Compensational Exercise

[If not existent on your system] Install Node

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

Prepare project environment

Prepare for this exercise by extracting the skeleton solution (`Skeleton-Solution.zip`) provided 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
```

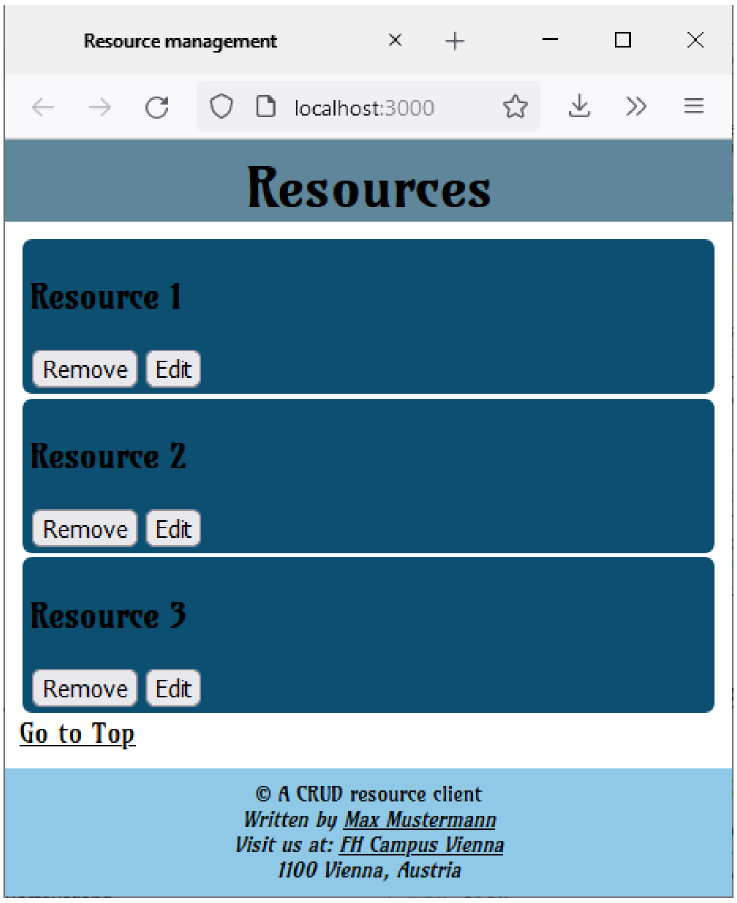
in the project's root folder (where the file `package.json` resides).

Run the server

You can now start the server process by running

```
$ node server.js
```

in the project root folder.



If you now browse to http://localhost:3000/ you see the basic application in action.

It already requested a list of resources from the server and rendered them in a list.

[Optional] 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
```

[Optional] 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
```

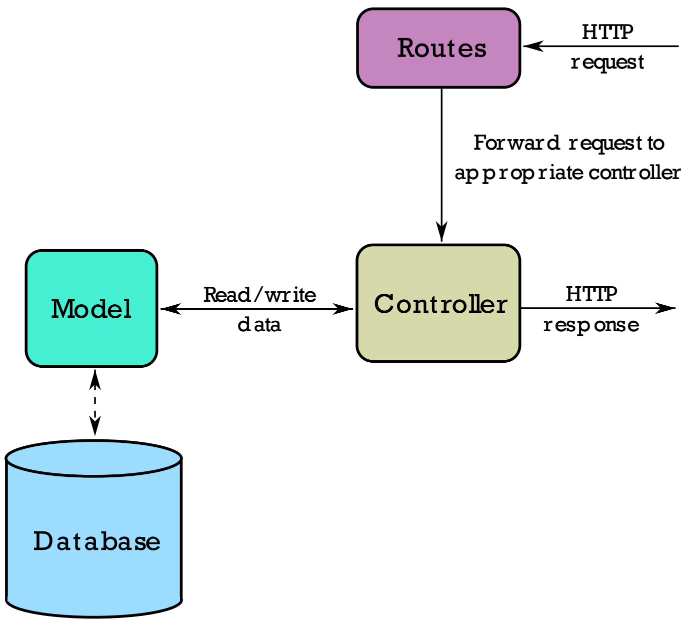
Then restart the server application.

Review the given skeleton solution

Make yourself comfortable with the skeleton 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 basic client-side solution, which is the starting point for your work, which you will mainly carry out on the client-side.

- The **api** directory contains the basic server-side 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.

  There are already five endpoints implemented: two GET endpoints and one POST, PUT and DELETE endpoint respectively; the controller and model also exist.
  Your main job it to extend the existing resource on the server- and client-side, most of the tasks will be implemented on client-side. The next page details the tasks.

Tasks

Task 1 [4 Points]. Extend the Resource class in model.js
This is the only server-side task in this exercise. You will find additional information concerning this task in **`api/models/model.js`**.

In the model you find the **`Resource`** class. It only has a name, which is initialized in the constructor. Scroll to the bottom of the file to see where the three resources you saw earlier in the browser are added to the model.

Now think of another class of objects you want to use for this exercise. This can be anything, e.g., animals, persons, ... whatever comes to mind.

Replace the Resource class with a class representing your object, e.g., an Animal class, or a Person class. Make sure that you add at least three properties to your class (and an optional fourth):

- A String property,
- a Number property,
- a Boolean property,
- and – optionally - a date property

Change the initial data added to your model so that your objects are sent to the browser. Of course you can add as many properties as you like, you are not limited to four properties.

Task 2 [3 Points]. Extend the rendering of your resource on the client-side
Now move on to the client-side, focus on **files/main.js**, where all the client-side logic resides.

In this task, you represent the properties of your resource in the DOM. Make yourself familiar with how the code works on the client-side and add HTML elements that represent the properties best. For more information on this task, refer to the details in **files/main.js**.

Task 3 [3 Points]. Call the DELETE endpoint
Now, we start to use our API. First, we are going to delete resources. The click listener on the "Delete"-Button is already in place and a call to the `remove(...)` function is made when the button is clicked.

But we must remove the resource from the server or else it is going to reappear when we reload the page. Add a DELETE request to the API using either an `XMLHttpRequest` or the `Fetch API`. Once this asynchronous API call is finished, call the remove(...) function (that at the moment is called synchronously) to remove the resource from the DOM also.

Task 4 [3 Points]. Add the possibility to edit the resource
This task is the most complex one, it consists of three parts.

- Part 1. Adapt the form that is already added to the DOM. Remove the name from the example resource and add input elements that reflect the properties your resource has. Add an id to each of your input elements so that you can identify it easily in Part 2.
- Part 2. After the user clicks "Save", we transfer the values the user entered to our resource. We must do that manually for each property. Remove the transfer of the name property of the example property add transfer the properties your resource has. Here you use the ids you set on your input elements to locate them in the DOM.
- Part 3. The last part. Call the PUT endpoint using either `XMLHttpRequest` or the `Fetch API`. Like in Task 3, we quit the edit mode only **after** the PUT call returns. That is, you call the `add(...)` method only after the calls terminates.

Task 5 [3 Points]. Add a way to create a resource
Reusing code from Task 4, it is easy to add a "Create" button to the application. Do that using the POST endpoint available!