Full Stack Project

1. Tools

1.1. Prerequisites

Install node: https://youtu.be/VShtPwEkDD0

Install MongoDB: https://youtu.be/qj2oDkvc4dQ

Install Git: https://youtu.be/IHaTbJPdB-s

1.2. Videos

1: https://youtu.be/qj2oDkvc4dQ

1.3. Developing environment

This file and the videos assume you use **Visual Studio Code**, but most instructions should be valid for any developing environment.

You'll need to install **node.js** before starting this project.

This project will be initialized in a **root folder**, that you can call whatever you want. This root folder is the highest scope of the project, no path should try and access anything outside it on your machine. We'll send content of this folder will be sent to GitHub as part of the project.

By using Express.js, the design pattern we'll use is Model-View-Controller (MVC).

1.4. Node modules list

```
express
ejs
express-ejs-layouts
dotenv
body-parser
```

2. Setting up

2.1. Node commands

2.1.1. Initialize node

```
npm init -y
```

Change "index.js" to "server.js" in the package.json file, for conveniency.

2.1.2. Install dependencies

```
npm i express ejs express-ejs-layouts
```

("i" is the same as "install")

Express is used for the server part, ejs for templating web pages, and express-ejs-layouts to bind those two together easily.

```
npm i --save-dev nodemon
```

("--save-dev" will only install as a dev tool)

The `nodemon` module is so useful you might want to install it globally (for all your projects) by using the following command instead:

```
npm i -g nodemon
```

("-g" stands for "–global")

2.1.3. Create scripts

In the "package.json" file, change the scripts as follows:

```
"scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
}
```

Those two scripts can later be called by typing `npm run start` or `npm run dev` to run the corresponding commands. They will basically do the same thing, except `nodemon` will re-fresh the server every time we save a file.

2.2. Server initialization

2.2.1. Create server file

Create a "server.js" file in the project folder.

Inside it, import `express` and `express-ejs-layouts`. Create an `app` constant for the express app for conveniency:

```
const express = require("express");
const app = express();
const expressLayouts = require("express-ejs-layouts");
```

We'll use the `app.set(setting: string, val: any)` method to configure our server in various ways.

We'll use the `app.use(middleware)` method either:

- with a single argument so the method is called every time a request is sent to the server.
- with two arguments, the first one being a URL to narrow down the use of the second argument.

2.2.2. Set view engine

Set ejs as our view engine:

```
app.set("view engine", "ejs");
```

Create a "views" folder and set its path as the being the one for views:

```
app.set("views", __dirname + "/views");
```

These files will be rendered by the server into proper html files that a browser can interpret.

Create a "layout" folder with a "layout.ejs" file inside it, and set the path of the file as the be-ing the main template for our other ejs files:

```
app.set("layout", "layouts/layout");
```

By setting this, every other ejs file will inherit from the layout.ejs file; essentially, it's going to be put inside it. This is helpful to avoid copy-pasting <header>, the main <nav> and <footer> everywhere, for example.

Instruct the express application that we'll use express-ejs-layouts:

```
app.use(expressLayouts);
```

Finally, create a "public" folder in the root folder, then tell express what its name is:

```
app.use(express.static("public"));
```

This folder is where we'll put our css files, js files and images for the website. These files will be sent directly to the user, as opposed to server-side rendered files that we put in the "views" folder.

2.2.3. Set the listening port

```
app.listen(process.env.PORT || 3000);
```

In production mode, the server will know the port it is using, but for development purposes, we need to use another one (you can use any port number you want).

2.2.4. Testing

At this point, the server should be able to run.

Boot the server with:

```
npm run dev
```

Go to "localhost:3000" in a web browser. You should get a "Cannot GET /" message.

This is normal, because no routes have been configured so far. The routes in Express.js are our controllers.

2.3. Creating routes

2.3.1. Create route files

Create a "routes" folder in the root folder and an "index.js" file inside it.

In the "index.js" file, import express again.

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

Get a router from express by using the `Router()` method:

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

With this `router` constant, we can set up routes. We set up the route for the entry point of our application, the root folder, hence "/" :

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

The first argument is a server endpoint; it makes it possible to resolve this request as a URL in a browser. We then have to make sure there is something to be resolved on that URL, which we do with the second argument of the `get` method. The second argument is a function (a handler) that will be called every time a user requests this route.

Here, the function has **two object parameters**. We'll only use the second object for now, the "Response". In this example, we use the `send` method to simply send a string back to the user.

To use this `router` variable in another file, we need to export it:

```
module.exports = router;
```

2.3.2. Import routers in the server

Instruct the server that this route exists by importing it in the "server.js" file:

```
const indexRouter = require("./routes/index");
```

Now, using this `indexRouter` in the "server.js" file is exactly the same as using the `router` in the "index.js" file.

Tell the server which route this router is handling:

```
app.use("/", indexRouter);
```

At this point, you can refresh the page on "localhost:3000" and see the message sent by the `send` method.

2.4. Integrate routes with views

Put basic html inside the "layout.ejs" file by simply typing "!" and press ENTER.

Inside the `<body>` tag, write this:

```
<body>
    <p>Before</p>
    <%- body %>
    <p>After</p>
</body>
```

In the "views" folder, create an "index.ejs" file and simple write this inside:

```
<p>Middle</p>
```

In the "/routes/index.js" file, change the response message to be a properly rendered file:

~~res.send("Hello World");~~
```
res.render("index");
```

At this point, you can refresh the page on "localhost:3000" and see three paragraphs rendered. Two are from the "layout.ejs" file, and the middle one is from the "index.ejs" file.

2.5. Model integration

Install the library for mongoDB:

```
npm i mongoose
```

In the "server.js" file, import mongoose:

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

Connect to the mondoDB database:

```
mongoose.connect(process.env.DATABASE_URL);
```

Setup a log to check if the connection was OK or if we ran into an error:

```
const db = mongoose.connection;
db.on("error", error => console.error(error));
db.once("open", () => console.log("Connected to mongoose"));
```

We now need to integrate a .env file to our root folder to fetch the DATABASE_URL variable:

Install the library for dotenv:

```
npm i --save-dev dotenv
```

Create a ".env" file in the root folder and write:

```
DATABASE_URL=mongodb://localhost/mybrary
```

Finally, at the top of the "server.js" file, write:

```
if (process.env.NODE_ENV !== "production") {
    require("dotenv").config();
}
```

This will make sure that we only use the .env file during development, and not in production.

Now, if you simply save your files, node should tell you that everything is fine. You should see "Connected to mongoose" in your node console. Your application should still be up and running on localhost:3000

2.6. Git everything

Now that everything is set up, we want to save the skeleton of our project to a remote repository on GitHub.

Make sure your are in your root folder and initialize Git with:

```
git init
```

Create a ".gitignore" file in the root folder and add the following elements in it, so that they aren't sent to the repository:

```
node_modules
.env
```

Add every other files to your Git with:

```
git add .
```

Commit everything:

```
git commit -m "Initial Commit"
```

Create a repository called “Mybrary” (or whatever you want) on GitHub. Copy paste the following command proposed by GitHub:

```
git remote add origin git@github.com.......
```

Then push everything for the first time with:

```
git push -u origin master
```

Your files are now on GitHub. For following pushes, you simply need to `git add .` all the new files, `git commit` with a message first, and then simply run `git push`.

3. Create/Modify/Search authors

3.1. Authors pages

3.1.1. Create controllers

In the “routes” folder create an “author.js” file.

Inside it, we need to import Express again, and its router:

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

We then create a route for the “All authors” page:

```
// All authors route
router.get("/", (req, res) => {
    res.render("authors/index");
});
```

A route for the <form> that will create a new author:

```
// New author route
router.get("/new", (req, res) => {
    res.render("authors/new");
});
```

And finally, a route to send and create the data entered in the `<form>`:

```
// Create author route
router.post("/", (req, res) => {
    res.send("Create");
});
```

This is a `post` request, not a `get` since it **creates** data. Right now, we simply `send` a message to the client, we’ll deal with the actual rendering later.

In order to use this controller outside this file, we export it:

```
module.exports = router;
```

3.1.2. Create views

Create a new “authors” folder inside the “views” folder. Create an “index.ejs” file and a “new.ejs” file inside this new folder. Write some random text inside each file.

Note that each route has its own corresponding file nested inside the **authors** folder; which bears the same name as the **authors**.js file. This is good practice in MVC to work this way, because it makes it easy to figure out where things are.

3.1.3. Link controllers to the server

Like for the `indexRouter`, we now import the `router` from and call it `authorRouter`.

Inside the "server.js" file, add:

```
const authorRouter = require("./routes/authors");
```

Instruct the server to use this router when requested with the /authors route. Add:

```
app.use("/authors", authorRouter);
```

Let's look back at the "authors.js" file to see if everything look right.

```
router.get("/", (req, res) => {
    res.render("authors/index");
});
```

The first route inside it is simply `"/"`. This means that when a user requests `[website]/authors`, they will end up using `authorRouter`, and get the "authors/index.ejs" file rendered on their browser.

```
router.get("/new", (req, res) => {
    res.render("authors/new");
});
```

On the other hand, if they request `[website]/authors/new`, the server will render "authors/new.ejs".

Everything looks fine!

Save all your files (restart the server if needed with `npm run dev`) and go to "localhost:3000/authors" and "localhost:3000/authors/new" to behold you newly created pages.

3.2. Create a navbar

It's quite tedious to remember and type all the URLs that our app will use. Let's create a simple navbar containing links to all our existing pages.

Create a "partials" folder inside the "views" folder. Create a "header.ejs" file inside it.

The "partials" folder will contain pieces of webpages that can be included in other .ejs files to avoid code duplication. Think of it like user-made functions that you'd make in a program to make your code more concise.

Add the following code to your "header.ejs" file:

```
<header>
    <nav>
        <a href="/">Mybrary</a>
        <ul>
            <li><a href="/authors">Authors</a></li>
            <li><a href="/authors/new">Add Authors</a></li>
        </ul>
    </nav>
</header>
```

Modify the <body> tag inside the "/views/layouts/layout.ejs" by getting rid of the placeholders and including the "header.ejs" file:

```
<body>
<body>
    ~~<p>Before</p>~~
    <%- body %>
    ~~<p>After</p>~~
    <%- include("../partials/header.ejs") %>
</body>
```

Refresh your browser and try and use these links.

3.3. Use the database

We have several pages but nothing to do with them so far. A basic thing that it should do is save authors added by users.

3.3.1. Create a model

To add authors to our database, we first need to create a model for them.

Create a "models" folder in the root folder and a "author.js" file inside it.

This file will contain the author table, a **schema**, with its various attributes, to use with our database.

Import the mongoose library in the "author.js" file.

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

Create an `authorSchema` constant with the `new mongoose.Schema()` method and require a "name" attribute by sending a JSON as an argument:

```
const authorSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
    },
});
```

There are a ton of ways to customize a property, but `type` and `required` are fine for now.

Export this schema:

```
module.exports = mongoose.model("Author", authorSchema);
```

We can now use this model in the rest of our application.

3.3.2. Use the schema

In the "/routes/authors.js" file, import the schema and add an argument to the new author route:

```
const Author = require("../models/author");
```

```
router.get("/new", (req, res) => {
    res.render("authors/new", { author: new Author() });
});
```

Creating a `new Author()` doesn't add it to the database just yet, but it gives us an `Author` object that we can now use in our .ejs files and save to the database later.

3.3.3. Create a form to add new authors

Some fields inside the `<form>` we are going to create will be used in several forms. It's a good opportunity to create another partial .ejs file. We'll put it in the "/views/authors" folder for conveniency because it's not going to be used elsewhere.

Create a "_form_fields.ejs" inside the "/views/authors" folder and put a label and a text input inside:

```
<label>Name</label>
<input type="text" name="name" value="<%= author.name %>">
```

The `name` property sets the value under which we can reach this sent data from the "server.js" file. When we'll write `req.body.name` later on, the `name` portion is called like this because we set `"name"` here as the value of `name`.

The `value` property will be used for a future "edit author" page.

Inside the "/views/new.ejs" file, create the actual `<form>`. Make sure to include the form fields, to use the POST as a method, to have a Cancel "button" that sends back to the "/authors.ejs" file and to have a "submit button":

```
<h2>New Author</h2>
<form action="/authors" method="POST">
    <%- include("_form_fields") %>
    <a href="/authors">Cancel</a>
    <button type="submit">Create</button>
</form>
```

The `action` property sets the route that the form will take when sent. Remember that earlier, we created a post route in the "/routes/authors.js" file:

```
router.post("/", (req, res) => { ... });
```

Since the route is just `"/"`, we can indeed send this form with a POST method to the `"/authors"` route.

Also note that the "Cancel" button is a simple `<a>`nchor tag (instead of an actual button). This is good practise because `<a>` are used to link pages and <button> are use to interact with `<form>`s and other objects. Once our CSS is done, they'll both look like buttons to the end user, though.

3.3.4. Access the new authors

By default, Express.js has no easy way to access the variable we send. We need to install body-parser to make our lives easier.

Install body-parser:

```
npm i body-parser
```

In the "server.js" file, import body-parser:

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

Tell the server that we use body-parser by sending data through URLs:

```
app.use(bodyParser.urlencoded({ limit: "10mb", extended: false }));
```

The JSON we pass as argument is used to configure a bigger file size than allowed by default; this will be useful when sending images later.

Inside the "/routes/authors.js" file, modify the argument of the `send` method:

```
router.post("/", (req, res) => {
    res.send(req.body.name);
});
```

You can now see the use of the `req` argument: it holds the whole data that was sent (and more) as a request, which is our new author, in this context. To access the actual data, we reach its `req.body` property, which takes the form of a JSON object: `{"name":"John Doe"}`. Thus, we can access the value with: `req.body.name`.

Now, on http://localhost:3000/authors/new, when creating a new author, its name is being sent to the user as a string, that we can use to add to our database.