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 **auth**entication-**auth**orization (**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.

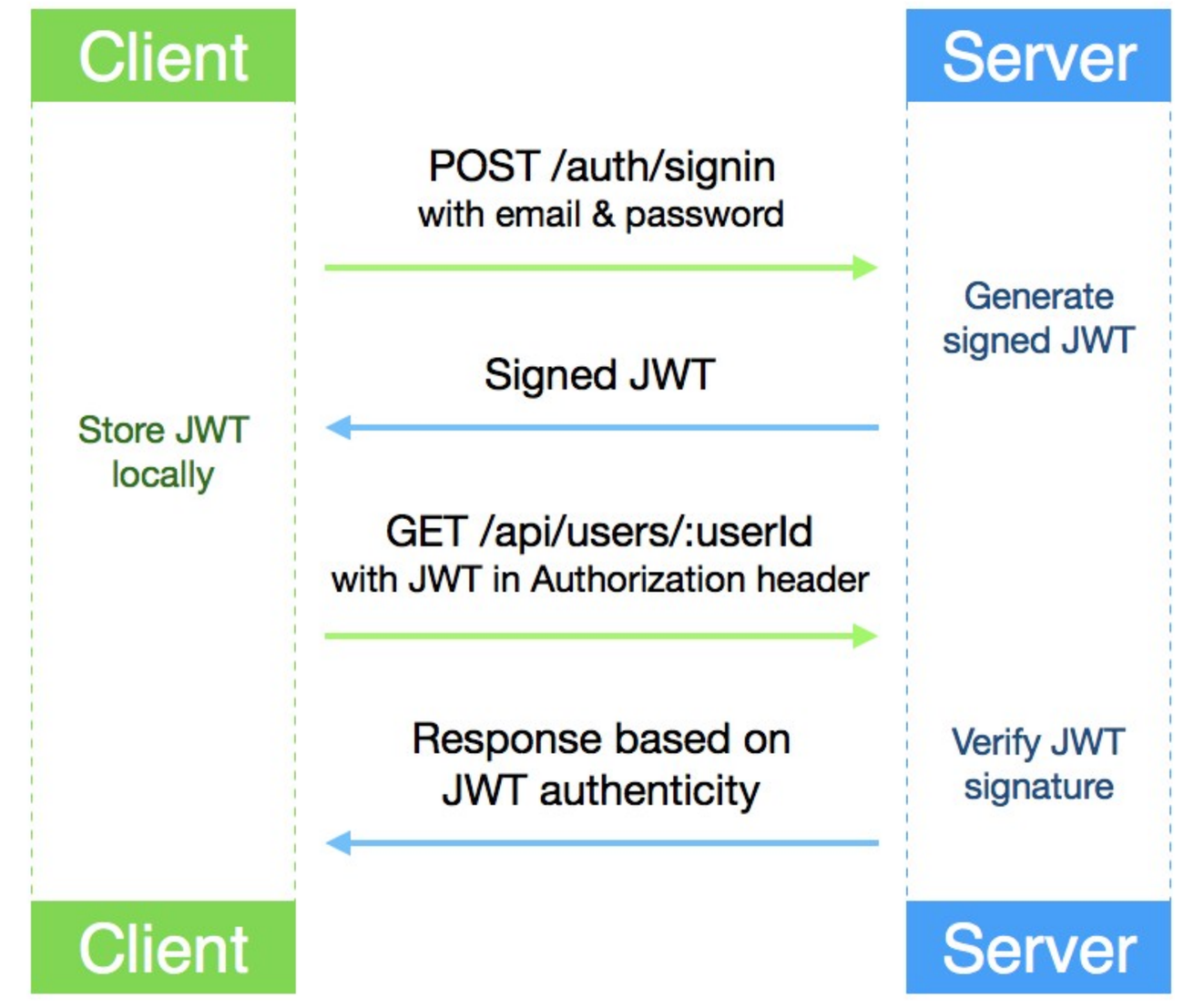
Auth with JSON Web Tokens

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:



JWT Auth Flow

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 request `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, stand-alone 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_pass
  },
  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 ch
  }
  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.sl
      ' 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 cr
    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 time-stamp. 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, hasAuthorizati
```

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.messa
  }else if (err) {
    res.status(400).json({"error" : err.name + ": " + err.messa
    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

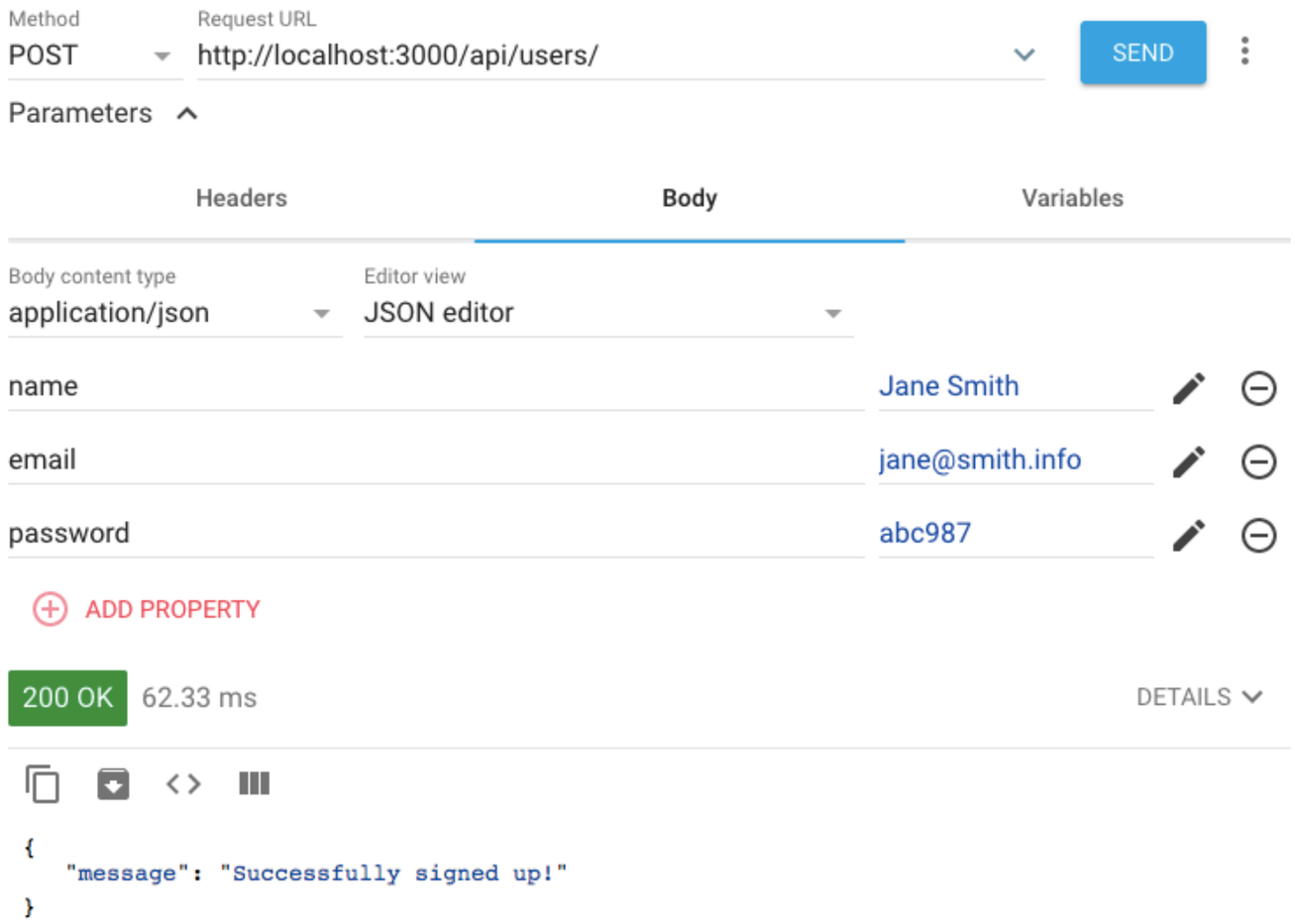
There are a number of options when it comes to selecting tools to check backend APIs, ranging from the command-line tool curl (https://github.com/curl/curl) to **Advanced REST Client** (**ARC**) (https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo), a Chrome 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

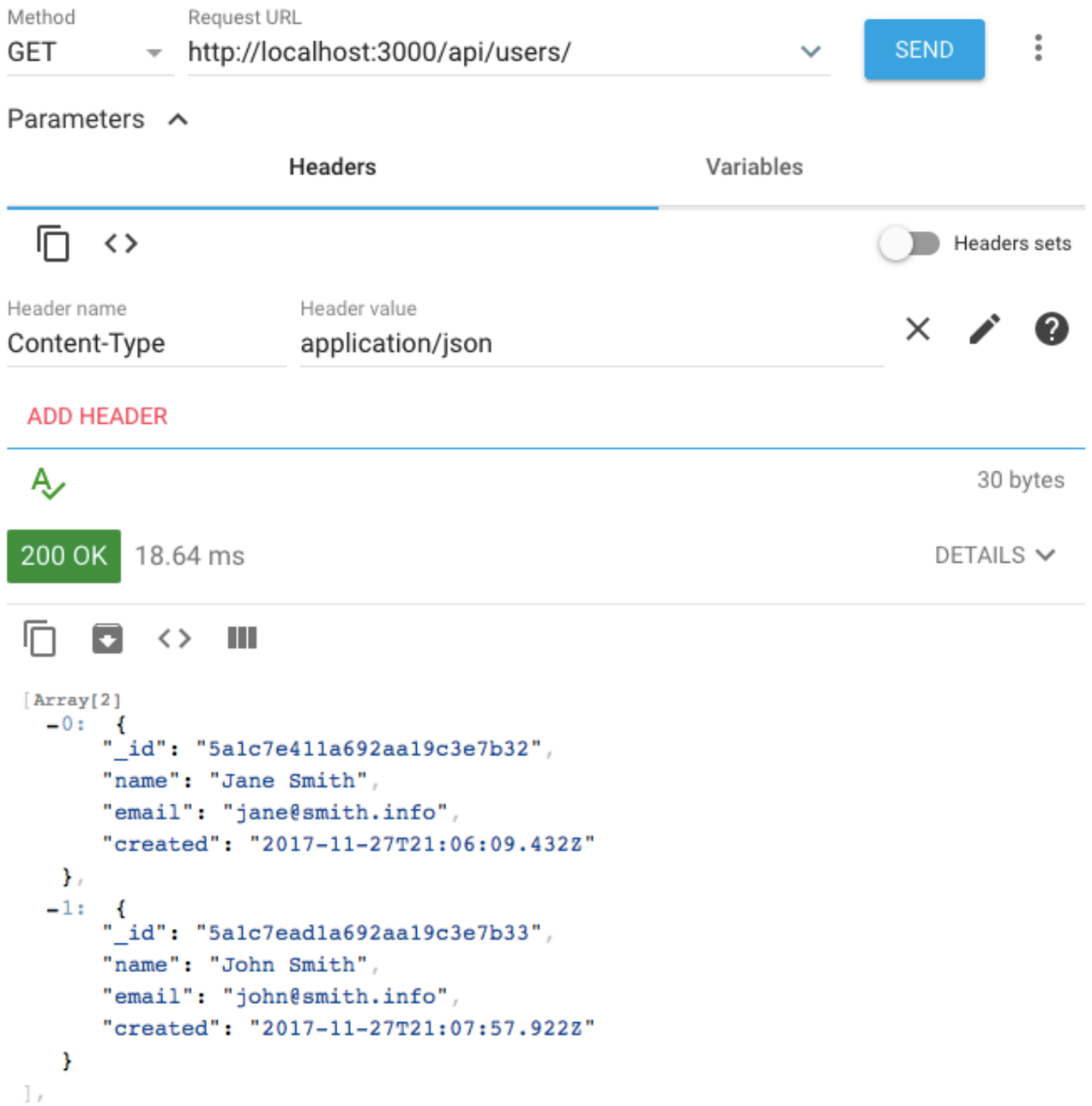
First, we will create a new user with the `/api/users` POST request and pass name, email, and password values in the request body. When the user is successfully created in the database without any validation errors, we will see a 200 OK success message, as shown in the following screenshot:



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

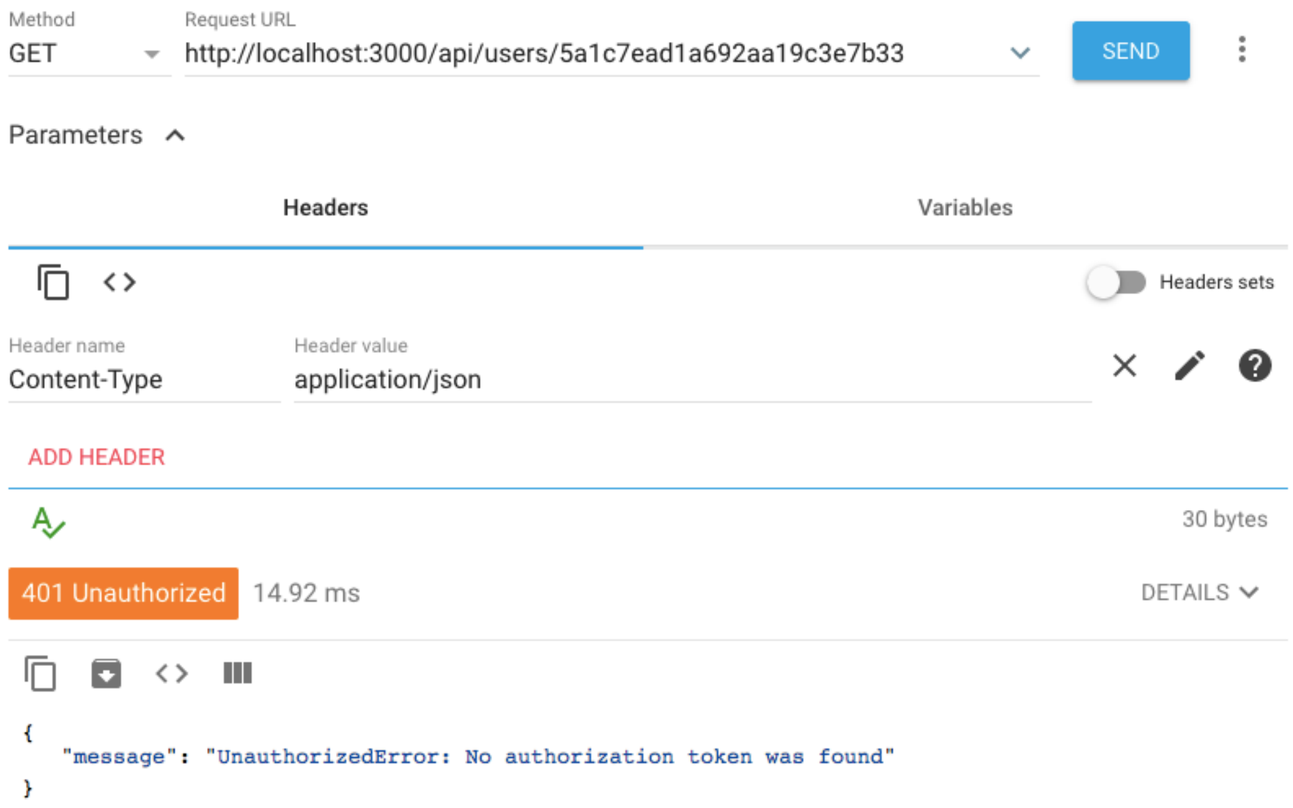
We can see whether a new user is in the database by fetching a list of all users with a `GET` request to `/api/users`. The response should contain an array of all the user objects stored in the database:



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

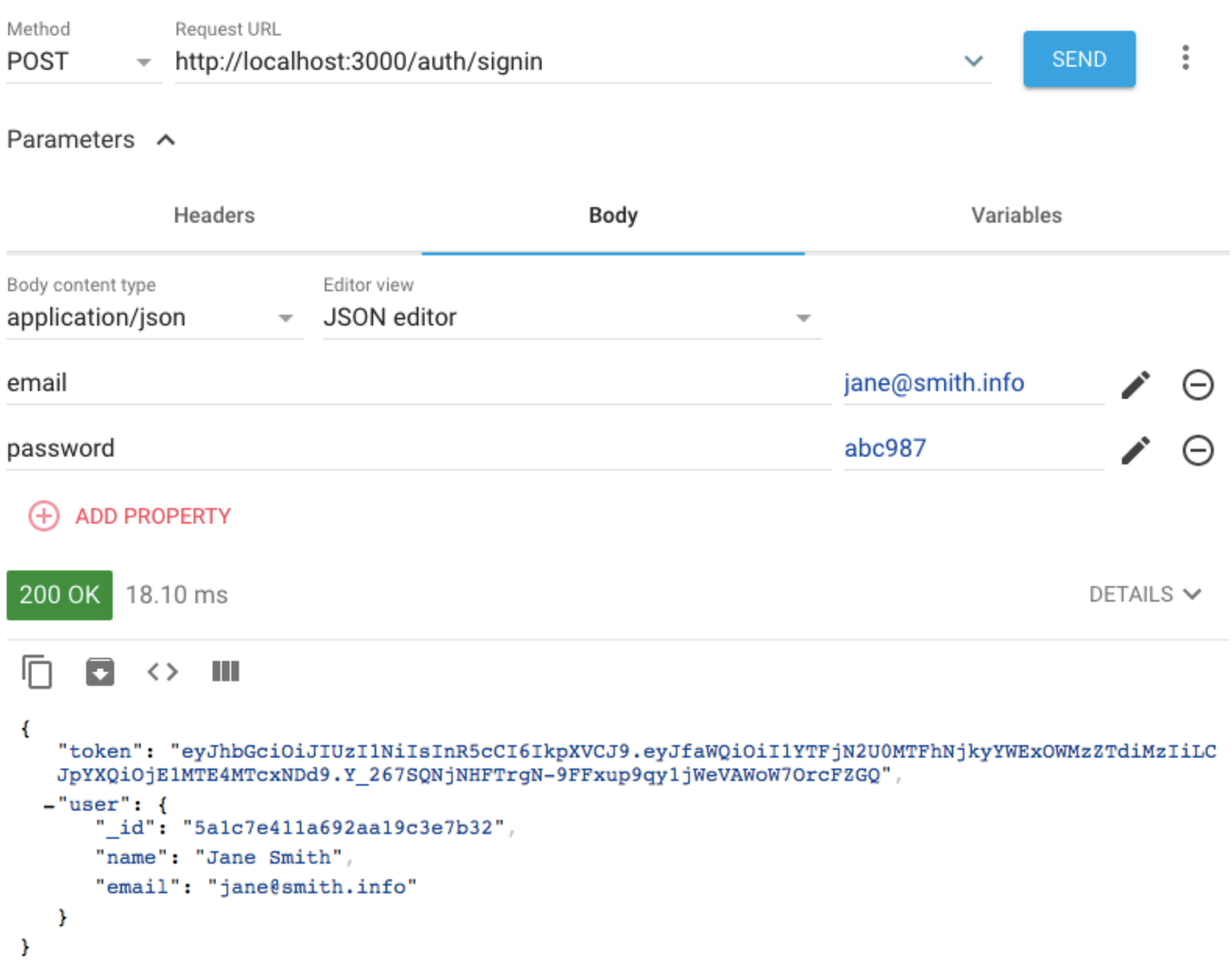
Next, we will try to access a protected API without signing in first. A `GET` request to read any one of the users will return a 401 Unauthorized error, such as in the following example. Here, a `GET` request to `/api/users/5a1c7ead-1a692aa19c3e7b33` returns a 401 error:



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

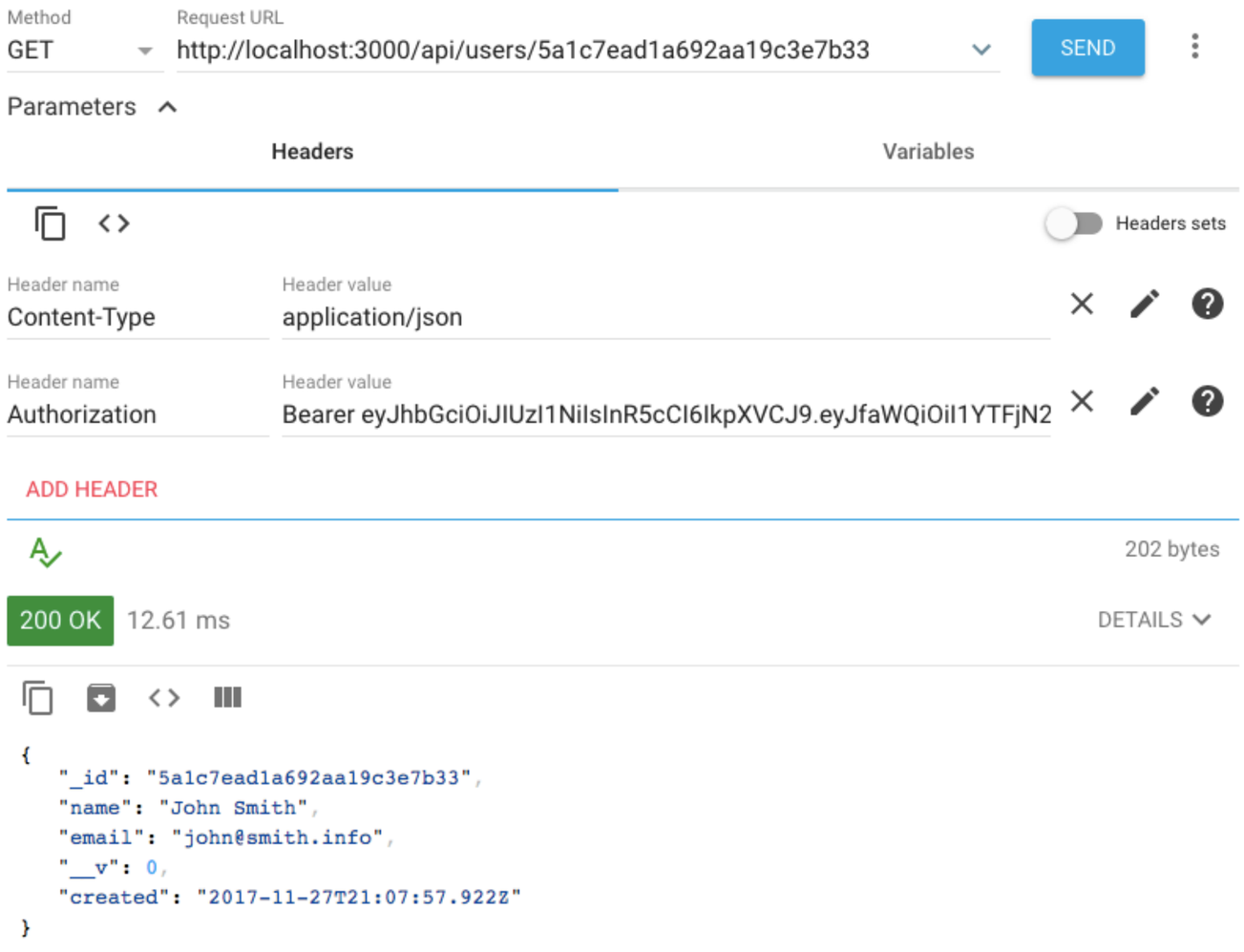
To be able to access the protected route, we will sign-in using the credentials of the user we created in the first example. To sign-in, a POST request is sent to `/auth/signin` with the email and password in the request body, as shown in the following screenshot:



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/5a1c7ead-1a692aa19c3e7b33`. This time, the user object is returned successfully:



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.