional components can make the frontend code manageable, clean, and easier to test. However, some components will require the state or life cycle Hooks to be more than pure presentational components. In the following sections, we will look at what it takes to build stateful and stateless functional React components, when to use one or the other, and how often.

Stateful React components with ES6 classes or Hooks

We can define stateful React components with ES6 classes or by using Hooks without writing a class. React components defined using ES6 classes have access to life cycle methods, the `this` keyword, and can manage state with `setState` when building stateful components. Similarly, React components defined with a function can also access some of these features using Hooks, such as managing state with the `useState` Hook, in order to build stateful components.

Stateful components allow us to build interactive components that can manage to change data in the state, and propagate any business logic that needs to be applied across the UI. Generally, for complex UIs, stateful components should be higher-level container components that manage the state of the smaller, stateless functional components they are composed of. In comparison, these simpler stateless components can be defined as pure functions, as discussed in the next section.

Stateless React components as pure functions

React components can be defined as stateless functional components using the ES6 class syntax or as pure functions. The main idea is that a stateless component does not modify state and only receives props.

The following code defines a stateless component using the ES6 class syntax:

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

This component, although defined with a class, does not use state. The same component can also be defined using JavaScript pure functions, as follows:

```
function Greeting(props) {
  return <h1>Hello, {props.name}</h1>
}
```

A pure function always gives the same output when given the same input without any side effects. Modeling React components as pure functions enforces the creation of smaller, more defined, and self-contained components that emphasize UI over business logic as there is no state manipulation in these components. These kinds of components are composable, reusable, and easy to debug and test. In the next section, we will discuss how to combine stateful and stateless components when designing the UI.

Designing the UI with stateful components and stateless functional components

When thinking about the component composition for a UI, you can design the root or a parent component as a stateful component that will contain child components or as the composable components that only receive props and cannot manipulate state. All the state-changing actions and life cycle issues will be handled by the root or parent component, and the changes will be propagated to the child components.

In the applications developed for this book, there is a mixture of stateful higher-level components and smaller stateless components. For example, in the MERN Social application, the `Profile` component modifies the state for stateless child components, such as the `FollowProfileButton` and `FollowGrid` components. There is scope for refactoring some of the larger components that were developed in this book into smaller, more self-contained components, and this should be considered before extending the applications to incorporate more features.

The main takeaway that can be applied to new component designs, or when refactoring existing components, is that as the React application grows and gets more complex, it is better to have more stateless functional components added to higher-level stateful components that are in charge of managing state for the inner components. In the next section, we will briefly discuss popular libraries and patterns that can be utilized on top of React to handle state management across growing React applications.

Using Redux or Flux

When React applications begin to grow and become more complex, managing communication between components can become problematic. When using regular React, the way to communicate is to pass down values and callback functions as props to the child components. However, this can be tedious if there are a lot of intermediary components that the callback must pass through. To address these state communication and management-related issues as the React application grows, people turn to use React with libraries and architecture patterns such as Redux and Flux.

It is outside the scope of this book to delve into the details of integrating React with the Redux library or the Flux architecture, but you can consider these options for their growing MERN applications while keeping the following in mind:

- Redux and Flux utilize patterns that enforce changing states in a React application from a central location. A trick to avoid using Redux or Flux in React applications of manageable sizes is to move all state changes up the component tree to the parent components.
- Smaller applications work just as well without Flux or Redux.



You can learn more about using React with Redux at <https://redux.js.org/>, and about using React with Flux at <facebook.github.io/flux/>.

You can choose to integrate Flux or Redux based on your application size and complexity. In the next section, we will discuss the security implementations applied to the MERN applications developed in this book and the possible enhancements that could be made.

Enhancing security

In the MERN applications developed for this book, we kept the auth-related security implementations simple by using **JSON web tokens (JWTs)** as an authentication mechanism and by storing hashed passwords in the user collection. The approaches followed in these implementations are standard practices for adding authentication to a web application. However, there are advanced options available for adding more layers of security, if that is required for certain applications. In the following sections, we will go over the security choices made for building the applications in this book and point to possible enhancements.

JSON web tokens – client-side or server-side storage

With the JWT authentication mechanism, the client side becomes responsible for maintaining the user state. Once the user signs in, the token sent by the server is stored and maintained by the client-side code on browser storage, such as `sessionStorage`. Hence, it is also up to the client-side code to invalidate the token by removing it when a user signs out or needs to be signed out. This mechanism works out well for most applications that require minimal authentication to protect access to resources. However, for instances where it may be necessary to track user sign-ins, sign-outs, and to let the server know that a specific token is no longer valid for signing in, just the client-side handling of the tokens is not enough.

For these cases, the implementation discussed for handling JWT tokens on the client side can be extended to storage on the server side as well. In the specific case of keeping track of invalidated tokens, a MongoDB collection can be maintained by the server to store these invalidated tokens as a reference, which is moderately similar to how it is done for storing session data on the server side.

The thing to be cautious about and to keep in mind is that storing and maintaining auth-related information on both the client and server side may be overkill in most cases. Therefore, it is entirely up to the specific use case and the related trade-offs to be considered. In the next section, we will review our options for storing user passwords securely.

Securing password storage

While storing user credentials for authentication in the user collection, we made sure that the original password string provided by the user was never stored directly in the database. Instead, we generated a hash of the password along with a salt value using the `crypto` module in Node.

In `user.model.js` from our applications, we defined the following functions to generate the hashed `password` and `salt` values:

```
encryptPassword: function(password) {
  if (!password) return ''
  try {
    return crypto
      .createHmac('sha1', this.salt)
      .update(password)
      .digest('hex')
  } catch (err) {
    return ''
  }
},
makeSalt: function() {
  return Math.round((new Date().valueOf() * Math.random())) + ''
}
```

With this implementation, every time a user enters a password to sign in, a hash is generated with the salt. If the generated hash matches the stored hash, then the password is correct; otherwise, the password is wrong. So, in order to check whether a password is correct, the salt is required, and therefore it is stored with the user details in the database along with the hash.

This is the standard practice for securing passwords stored for user authentication, but there are other advanced approaches that may be explored if a specific application's security requirements demand it. Some options that can be considered include multi-iteration hashing approaches, other secure hashing algorithms, limiting the number of login attempts per user account, and multi-level

authentication with additional steps, such as answering security questions or entering security codes. These options can add more layers of security as needed. In the next section, we will discuss options for adding test code in full-stack React applications, which is essential for building sturdy production-ready applications.

Writing test code

Though discussing and writing test code is outside the scope of this book, it is a crucial part of developing reliable software. As full-stack JavaScript applications become more mainstream over time, the need for better testing capabilities is producing a good number of testing tools in this ecosystem. In the following sections, we will first look at some of the popular testing tools that are available for testing the different parts of a MERN-based application. Then, to help you get started with writing test code for the MERN applications developed in this book, we will also discuss an example of adding a client-side test to the MERN Social application from [Chapter 5, *Growing the Skeleton into a Social Media Application*](#).

Testing tools for full-stack JavaScript projects

A whole range of testing tools is available for incorporating testing and maintaining code quality in full-stack JavaScript projects. These include tools that can help with performing static analysis on the code to maintain readability, and with integrating unit testing, integration testing, and end-to-end testing in MERN-based applications. In the following sections, we will highlight a few of these popular testing tools that can be used with the projects in this book, such as ESLint for static analysis, Cypress for frontend testing, and Jest for comprehensive testing in JavaScript applications.

Static analysis with ESLint

A good practice for improving and maintaining code quality is to use a linting tool with your project. Linting tools perform static analysis on the code to find problematic patterns or behaviors that violate specified rules and guidelines. Linting code in a JavaScript project can improve overall code readability and also help you to find syntax errors before the code is executed. For linting in MERN-based projects, you can explore ESLint, which is a JavaScript linting utility that allows developers to create their own lint rules.



You can learn more about using and customizing ESLint at eslint.org. You can choose to use the Airbnb JavaScript Style Guide (github.com/airbnb/javascript) to define your lint rules with `eslint-config-airbnb`.

You can configure ESLint in your preferred editor and make it a seamless part of your development workflow. This will help you to maintain standards in your code while you are writing it. In the next section, we will take a look at Cypress, which can help to test any code that runs in the browser.

End-to-end testing with Cypress

Cypress provides a complete set of tools for testing the frontend of modern web applications. Using Cypress, we can write end-to-end tests, unit tests, and integration tests for the frontend of our MERN-based applications. Cypress also provides its own locally installed test runner, allowing us to write and run tests, and debug in real time in the browser as we build the application.



You can learn more about using Cypress at cypress.io to get started with setting up end-to-end testing for JavaScript applications in the browser.

Performing UI testing with Cypress will allow you to ship out your projects more confidently, as you will be able to catch more bugs early on before they are encountered by the end users of the application. In the next section, we will discuss Jest, which can be used to add tests to any JavaScript code base.

Comprehensive testing with Jest

Jest is a comprehensive testing framework for JavaScript. Although it has been more commonly known for testing React components, it can be used for general-purpose testing with any JavaScript library or framework. Among the many JavaScript testing solutions in Jest, it provides support for mocking and snapshot testing, comes with an assertion library, and tests in Jest are written in the **Behavior-Driven Development (BDD)** style.



To learn more about Jest, read the documentation at <https://facebook.github.io/jest/docs/en/getting-started.html>.

Besides testing the React components, Jest can also be adapted to write test code for the Node-Express-Mongoose-based backend as required. Hence, it is a solid testing option to add test code for MERN applications. In the next section, we will explore how you can use Jest to add a test to the MERN Social application, which was developed in [Chapter 5](#), *Growing the Skeleton into a Social Media Application*.

Adding a test to the MERN Social application

In order to demonstrate how to get started with adding tests to MERN applications, we will set up Jest and use it to add a client-side test to the MERN Social application. Before defining a test case, followed by writing and running the corresponding test code, first, we will set up for testing by installing the necessary packages, defining the test run script, and creating a folder for the test code, as discussed in the following sections.

Installing the packages

In order to set up Jest and integrate the test code with our projects, we first need to install the relevant Node packages. The following packages will be required in order to write the test code and run the tests:

- `jest`: To include the Jest testing framework
- `babel-jest`: To compile JS code for Jest
- `react-test-renderer`: To create a snapshot of the DOM tree rendered by a React DOM without using a browser

To install these packages as `devDependencies`, run the following `yarn` command from the command line:

```
| yarn add --dev jest babel-jest react-test-renderer
```

Once these packages are installed, we can start adding tests after configuring the test runner script, as discussed in the next section.

Defining the script to run tests

In order to run any test code that we write using Jest, we will define a script command to run the tests. We will update the run scripts defined in `package.json` in order to add a script for running tests with the `jest` command, as shown in the following code:

```
| "scripts": {  
|   "test": "jest"  
| }
```

With this script defined, if we run `yarn test` from the command line, it will prompt Jest to find the test code in the application folders and run the tests. In the next section, we will add the folder that will contain the test code files for the project.

Adding a tests folder

To add the client-side test in the MERN Social application, we will create a folder, called `tests`, in the client folder, which will contain test files relevant to testing the React components. When the test command is run, Jest will look for the test code in these files.

The test case for this example will be a test on the `Post` component in the frontend of the MERN Social application, and we will add tests for the `Post` component in a file called `post.test.js`. This file will be placed in the `tests` folder. Now that we have a file ready for adding the test code, in the next section, we will demonstrate how to add an example test case.

Adding the test

For the MERN Social application, we will write a test to check whether the delete button on a post is only visible when the signed-in user is also the creator of the post. This means that the delete button will only be a part of the rendered Post view if the `_id` of the authenticated user is the same as the `postedby` value of the post data being rendered.

In order to implement this test case, we will add code that takes care of the following:

- Defines dummy data for a post and an `auth` object containing authenticated user details
 - Mocks the methods in `auth-helper.js`
 - Defines the test, and, within the test definition, does the following:
 - Declares the `post` and `auth` variables
 - Sets the return value of the mocked `isAuthenticated` method to the dummy `auth` object
 - Uses `renderer.create` to create the `Post` component with the required dummy props passed and wrapped in `MemoryRouter` to provide the props related to `react-router`
 - Generates and matches snapshots

The code in `post.test.js` to incorporate the steps described for this specific test will be as follows:

```

        "created":"2017-12-22T07:20:25.611Z",
        "comments":[], "likes":[]}
const dummyAuthObject = {user: {_id:"5a3cb1779bcc621874d7e428",
                                "name":"Joe",
                                "email":"abc@def.com"}}

test('delete option visible only to authorized user', () => {
  const post = dummyPostObject
  const auth = dummyAuthObject

  auth.isAuthenticated.mockReturnValue(auth)

  const component = renderer.create(
    <MemoryRouter>
      <Post post={post} key={post._id} ></Post>
    </MemoryRouter>
  )

  let tree = component.toJSON()
  expect(tree).toMatchSnapshot()
})

```

In this code, we first defined dummy posts and `auth` objects, and then added the test case for checking the visibility of the delete option. In this test case, we mocked the `isAuthenticated` method and rendered the `Post` component using the dummy post data. Then, we generated a snapshot with this rendered component, which will be matched with the expected snapshot. In the next section, we will discuss how generated snapshots are compared in this test.

Generating a snapshot of the correct Post view

The first time this test is run, we will provide it with the values required to generate the correct snapshot of the Post view. The correct snapshot for this test case will contain the delete button when the `user._id` of the `auth` object is equal to the `postedBy` value of the `post` object. This snapshot is generated when the test is run for the first time, and it will be used for comparison in future test executions.

This kind of snapshot testing in Jest basically records snapshots of rendered component structures to compare them to future renderings. When the recorded snapshot and the current rendering don't match, the test fails, indicating that something has changed. In the next section, we will go over the steps of running the test and checking the test output.

Running and checking the test

In the code that we added to the `post.test.js` file, the dummy `auth` object and the `post` object refer to the same user; therefore, running this test in the command line will prompt Jest to generate a snapshot that will contain the delete option and also pass the test.

To run the test, go into the project folder from the command line:

```
| yarn test
```

The test output generated when this command runs will show that the test passed, as portrayed in the following screenshot:

```
> mern-social@1.0.0 test /Users/shoque/packtBook/mern-social
> jest

PASS client/tests/post.test.js
  ✓ Test for Post (112ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        1.436s, estimated 2s
Ran all test suites.
```

The recorded snapshot that is generated, when this test runs successfully for the first time, is added automatically to a `_snapshots_` folder in the `tests` folder. This snapshot represents the state where the delete button is rendered in the view since the authenticated user is also the creator of the post.

We can now check whether the test actually fails when the component is rendered with an authenticated user that is not the creator of the post. To perform this check, we will update the dummy data objects by changing the `user._id`, so it does not match the

`postedBy` value, and then run the test again. This will give us a failed test, as the current rendering will no longer have a delete button that is present in the recorded snapshot.

As shown in the following test log, the test fails and indicates that the rendered tree does not match the recorded snapshot since the elements representing the delete button are missing in the received value:

```
> mern-social@1.0.0 test /Users/shoque/packtBook/mern-social
> jest

 FAIL client/tests/post.test.js
 × Test for Post (175ms)

● Test for Post

  expect(value).toMatchSnapshot()

    Received value does not match stored snapshot 1.

      - Snapshot
      + Received

      @@ -37,51 +37,10 @@
          className="MuiTypography-root-45 MuiTypography-body2-53
lorSecondary-66 MuiCardHeader-subheader-40"
        >
          Fri Dec 22 2017
          </span>
        </div>
      - <div
      -   className="MuiCardHeader-action-37"
      - >
      -   <button
      -     aria-label="delete"
      -     className="MuiButtonBase-root-78 MuiIconButton-root-69"
      -     disabled={false}
      -     onBlur={[Function]}>
```

We have a client-side test for checking whether a signed-in user can view the delete button on their posts. Using this setup, more tests can be added for the MERN application utilizing the capabilities of Jest.

Writing test code will make the application you develop reliable and also help ensure code quality. Using tools such as ESLint, Cypress,

and Jest, we can incorporate different ways of ensuring the overall quality of MERN-based applications. In the next section, we will move on to discussing ways to optimize the bundling of the application code.

Optimizing the bundle size

As you develop and grow a MERN application, chances are the size of the bundles produced with webpack will also grow, especially if large third-party libraries are used. Larger bundle sizes will affect performance and increase the initial load time of the application. We can make changes in the code to ensure we don't end up with large bundles and also utilize features packed in webpack to help optimize bundling.



Before going into the code to update it for bundle size optimization, you can also get familiar with the default optimization options that are part of webpack. In the MERN applications, we used the `mode` config to utilize the default settings for both development and production mode. To view an overview of the options that are available, please refer to the article at <https://medium.com/webpack-4-mode-and-optimization-5423a6bc597a>.

In the following section, we will highlight concepts such as code splitting and dynamic imports, which can give us control over producing smaller bundles and decreasing load time.

Code splitting

Instead of loading all the code at once in one bundle, we can use the code splitting feature supported by webpack to lazy-load parts of the application code as currently needed by the user. After we modify the application code to introduce code-splitting, webpack can create multiple bundles rather than one large bundle. These bundles can be loaded dynamically at runtime, allowing us to improve the performance of the application.



To learn more about code splitting support in webpack and how to make necessary changes to the setup and configuration, take a look at the guidelines in the documentation at <https://webpack.js.org/guides/code-splitting/>.

There are several ways to introduce code splitting for the application code, but the most important syntax you will come across for this purpose is the dynamic `import()`. In the next section, we will look at how to use `import()` with our MERN applications.

Dynamic import()

Dynamic `import()` is a function-like version of the regular import, and it enables the dynamic loading of JS modules. Using

`import(moduleSpecifier)` will return a promise for the module namespace object of the requested module. When using regular static imports, we import a module at the top of the code and then use it in the code as follows:

```
import { convert } from './metric'  
...  
console.log(convert('km', 'miles', 202))
```

In contrast, if we were to use dynamic `import()` instead of adding the static import at the beginning, the code would look like this:

```
import('./metric').then(({ convert }) => {  
    console.log( convert('km', 'miles', 202) )  
})
```

This allows us to import and load the module when the code requires it. While bundling the application code, webpack will treat calls to `import()` as split points and automatically start code splitting by placing the requested module and its children into a separate chunk from the main bundle.

In order to optimize the bundling of the frontend React code by applying code splitting at a given component, we need to pair dynamic `import()` with React Loadable – a higher-order component for loading components with promises. As an example, we will look at the shopping cart developed in [Chapter 8, Extending the Marketplace for Orders and Payments](#). While building the interface of the cart, we composed the `Cart` component by importing and adding the `Checkout` component to the view, as follows:

```
import Checkout from './Checkout'  
class Cart extends Component {  
    ...
```

```
    render() {
      ...
      <Checkout/>
    }
  ...
}
```

To introduce code splitting here and import the `Checkout` component dynamically, we can replace the static import at the beginning with a `Loadable` `Checkout`, as shown in the following code:

```
import Loadable from 'react-loadable'
const Checkout = Loadable({
  loader: () => import('./Checkout'),
  loading: () => <div>Loading...</div>,
})
```

Making this change and using webpack to build the code again will produce a `bundle.js` file of reduced size, and generate another smaller bundle file representing the split code, which will now only load when the `Cart` component is rendered.



Route-based code splitting is also another option besides using dynamic imports. It can be an effective approach for introducing code splitting in React apps that use routes to load components in the view. To learn more about implementing code splitting, specifically with React Router, view the article at <https://tylermcginnis.com/react-router-code-splitting/>.

We can apply code-splitting mechanisms across our application code as required. The thing to keep in mind is that effective code splitting will depend on using it correctly and applying it at the right places in the code – places that will benefit in optimization from resource-load prioritization. In the next section, we will outline the steps that can be repeated to add new features to the MERN applications developed in this book.

Extending the applications

Throughout the chapters of this book, as we developed each application, we added features by extending the existing code in a common and repeatable number of steps. In this final section, we will review those steps, setting a guideline for adding more features to the current versions of the applications.

Extending the server code

For a specific feature that will require data persistence and APIs to allow the views to manipulate the data, we can start by extending the server code and adding the necessary models, routes, and controller functions, as outlined in the following sections.

Adding a model

For the data persistence aspect of the feature, design the data model considering the fields and values that need to be stored. Then, define and export a Mongoose schema for this data model in a separate file, and place it in the `server/models` folder. With the data structure defined and ready for the database, you can move on to adding the API endpoints for manipulating this data, as discussed next.

Implementing the APIs

In order to manipulate and access the data that will be stored in the database based on the model, you need to design the APIs relevant for the desired feature. To start implementing the APIs, you have to add the corresponding controller methods and route declarations, as discussed in the following sections.

Adding controllers

With the APIs decided, add the corresponding controller functions that will respond to the requests to these APIs in a separate file in the `server/controllers` folder. The controller functions in this file should access and manipulate the data for the model defined for this feature. Next, we will look at how to declare the routes that will invoke these controller methods when the requests come in.

Adding routes

To complete the implementation of the backend APIs, corresponding routes need to be declared and mounted on the Express app. In a separate file in the `server/routes` folder, first, declare and export the routes for these APIs, assigning the relevant controller functions that should be executed when a specific route is requested. Then, load these new routes on the Express app in the `server/express.js` file, just like the other existing routes in the application.

This will produce a working version of the new backend APIs that can be run and checked from a REST API client application. Then, these APIs can be used in the frontend views for the feature being developed, which you will add by extending the client code, as discussed in the next section.

Extending the client code

On the client side, first, design the views required for the feature, and determine how these views will incorporate user interaction with the data relevant to the feature. Then, add the fetch API code to integrate with the new backend APIs, define the new components that represent these new views, and update the existing code to include these new components in the frontend of the application, as outlined in the following sections.

Adding the API fetch methods

Before adding the fetch methods that will make calls to the backend APIs, you will determine a location for placing the new frontend code. In the client folder, create a new folder to house the components and helper code relevant to the feature module being developed. Then, to integrate the new backend APIs with the frontend of the application, define and export the corresponding fetch methods in a separate file in this new components folder. Finally, you can populate this folder with the React components that will be the frontend of this feature, as discussed in the next section.

Adding components

To start adding the UI for the feature, you can create and export new React components that represent views for the desired feature in separate files in the new folder. If authentication is required, you can integrate it into these new components using the existing auth-helper methods. Once the React components are implemented, they need to be loaded into the main application view, as discussed in the next section.

Loading new components

In order to incorporate these new components into the frontend, the components either need to be added into existing components or rendered at their own client-side routes.

If these new components need to be rendered at individual routes, update the `MainRouter.js` code to add new routes that load these components at given URL paths. Then, these URLs can be used as links to load the components from other views in the application, or directly by visiting the URL from the browser address bar.

However, if the new components need to become part of existing views, then import the components into the existing components to add them to the view as desired.

The new components can also be linked with existing components, such as in the `Menu` component, by linking to new components that were added with individual routes.

With the components integrated and connected to the backend, the new feature implementation is complete. These steps can be repeated to add on even more new features to the existing MERN-based applications built throughout this book.

Summary

In this final chapter, we reviewed and elaborated on some of the best practices that we used while building the MERN applications in this book, highlighted areas of improvement, gave pointers to address issues that may crop up when applications grow, and, finally, set down steps to continue developing more features into the existing applications.

We saw that modularizing the application's code structure helped to extend the application easily, choosing to use JSS over inline CSS and external style sheets kept the styling code contained and easy to work with, and only implementing server-side rendering for specific views as required kept unnecessary complications out of the code.

We discussed the benefits of creating fewer stateful components that are composed of smaller and more defined stateless functional components, and how this can be applied while refactoring existing components or designing new components to extend the applications. For growing applications that may run into issues with managing and communicating state across hundreds of components, we pointed to options such as Redux and Flux, which may be considered to address these issues.

For applications that may have higher demands for stricter security enforcement, we looked back at our existing implementation of user authentication with JWT and password encryption and discussed possible extensions for improved security.

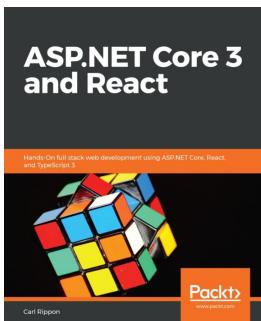
We highlighted testing tools such as ESLint, Cypress, and Jest. Then, we used Jest to demonstrate how test code can be added to the MERN applications and discussed how good practices, such as writing test code and using a linting tool, can improve code quality besides ensuring reliability in an application.

We also looked at bundle optimization features, such as code splitting, that can help to improve performance by reducing the initial bundle size, and by lazy-loading parts of the application as required.

Finally, we reviewed and set down the repeatable steps that were used throughout the book, which you can use as a guideline moving forward to extend the MERN applications by adding more features as desired.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



ASP.NET Core 3 and React

Carl Rippon

ISBN: 978-1-78995-022-9

- Build RESTful APIs with .NET Core using API controllers
- Create strongly typed, interactive, and function-based React components using Hooks
- Build forms efficiently using reusable React components
- Perform client-side state management with Redux and the React Context API
- Secure REST APIs with ASP.NET identity and authorization policies
- Run a range of automated tests on the frontend and backend
- Implement continuous integration (CI) and continuous delivery (CD) processes into Azure using Azure DevOps



Hands-on Full-Stack Web Development with GraphQL and React

Sebastian Grebe

ISBN: 978-1-78913-452-0

- Resolve data from multi-table database and system architectures
- Build a GraphQL API by implementing models and schemas with Apollo and Sequelize
- Set up an Apollo Client and build front end components using React
- Use Mocha to test your full-stack application
- Write complex React components and share data across them
- Deploy your application using Docker

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt.
Thank you!