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
```

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 refresh 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 being 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

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 the route, while the second one is a function with two parameters. We'll only use the second parameter for now, the "response" parameter. Here, we use the `send` method to simply send a string.

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

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

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

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

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 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.

If you simply save your files, node should tell you that everything is fine, and 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

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:

```
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`.