Build a Web Application with Node, Express and MongoDB - From Scratch to Production

A Comprehensive Introduction

Kevin De Notariis

Contents

List of Tables

List of Figures

1 Introduction

In this course we will be building a simple website called *Handle your Training Sessions* in which most of the features involving the creation of a website will be explored, both *back-end* and *front-end*, using **Node.js**, **Express.js** and **MongoDB**.

1.1 Requirements

I'm going to assume that the reader is (somewhat) familiar with:

- Javascript (ES6) ;
- HTML ;
- JQuery ;
- npm .

1.2 Tools

I'm going to use VSCode as the editor for the course, both because of its wholesome number of functionalities and its built-in terminals.

2 First Steps

2.1 Initialize Node

In order to start our project, first open the terminal integrated in VSCode and run the following command:

```
npm init
```

There will be asked some questions and we can answer them as we see fit, I will leave almost all of them blank, I'll just write `server.js` when asked about the `entry point` (the default would be `index.js`, but we will use `server.js` instead. note: this is completely arbitrary). I'll also write my name in the `author` entry.

This will create a **`package`**`.json` file in the root folder of the following form:

```
{
    "name": "test_1",
    "version": "1.0.0",
    "main": "server.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "kevin de notariis",
    "license": "ISC"
}
```

2.2 Add main dependencies

We now need to install `express` by typing in the terminal:

```
npm i express
```

This will create a `node_module` folder and a **`package`**`-lock.json`. The `node_module` folder will store all the installed modules, while the **`package`**`-lock.json` will be used by `npm` to check the correct versions and compatibilities of all the modules used.

2.3 Create and start the server

- In order to start a server, create a `server.js` file in the root directory.
- Open it and type:

  ```
  const express = require("express");
  ```

  this will simply import the express module and ”store” it into a variable called `express`.
- Define the actual `app`, below the line just written, type:

  ```
  const app = express();
  ```
- Finally, to start the server, the `app` just needs to listen to a given port, and this can be done by employing the method `.listen(PORT, callback)` of `app`, taking the PORT number as first argument and a callaback function. E.g. :ù

  ```
  app.listen(3000, () => {
      console.log("Server listening to port 3000");
  });
  ```
- We can save the file and run the server by typing in the terminal:

  ```
  node server
  ```

  and we will see the following output in the terminal:

  ```
  Server listening to port 3000
  ```

> **Note:** We can stop the server by hitting `Ctrl-C`.

At this point our root directory will have the following structure:

```
.├──
 _node_modules│└──
    ...├──
 package-lock.json├──
 package.json└──
 server.js
```

3 Adding Routes

3.1 app.use()

Now that we have a server which will listen to the port `3000`, we can start to add some routes. The syntax is pretty straightforward, in `server.js`, before `app.listen` type:

```
app.use("/", (req, res) => {
    res.send("Hello World");
});
```

Now, if we start the server by typing `node server` and we open the browser and navigate to `http://localhost:3000/`, we will see a "Hello World" response onto the we page sent by our route.

> ***Note:***
>
> We can make use of the `scripts` entry in the **`package.json`** to run our server. Open **`package.json`** and substitute the following entry:
>
> ```
> "scripts": {
>             "test": "echo \"Error: no test specified\" && exit 1"
>         },
> ```
>
> with the following:
>
> ```
> "scripts": {
>             "start": "node server"
>         },
> ```
>
> Now, in the terminal, in order to start the server we will just type:
>
> ```
> npm start
> ```

3.2 app.get()

The middleware `.use` will match every path containing '/', namely every possible route (in fact, one can navigate to, for example, `http://localhost:3000/hello` and we will still get the "Hello world" response)

If we would like to match only the wanted route, we should use the HTTP verb `get` as follows:

```
app.get("/", (req, res) => {
    res.send("Hello World");
});
```

If we now try to go to `http://localhost:3000/hello` we will get a

```
Cannot GET /hello
```

response.

We can now add an arbitrary number of routes, namely we can add the following:

```
app.get("/user", (req, res) => {
    res.send("Hello user page");
});
```

```
app.get("/user/:id", (req, res) => {
    res.send(`Hello user with id: ${req.params.id}`);
});
```

If we restart the server, now we can navigate to `http://localhost:3000/user` and we will get the response:

```
Hello user page
```

we can also navigate to whatever route we want from user, examples would be

`http://localhost:3000/user/222`

`http://localhost:3000/user/333`

and as response we will get, respectively:

```
Hello user with id: 222
Hello user with id: 333
```

What we are employing here is a dynamic route. The parameter `id` in the route, can be accessed in `req.params` which stores the parameters in the request.

> ***Note:***
>
> Every time we change the server, we have to stop it and then re-run:
>
> ```
> npm start
> ```
>
> However, we can install a package which will allow us to make changes in the server files and upon saving the file, the server will restart automatically. This module is called `nodemon`, and we can type in the terminal:
>
> ```
> npm i nodemon -D
> ```
>
> where the `-D` means that we are saving this module under the development dependencies, which will not be carried over in the build. We also modify the `scripts` element in the **`package`**`.json` as follows:
>
> ```
> "scripts": {
>                 "start": "nodemon server"
>             },
> ```

If we now make some changes in the server file, i.e. we add a route `/hello` and we save the file, we will see the following prompt from `nodemon`:

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server listening to port 3000
```

and we will readly be able to navigate to the newly created route.

At this point, the file `server.js` should look like this:

```
const express = require("express");

const app = express();
```

```
app.get("/", (req, res) => {
    res.send("Hello World");
});

app.get("/user", (req, res) => {
    res.send("Hello user page");
});

app.get("/user/:id", (req, res) => {
    res.send(`Hello user with id: ${req.params.id}`);
});

app.listen(3000, () => {
    console.log(`Server listening to port 3000`);
});
```

while the **package**.json as follows:

```
{
    "name": "test_1",
    "version": "1.0.0",
    "main": "server.js",
    "scripts": {
        "start": "nodemon server"
    },
    "author": "kevin de notariis",
    "license": "ISC",
    "dependencies": {
        "express": "^4.17.1"
    },
    "devDependencies": {
        "nodemon": "^2.0.6"
    }
}
```

4 More Structure to the Project

It is good practice to not clutter the server.js file with all the routes of the application by moving them into their own folder and then use the built-in Router class to indeed create modular and mountable route handlers.

Let's then create a routes folder in the main directory and a index.js which will be the entry point. The folder structure will then look like this:

```
.├──
 _node_modules│└──
    ...├──
 _routes│└──
    index.js├──
 package-lock.json├──
 package.json└──
 server.js
```

Let's now open the index.js and write the following:

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

module.exports = () => {
    router.get("/", (req, res) => {
        res.send("Hello World");
    });

    app.get("/user", (req, res) => {
        res.send("Hello user page");
    });

    app.get("/user/:id", (req, res) => {
        res.send(`Hello user with id: ${req.params.id}`);
    });

    return router;
};
```

We can now delete the routes in `server.js` and instead add the following:

```
...
const routes = require('./routes');
...
app.use('/', routes());
...
```

The server will run exactly as before, but we managed to decouple the routes with the actual server and we will be able to add more routes in a more structured way.

4.1 Moving user route into it's own folder

Now that we have a route folder, we can create another folder inside it called `user` and then create an `index.js` file inside it. The root folder structure should be as follows:

```
.├──
 _node_modules│└──
    ...├──
 _routes│├──
    _user│└──
      index.js│└──
    index.js├──
 package-lock.json├──
 package.json└──
 server.js
```

In `/routes/user/index.js` write:

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

module.exports = () => {
    router.get("/", (req, res) => {
        res.send("Hello user");
    });
```

```
    router.get("/:id", (req, res) => {
        res.send(`Hello user with id: ${req.params.id}`);
    });

    return router;
};
```

while in the `/routes/index.js` just remove the routes for `/user` and `user/:id` and add:

```
...
const userRoute = require("./user");

module.exports = () => {
    ...
    router.use("/user", userRoute());

    return router;
}
```

Now everything should work as before, we can navigate to `http://localhost:3000/user` as before and to any other route in `/user`.

5 Render an HTML Page

Now that we have set-up some routes, we should consider rendering some actual HTML page. Since we want a dynamic website, namely dynamic webpages, we need to employ a **view engine.**

In this regard, we are going to use `ejs`, to see the documentation check the website https://ejs.co.

But how do we tell express to employ this view engine?

First, we need to install the `ejs` module:

```
npm i ejs
```

Once completed, we can open up the `server.js` file and add the following code:

```
const path = require("path");

app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "./views"));
```

> ***Notes:***
>
> - In the first line we are requiring the node path module used in the last line;
> - The second line will tell express to consider `ejs` as the chosen view engine;
> - The third line instructs express to look for the views in the `./views` folder (in other words, when we will call `res.render()` in our routes, the root folder will be `./views/`), which will be in the root directory and that we are going to create and populate soon.

Our `server.js` will look like:

```
const express = require("express");
const routes = require("./routes");
const path = require("path");
```

```
const app = express();

app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "./views"));

app.use("/", routes());

app.listen(3000, () => {
    console.log(`Server listening to port 3000`);
});
```

Let's now create the `views` folder in root directory and a `index.ejs` in it.

The root folder structure should now look like this:

```
.├──
 _node_modules│└──
    ...├──
 _routes│├──
    _user│└──
       index.js│└──
    index.js├──
 _views│└──
    index.ejs├──
 package-lock.json├──
 package.json└──
 server.js
```

Let's open now the `views/index.ejs` file and simply write the base HTML code below:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Project Title</title>
    </head>

    <body>
        <header>
            <h1>Hello Home Page</h1>
        </header>
    </body>
</html>
```

Open up now the `routes/index.js` file and instead of the line

```
res.send("Hello World");
```

we are going to put the following code:

```
res.render("./");
```

And upon saving all the files and opening the browser at `http://localhost:3000`, we should see the rendered HTML page with "Hello Home Page".

5.1 Further Step - Pass parameters from routes to views.

If we want to render some dynamic parameters in `views/index.ejs`, we can pass them in an object as a second argument to the `res.render()` call in `routes/index.js` file. Let's open it and modify the line:

```
res.render("./");
```

to:

```
res.render("./", {
    pageTitle: "Home Page",
    header: "Home Page Header",
});
```

These parameters can be accessed in `views/index.ejs` by employing the `ejs` syntax, namely as follows

```
<%= pageTitle %>
<%= header %>
```

In particular, we can replace the hard-coded `title` and `h1` in `views/index.ejs` with the above lines, obtaining:

```
<!DOCTYPE html>
<html>
    <head>
        <title><%= pageTitle %></title>
    </head>

    <body>
        <header>
            <h1><%= header %></h1>
        </header>
    </body>
</html>
```

Unpon saving and refreshing the browser we should be able to see the changes.

6 Creating a Layout for our Webpages

In order to not repeat everytime the same HTML code, we can create a common layout and then define different "components" (in a subdirectory called `pages`) which will be "mounted" when needed.

In `views` let's create a `layout` folder and move in there the `index.ejs` file. The structure should look like this:

```
.├──
 _node_modules│└──
    ...├──
 _routes│├──
    _user│└──
       index.js│└
    index.js├──
```

```
 _views│└──
    _layout│└──
        index.ejs├──
 package-lock.json├──
 package.json└──
 server.js
```

Also, change in the `routes/index.js` file the `.render()` method by taking into account the change of the `index.ejs`, but also the fact that we will dynamically pass to the layout index page the actual page that we would like to render, namely (we will also remove the ”header” key):

```
res.render("layout", {
    pageTitle: "Home Page",
    template: "index",
});
```

Now, open up `views/layout/index.ejs` and modify it as follows:

```
<!DOCTYPE html>
<html>
    <head>
        <title><%= pageTitle %></title>
    </head>

    <body>
        <header>
            <h1>Welcome to the <%= pageTitle %></h1>
        </header>

        <%- include(`../pages/${template}`) %>
    </body>
</html>
```

> With the `<%- include('../pages/${template}')%>` we are telling `ejs` to take everything in the file `../pages/${template}` and put it in there unescaped.

Create now the `pages` in `views` with a `index.ejs` file in there.

The folder structure should look like:

```
.├──
 _node_modules│└──
    ...├──
 _routes│├──
    _user│└──
       index.js│└──
    index.js├──
 _views│├──
    _layout│└──
       index.ejs│└──
    _pages│└──
        index.ejs├──
 package-lock.json├──
 package.json└──
 server.js
```

In `views/pages/index.ejs` let's write the following:

```
<div>
    <h2>h2 in there!</h2>
</div>
```

After saving the files we should see the new `h2` in the Home Page.

6.1 Home Page Creation

Now that we have a base structure for the project, let's add some HTML code to render a nice looking front home page. We are going to use `Bootstrap`, so let's install it.

6.1.1 Bootstrap in Express

```
npm i bootstrap
```

Bootstrap also uses `jquery` so we need to install it too:

```
npm i jquery
```

This will furnish us with lots of cool css and components to ease the front-end building process.

In for us to use the bootstrap `CSS` and components, we need to tell express where to find the static files. In this regard, let's create a **`public`** folder, and inside it a `styles` and a `js` folder. Inside **`public`**`/styles` create a `css` folder. The directory structure should look like:

```
.├──
 _node_modules│└──
    ...├──
 _public│├──
    _styles│└──
       css│└
    js├──
 _routes│├──
    _user│└──
       index.js│└
    index.js├──
 _views│├──
    _layout│└──
       index.ejs│└──
    _pages│└──
        index.ejs├──
 package-lock.json├──
 package.json└──
 server.js
```

In `server.js` let's add:

```
app.use(express.static(path.join(__dirname, "public")));
```

Since the `CSS` we will be using from bootstrap is in `node_modules/bootstrap/dist/css` and the `javascript` is in `node_modules/bootstrap/dist/js`, we need to tell express to consider these

as if it were in the newly created `public` folder. We also need to tell express where to find `jquery`, so in `server.js` write:

```
app.use(
    "/styles/css",
    express.static(path.join(__dirname, "node_modules/bootstrap/dist/css")
        )
);

app.use(
    "/js",
    express.static(path.join(__dirname, "node_modules/bootstrap/dist/js"))
);

app.use(
    "/js",
    express.static(path.join(__dirname, "node_modules/jquery/dist"))
);
```

6.1.2 Serve Bootstrap's CSS and JS to HTML

Let's create `components` folder inside `layout` in which we will be storing the components commonly used by evey page, then create a `scripts.ejs` file inside it. The folder structure should now look like:

```
.├──
 _node_modules│└──
    ...├──
 _public│├──
    _styles│└──
       css│└
    js├──
 _routes│├──
    _user│└──
       index.js│└
    index.js├──
 _views│├──
    _layout│├──
       _components│└──
          scripts.ejs│└──
       index.js│└──
    _pages│└──
        index.ejs├──
 package-lock.json├──
 package.json└──
 server.js
```

In `scripts.ejs` add the following script tags:

```
<script language="javascript" src="/js/jquery.slim.min.js"></script>
<script language="javascript" src="/js/bootstrap.bundle.min.js"></script>
```

and then in `views/layout/index.ejs` we should serve this `scripts` file and add the `CSS` link. This `index.ejs` should then look like this:

```
<!DOCTYPE html>
<html>
    <head>
        <!-- Meta tags -->
        <meta charset="utf-8" />
        <meta
            name="viewport"
            content="width=device-width, initial-scale=1, shrink-to-fit=no
                "
        />

        <!-- Bootstrap CSS -->
        <link rel="stylesheet" href="/styles/css/bootstrap.min.css" />

        <title><%= pageTitle %></title>
    </head>

    <body>
        <header>
            <h1>Welcome to the <%= pageTitle %></h1>
        </header>

        <%- include(`../pages/${template}`) %>

        <%-include('./components/scripts') %>
    </body>
</html>
```

> ***Note:***
>
> We have added also some meta tags which are recommended by `bootstrap`. For more information visit https://getbootstrap.com/.

Everything should now be set correctly, and we should be able to proceed with the actual implementation of some HTML code using `bootstrap`.

6.1.3 Footer

Let's add a `footer.ejs` component in `views/layout/components`. Open it up and add the following code:

```
<footer class="container fixed-bottom">
    <div class="text-center py-3">
        <p class="footer-text">
            <strong>© 2020 Copyright: Kevin De Notariis</strong>
        </p>
    </div>
</footer>
```

where the class `.footer-text` will be defined in a `.css` file in a moment and the other classes are from bootstrap's `CSS`.

Create `style.css` in **`public`**`/styles/css` and put there the following code:

```
.footer-text {
    color: white;
}
```

At this point, the project structure should look like the following:

```
.
├── _node_modules
│   ├── ...
├── _public
│   ├── _styles
│   │   └── _css
│   │       └── style.css
│   └── js
├── _routes
│   ├── _user
│   │   └── index.js
│   └── index.js
├── _views
│   ├── _layout
│   │   ├── _components
│   │   │   ├── footer.ejs
│   │   │   └── scripts.ejs
│   │   └── index.js
│   └── _pages
│       └── index.ejs
├── package-lock.json
├── package.json
└── server.js
```

6.1.4 Home Page Body

Let's now personalize the body of the front page.

> ***Note:***
>
> This course is not about neither HTML nor CSS, for that reason I'm not going to deeply explain how does the pure HTML and CSS code that I'll put in work.

This is the structure that we will create:

- A background image covering all the screen ;
- a jumbotron header with a welcoming message ;
- a button in center of the screen allowing user to login (or eventually sign in).

As the background image you might use a cool image taken from https://unsplash.com/s/photos/fitness. Download it and create a folder named `img` inside the **`public`** folder and put the image in there. I will call this image `front-image.jpg`.

Open up the `views/pages/index.ejs` and substitute it's content with the following:

```
<div class="homePage">
    <!-- Background Image -->
    <img src="/img/front-image.jpg" class="bg" />

    <!-- Jumbotron header with welcoming message -->
    <div class="jumbotron">
```

```
        <div class="col-md-6 px-0">
            <h1 class="display-4 font-italic">
                Welcome to <strong> <%= siteName %></strong>
            </h1>
            <p class="lead">
                A Website built for athletes and people which are
                    regularly
                exercising/going to the gym and would like to keep track
                    of
                their progresses
            </p>
        </div>
    </div>

    <!-- Login Button -->
    <div class="d-flex justify-content-center up-front">
        <a href="/login" class="brk-btn" href="#">Login </a>
    </div>

    <!-- Text below Login Button with link to register page -->
    <div class="d-flex justify-content-center up-front">
        <p style="color: white">
            Login in order to access your profile or
            <a class="underlined-a" href="/register">register here</a>
        </p>
    </div>
</div>
```

I've also added some comments explaining the different parts coded. The CSS classes that we have used here can be added in the **public**/styles/css/style.css and are the following (there is no need to understand how this work, I just post it there for completeness):

```
img.bg {
    min-height: 100%;
    min-width: 1024px;

    width: 100%;
    height: auto;

    position: fixed;
    top: 0;
    left: 0;
}

@media screen and (max-width: 1024px) {
    img.bg {
        left: 50%;
        margin-left: -512px;
    }
}

.up-front {
    position: relative;
    z-index: 2;
}

.underlined-a {
    text-decoration: none;
    color: white;
    padding-bottom: 0.15em;
```

```
    box-sizing: border-box;
    box-shadow: inset 0 -0.2em 0 white;
    transition: 0.2s;
}
.underlined-a:hover {
    color: #222;
    box-shadow: inset 0 -2em 0 white;
    transition: all 0.45s cubic-bezier(0.86, 0, 0.07, 1);
}

.brk-btn {
    position: relative;
    background: none;
    color: rgba(255, 255, 255, 0.356);
    text-transform: uppercase;
    text-decoration: none;
    border: 0.2em solid rgba(255, 255, 255, 0.356);
    padding: 0.8em 2em;
    font-size: 20px;
    transition: 0.3s;
}
.brk-btn:hover {
    color: white;
    border: 0.2em solid white;
    padding: 1em 2.4em;
    text-decoration: underline;
    font-size: 22px;
}

.brk-btn::before {
    content: "";
    display: block;
    position: absolute;
    width: 10%;
    background: #222;
    height: 0.3em;
    right: 20%;
    top: -0.21em;
    transform: skewX(-45deg);
    -webkit-transition: all 0.45s cubic-bezier(0.86, 0, 0.07, 1);
    transition: all 0.45s cubic-bezier(0.86, 0, 0.07, 1);
}
.brk-btn::after {
    content: "";
    display: block;
    position: absolute;
    width: 10%;
    background: #222;
    height: 0.3em;
    left: 20%;
    bottom: -0.25em;
    transform: skewX(45deg);
    -webkit-transition: all 0.45 cubic-bezier(0.86, 0, 0.07, 1);
    transition: all 0.45s cubic-bezier(0.86, 0, 0.07, 1);
}
.brk-btn:hover::before {
    right: 80%;
}
.brk-btn:hover::after {
    left: 80%;
}
```

Now, in `views/pages/index.ejs` we can see that we have added a line of the form:

```
Welcome to <strong> <%= siteName %></strong>
```

and we have to define the variable `siteName`. Since this will be a global variable shared by every page, we can define it in the `locals` property of our server. In `server.js` just add the following line before the `app.use("/", routes());` and we will be good to procede further:

```
app.locals.siteName = "* Web Site Name *";
```

> We can also change in `views/layout/index.ejs` the following line:
>
> ```
> <title><%= pageTitle %></title>
> ```
>
> with:
>
> ```
> <title><%= siteName %> | <%= pageTitle %></title>
> ```

7 Register And Login

Since we have created a login button redirecting to `/login` and a register link redirecting to `/register`, we need to implement these routes and create the suitable pages, controllers and models for the Users. We will be using the **MVC design pattern** (Model - View - Controller) and we start here by incorporating MongoDB in our project.

7.1 Setting up MongoDB and mongoose

First, we need to install the `mongoose` module, which will bring with it the `mongodb` module itself, so let's type in the terminal:

```
npm i mongoose
```

Then open up the `server.js` file and import `mongoose`:

```
const mongoose = require("mongoose");
```

If you do not have mongoDB installed, you can go here https://www.mongodb.com and download it in you local machine and install it as a service, in this way it will be immediately ready to be used.

In `server.js` let's connect to mongoDB utilizing `mongoose`, namely write:

```
mongoose.connect("mongodb://localhost/trainingDB", {
    useNewUrlParser: true,
    useUnifiedTopology: true,
});
```

> - `trainingDB` will be the name of our database;
> - `useNewUrlParser` and `useUnifiedTopology` are two parameters that we need to set, otherwise mongoDB will complain about deprecation issues.

MongoDB does not have a predefined structure, the collections in a database (which can be compared to tables in a SQL-type database) are filled with documents which can have completely different structure. In order to have a predefined structure, mongoose allows us to define so-called *Schemas*.

7.1.1 Create a Mongoose Schema

Let's create a `models` folder in the root directory and then create a `models/userModel.js` file. The structure of the project should look like:

```
.├──
 _models│└──
    userModel.js├──
 _node_modules│└──
    ...├──
 _public│├──
    _styles│└──
       _css│└──
          style.css│└
    js├──
 _routes│├──
    _user│└──
       index.js│└
    index.js├──
 _views│├──
    _layout│├──
       _components│├──
          footer.ejs│└──
          scripts.ejs│└──
       index.js│└──
    _pages│└──
        index.ejs├──
 package-lock.json├──
 package.json└──
 server.js
```

In this newly created file, let's import `mongoose` and `bcrypt`, the latter will be used to hash the password (install it via `npm i bcrypt`):

```
const mongoose = require("mongoose");
const bcrypt = require("bcrypt");
```

Now, we define the `Schema`:

```
const Schema = mongoose.Schema;

module.exports = UserSchema = new Schema({
    email: {
        type: String,
        required: true,
    },
    hashPassword: {
        type: String,
        required: true,
    },
```

```
    create_date: {
        type: Date,
        default: Date.now(),
    },
});
```

> We have defined an `email` field of type `String` which is required, a `hashPassword` field also of type `String` and required and a `create_date` which will store the date in which the user was created. Note that we are storing the password not as the user types it, but we store the *hashed password*, so that if someone gains access of our database, they couldn't use the hashed password to login.

Finally, we need to add a method to `UserSchema` which compares the actual password (which will be used by the user to login) with the hashed password stored in the database:

```
UserSchema.methods.comparePassword = (password, hashPassword) => {
    return bcrypt.compareSync(password, hashPassword);
};
```

Now that we have a schema, we can proceed to create a **userController** which will be used to handle the actions of the users.

7.2 User Controller

Create a folder `controllers` in the root directory and a `userController.js` inside it. The folder structure shoul look like this:

```
.├──
 _controllers│└──
    userController.js├──
 _models│└──
    userModel.js├──
 _node_modules│└──
    ...├──
 _public│├──
    _styles│└──
       _css│└──
          style.css│└
    js├──
 _routes│├──
    _user│└──
       index.js│└
    index.js├──
 _views│├──
    _layout│├──
       _components│├──
          footer.ejs│└──
          scripts.ejs│└──
       index.js│└──
    _pages│└──
        index.ejs├──
 package-lock.json├──
```

```
package.json
server.js
```

In `controllers/userController.js` we will define the middlewares that are going to be used by a user in its interactions with the website. To be sure that the user is indeed logged in and authorized to make the requests, we will use **JWTs** (Json Web Tokens) and in node we have a module called `jsonwebtoken` which we install by typing:

```
npm i jsonwebtoken
```

Let's now open up `controllers/userController.js` and first import the needed modules:

```
const mongoose = require("mongoose");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcrypt");

const { UserSchema } = require("../models/userModel");
```

Let's define now the model that we are using, note that (from https://mongoosejs.com/docs/models.html):

> "Mongoose automatically looks for the plural, lowercased version of your model name"

Namely, if we define a model called `User` (first argument of `mongoose.model()`), then mongoose will search for the collection named `users` in the database. Also keep in mind that an instance of a model is a document which will then be saved in the corresponding collection. Using this insight we write:

```
const User = mongoose.model("User", UserSchema);
```

Now we can start adding the middlewares `register`, `login` and a `loginRequired`. The latter will be used before every other middleware to ensure that the user is logged in before doing anything.

7.2.1 loginRequired

in our `userController.js` file let's add:

```
const loginRequired = (req, res, next) => {
    if (req.user) {
        next();
    } else {
        return res.status(401).json({ message: "Not Authorized" });
    }
};
```

> If there is a user we pass to the next middleware, while if the user is not logged in, we return an "unauthorized" error status.

7.2.2 register

Following the `loginRequired` function we add the `register`:

```
const register = (req, res, next) => {
    const newUser = new User(req.body);
    newUser.hashPassword = bcrypt.hashSync(req.body.password, 10);
    newUser.save((err, user) => {
        if (err) {
            return res.status(400).json({ message: err });
        } else {
            user.hashPassword = undefined;
            return res.json(user);
        }
    });
};
```

> ***Note:***
>
> - First we create an instance of the model, which is nothing but a document (for mongoDB) ;
> - Then we hash the password returned form the user input ;
> - We save the document ;
> - In the callback we check whether there is any error;
> - If no errors occurred, we remove the password from the user document, since we do not want to send back the password.
> - Finally we return the json with the data.

7.2.3 login

Finally we implement the `login` as follow:

```
const login = (req, res, next) => {
    User.findOne(
        {
            email: req.body.email,
        },
        (err, user) => {
            if (err) throw err;
            if (!user) {
                return res
                    .status(401)
                    .json({ message: "Authentication failed" });
            } else {
                if (
                    !user.comparePassword(req.body.password, user.
                       hashPassword)
                ) {
                    return res
                        .status(401)
                        .json({ message: "Authentication failed" });
                } else {
                    user.hashPassword = undefined;
                    return res.json({
                        token: jwt.sign(
                            {
                                email: user.email,
                                _id: user.id,
                            },
                            "QuantumElectroDynamcics4Real"
                        ),
```

```
                    });
                }
            }
        }
    );
};
```

So let's break up this middleware:

> - First we query the database for the existence of a document with `email` field equals to the email typed in by the user ;
> - In the callback we receive an error (if occurs) and the document we asked for (if exists). So here we immediatly check if an error occurred, and if so we throw an error ;
> - If no errors occurred, we check whether there is a user in the database with the given email and if not we return a `401` status with a message telling that the authentication failed ;
> - If a user with the given email exists, then we check whether the password inserted coincide upon hashing with the hashed password stored with the given email. If the passwords do not match, we return a status `401` again with the same message as before ;
> - If the passwords match, we first remove the hashed password from user (since we do not pass to the front-end passwords) and then we return a JWT with the signed in email and user id, with the encryption word "QuantumElectroDynamics4Real".

As for now, we are just returning a response with the JWT token just for testing purposes. Afterwards we will store it in an appropriate cookie session for security purposes.

Finally, let's export all these functions:

```
module.exports = {
    loginRequired,
    register,
    login,
};
```

7.2.4 Set up JWT

In `loginRequired` we have checked for a property of the `request` objecy, namely we checked for the existence of a `req.user`. We need then to define this property. In `server.js` first import `jsonwebtoken`:

```
const jwt = require("jsonwebtoken");
```

and then, before `app.user("/", routes())` , let's implement JWT:

```
app.use((req, res, next) => {
    if (
        req.headers &&
        req.headers.authorization &&
        req.headers.authorization.split(" ")[0] === "JWT"
    ) {
        jwt.verify(
            req.headers.authorization.split(" ")[1],
            "QuantumElectroDynamics4Real",
            (err, decode) => {
```

```
                if (err) res.user = undefined;
                req.user = decode;
                next();
            }
        );
    } else {
        req.user = undefined;
        next();
    }
});
```

Let's break this up:

- We check whether the incoming message has an header and if this header has an authorization field with first element indeed equal to 'JWT' ;
- If this is the case, we check for the other part of the header authorization, namely the token itself with the secret word defined before.
- The result of `.verify` will be a decoded token if the secret word is valid and we will store it in `req.user`. If there is an error, `decode` will be undefined and we respond with a `res.user` undefined.
- Finally if there is no header / authorization / JWT part, then it means that the user is not authenticated.

7.3 Register Route and Page

Let's first add the register route. Create a `register` folder inside `routes` and a `index.js` inside it. The folder structure should now look like this:

```
.├──
 _controllers│└──
    userController.js├──
 _models│└──
    userModel.js├──
 _node_modules│└──
    ...├──
 _public│├──
    _styles│└──
       _css│└──
          style.css│└
    js├──
 _routes│├──
    _register│└──
       index.js│├──
    _user│└──
       index.js│└
    index.js├──
 _views│├──
    _layout│├──
       _components│├──
          footer.ejs│└──
          scripts.ejs│└──
       index.js│└──
```

```
    _pages
        index.ejs
 package-lock.json
 package.json
 server.js
```

Inside the newly created `routes/register/index.js` add the following code:

```
const express = require("express");

const { register } = require("../../controllers/userController");

const router = express.Router();

module.exports = () => {
    router.get("/", (req, res) => {
        res.render("layout", {
            pageTitle: "Register",
            template: "register",
        });
    });

    router.post("/", register);
};
```

- We first import all the necesary modules, including our `register` created before ;
- We then define a `GET` middleware as we have done in `routes/index`, but we pass a different `template` and `pageTitle` (we are going to create the register view in a moment) ;
- Define a `POST` middleware and passing the `register` we created.

Now, in `views/pages` add a `register.ejs` file. Open it up and put in there the following code:

```
<form type="POST" action="/register">
    <label>email</label>
    <input type="text" name="email" placeholder="email" />
    <label>password</label>
    <input type="password" name="password" placeholder="email" />
    <input type="submit" value="Submit" />
</form>
```

This is just a simple form in order to test our code, in the future we are going to style it more.

Now, when in the homepage `http://localhost:3000` we click on `register here` we will be redirected to `/register` with the form just written.

7.4 Body-Parser

In order to interpret what a form is returning, express needs a middleware called `body-parser` and we can install it:

```
npm i body-parser
```

Then we require it in `server.js` :

```
const bodyParser = require("body-parser");
```

and add the following lines of code (just before the JWT middleware created before), one to parse `x-www-form-urlencoded` and one to parse JSON:

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

7.5 Login Route and Page

Analogously to the register route and page, we create a `login` folder inside `routes` and a `index.js` file inside this folder. Also, create a `login.ejs` inside `views/pages`. At this point the folder structure should look like:

```
.
├── _controllers
│   └── userController.js
├── _models
│   └── userModel.js
├── _node_modules
│   └── ...
├── _public
│   ├── _styles
│   │   └── _css
│   │       └── style.css
│   └── js
├── _routes
│   ├── login
│   │   └── index.js
│   ├── _register
│   │   └── index.js
│   ├── _user
│   │   └── index.js
│   └── index.js
├── _views
│   ├── _layout
│   │   ├── _components
│   │   │   ├── footer.ejs
│   │   │   └── scripts.ejs
│   │   └── index.js
│   └── _pages
│       ├── index.ejs
│       ├── login.ejs
│       └── register.ejs
├── package-lock.json
├── package.json
└── server.js
```

In the same way as before, open up `routes/login/index.js` and put there the following code:

```
const express = require("express");

const { login } = require("../../controllers/userController");

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

```
module.exports = () => {
    router.get("/", (req, res) => {
        res.render("layout", {
            pageTitle: "Login",
            template: "login",
        });
    });

    router.post("/", login);

    return router;
};
```

while in `views/pages/login.ejs`:

```
<form action="/login" method="POST">
    <label>email</label>
    <input type="text" name="email" placeholder="email" />
    <label>password</label>
    <input type="password" name="password" placeholder="password" />
    <input type="submit" value="Submit" />
</form>
```

7.6 Serving the /login and /register routes

If we now try to start the server and click on the login button or the register link, we will see that the server is not able to get these routes, why is that?

Every route we have defined is inside `/routes` and it passes through `routes/index.js`, meaning that we need to use there the newly defined routes.

Open up `routes/index.js` and require the following:

```
const registerRoute = require("./register");
const loginRoute = require("./login");
```

Now, before the `return router`, just add:

```
router.use("/register", registerRoute());
router.use("/login", loginRoute());
```

If we now try to nagivate to `/login` and `/register` we should see the forms created before.

7.7 Validate and Sanitize User Inputs

At `http://localhost/register` one can register to the website, the email, password (hashed) and the creation date will be stored in the database, in particular in `trainingDB` database and in the collection `users`. However, one can insert everything they want in the email, even a non-email and there is still no way for our website to detect this fact, it will store it in the database regardless of its form.

Also, in order to protect from injections, we need to sanitize the input, namely remove eventual html tags which may compromise our website.

First, install the node module `express-validator`:

```
npm i express-validator
```

then in `controllers/userController.js` import the needed middlewares / functions:

```
const { check, validationResult } = require('express-validator);
```

and then define:

```
const validateAndSanitize = [
    check("email").trim().isEmail().normalizeEmail().escape(),
    check("password").trim().isLength({ min: 8 }).escape(),
];
```

and in both `register` and `login` middlewares, at the beginning, add:

```
const errors = validationResult(req);
```

Then we need to check whether this `errors` is empty, so that immediately after the above line, add:

```
if (!errors.isEmpty()) {
    res.json({ message: errors.toArray() });
} else {
    .........
}
```

and in the **`else`** statement just move all the code that we have written before. The final `register` middleware should look like this:

```
const register = (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        res.json({ message: errors.array() });
    } else {
        const newUser = new User(req.body);
        newUser.hashPassword = bcrypt.hashSync(req.body.password, 10);
        newUser.save((err, user) => {
            if (err) {
                return res.status(400).json({ message: err });
            } else {
                user.hashPassword = undefined;
                return res.json(user);
            }
        });
    }
};
```

Now, do the same for `login` and also remember to export it `validateAndSanitize`:

```
module.exports = {
    validateAndSanitize,
    loginRequired,
    register,
    login,
};
```

Now, in `routes/login/index.js` and `routes/register/index.js` we need to import from `controllers/userController.js` also this newly created `validateAndSanitize`. I'll take as

an example `routes/register/index.js` but the same thing is be replicated analogously also for `routes/login/index.js`:

```
const {
    validateAndSanitize,
    register,
} = require("../../controllers/userController");
```

and in the `router.post` add it as follows:

```
router.post("/", validateAndSanitize, register);
```

Do the same for `routes/login/index.js` and now let's see how does the site respond to different inputs. Navigate to `http://localhost/register` and:

1. Write:

email	password
kevin	helloworld

   and after pressing the submit, we should see a message telling us that the email we have inserted in not valid:

```
{"message":[{"value":"kevin","msg":"Invalid value","param":"email","
   location":"body"}]}
```

2. Write:

email	password
kevin@example.com[1]	hello

   after pressing the submit, we should see a message telling us that the password is not valid (it has a length < 8 characters):

```
{"message":[{"value":"hello","msg":"Invalid value","param":"password"
   ,"location":"body"}]}
```

3. Write:

email	password
kevin	hello

   here, we should see both error messages:

```
{"message":[{"value":"kevin","msg":"Invalid value","param":"email","
   location":"body"},{"value":"hello","msg":"Invalid value","param":"
   password","location":"body"}]}
```

[1] mailto:kevin@example.com

4. Write:

email	password
kevin@example.com[2]	helloworld

now, finally, we should see a message telling us that the user has been correctly created, so something of the following form:

```
{"created_date":"2020-10-30T17:08:05.895Z","_id":"5
   f9c4a5726b0dd2018df41d7","email":"kevin@example.com","__v":0}
```

If you now open up the shell, type `mongo` and then switch to the trainingDB, namely type

```
use trainingDB
```

now by querying the database to find all documents in the collection `users`:

```
db.users.find()
```

we should see the new element we have created.

If we now go to the login page `http://localhost:3000/login` and we type the email and password that we have used in the register page (the correct ones) then we should see a response with the token, namely something like this:

```
{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
   eyJlbWFpbCI6ImtldmluQGV4YW1wbGUuY29tIiwiX2lkIjoiNWY5YzRhNTcyNmIwZGQyMDE4ZGY0MWQ3Iiw
   .UOKCKo-bc8y4Il7pjzUUCQF1WS8y6obu1vXMvKF5jH4"}
```

[2] mailto:kevin@example.com