Exercise 4 – Node

Install Node

Download and install the latest LTS version of node (https://nodejs.org/).

Prepare project environment

Prepare for this exercise by extracting the base solution (`Base-Solution-Node.zip`) provided on Moodle to a local directory and configuring a project in your IDE that contains these files.

In Visual Studio Code this can be achieved by adding the folder to a workspace, in WebStorm this can be achieved by opening the folder (File -> Open).

Next you need to download the project dependencies configured in `package.json` in the base solution. You can do that by running

```
$ npm install
```

in the project's root folder (where `package.json` resides).

Run the server

You can now start the server process by running

```
$ node server.js
```

in the project root folder.

Install and use nodemon

It is tedious to restart the server every time you make a change. nodemon is a tool that simplifies development by automatically restarting the node application when file changes are detected in the directory.

To install nodemon, use `npm`. This time add the `-g` parameter to install nodemon globally, that is, making it available to all your node applications (not just the one you are working on right now).

```
$ npm install -g nodemon
```

Now you can run the server application by running nodemon. Whenever you make changes, nodemon will detect them and restart the application automatically:

```
$ nodemon
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server listening at http://localhost:3000
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Server listening at http://localhost:3000
```

Configuring the port using an environment variable

If you want, you can configure the port on which the API is made available by express. Per default, port 3000 is used.

You can change the port by configuring the environment variable PORT.

On a *nix system, you can

```
$ export PORT=8000
```

On a Windows system use

```
> set PORT=8000
```

Then restart the server application.

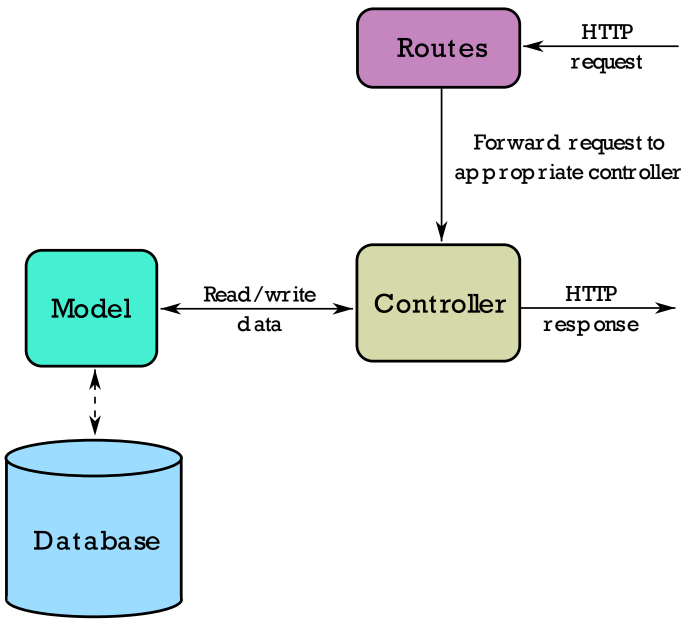
Test the API using postman

Postman is an app that simplifies the test of APIs. You can either download the app (https://www.postman.com/downloads/) or use the web version.

Review the given base solution

Make yourself comfortable with the base solution.

- **`server.js`** is the main JavaScript file, the one you run using **`node`** or **`nodemon`**
- **`server.js`** configures express to serve static files from the **`files`** directory. This directory contains a sample solution, you can replace these files if you want to work with your own implementation from exercises 1 to 3
- The **`api`** directory contains the base solution. It is implemented using a modular architecture using *Routes*, *Controllers* and *Models*. Routes to be used are configured in **`server.js`**. Configured routes then forward incoming requests to the appropriate Controller. The controller checks the incoming request and relays any modification to the data to the underlying model. Typically, the model communicates with a database, in our simple example, however, the data is stored in an in-memory data structure. You will therefore find no code communicating with a database.



Several requests are already implemented; the model also exists. Your main job it to extend the existing structure.

Tasks

Task 1. Fetch categories and books from the backend

This is the only client-side task (aside from the extra point task 😉) in this exercise. You will find details concerning this task in **`main.js`** in the **`files`** directory.

There are two endpoints already implemented on the server-side which provide information about book categories (`/api/categories`) and their books (`/api/categories/:category/books`).

Your job is to use the Fetch API to first fetch the available categories and add the corresponding links to the menu (**`ul`** in **`nav`**) of our bookstore. When clicked those links will in turn use the Fetch API to load the books of the category clicked and dynamically add the books to our **`main`** element.

Task 2. Create books to the server API

In the base solution, existing books can only be fetched using the API. The remaining three tasks are concerned with extending the API to add, modify and delete books.

In this task, you will add a new endpoint to the API, with which new books can be created. See the description in **`api/routes/book-router.js`**, **`api/controllers/book-controller.js`** and **`api/models/book-model.js`** for details.

Task 3. Update books to the server API

Again, you will add a new endpoint to the API, this one can be used to update existing books. See the description in **`api/routes/book-router.js`**, **`api/controllers/book-controller.js`** and **`api/models/book-model.js`** for details.

Task 4. Delete books to the server API

In this last task, you will also add a new endpoint to the API, this time to delete books from the model. Again, see the description in **`api/routes/book-router.js`**, **`api/controllers/book-controller.js`** and **`api/models/book-model.js`** for details.

Extra-Point. Remove books from the shopping cart

If you want to score an extra point, this is your task: Add a way to remove books from the shopping cart on the client!