Lab Guide for Node/Express

Routes and REST

The main support for this guide is the express manual: https://expressjs.com/en/5x/api.html

You should use the manual manly as a reference guide, but you can read the topics bellow before you do the guide to get a better grasp of the fundamental functionalities of Express.js.

This guide is a first step to create the server functionalities needed for the web pages of the previous tutorials using some of the concepts of node.js and express.js:

- Express generator (already used in the installation guide): https://expressjs.com/en/starter/generator.html
- Express.js routes:
  - Basic routing: https://expressjs.com/en/starter/basic-routing.html
  - More on routing: https://expressjs.com/en/starter/basic-routing.html
- REST. Two sites that explain what is REST:
  - https://www.restapitutorial.com/lessons/whatisrest.html
  - https://www.codecademy.com/articles/what-is-rest
- REST documentation: https://bocoup.com/blog/documenting-your-api

Open the project that you used for the previous lab guides ("alunos" project unless you choose another name). Since we are doing the server part, you can do this guide without finishing the previous ones, but you need at least to have read the guides since we will be talking about the same examples.

1. Design your REST API that will support the functionalities already existent in the web pages. For each rest endpoint you will need to define:
   - A title and short description that will allow any user to understand what functionality is associated to that endpoint. Example:
     - **Get a product by id**. Retrieves the product information that corresponds to the received id.
   - The URL of the endpoint. All URLs should start by "/api/" followed by the name of the resource usually a plural name (ex: users, products, etc). You should also represent in the path the parameters that are part of the path. Example:
     - **/api/products/:id**

     :id will be the id of the product and will allow to request different products for the same endpoint
   - The method: get, post, put, delete, patch, etc

   You do not need to write all the previous information to start the server, but you should at least organize in writing all the titles, paths and methods so you have a clear idea of your REST API. If you do write the full information you will have much less work to do in the end, since this will be the documentation of your REST API (the documentation includes some more sections for each endpoint).

   Now let's remember the functionalities we will need for our server:
   - Obtain the list of all units
   - Obtain the list of all students
   - Obtain the information of one student, including the units and grades of that student
   - Submit a grade for one student in one unit

SPOILER: Next pages will have most of the resolution so finish this task before changing page

2. We will need two REST resources, one for units and another for students. We will start by the one for units. This will define a route file with only one rule, a get rule with url "/api/units"

- Create a "models" directory with a "unitsModel.js" file inside
    - This file should have a variable with the array of units. Create an array with 5 units: Mathematics, Literature, Laws, Informatics and Cooking. For each unit object add a semester and the number of ECTS. (you should have a similar array in the "studentGrades.js")
    - It should export a function getAllUnits that returns that array
- Create the "unitsRoutes.js" file in the "routes" directory.
    - Import the model file
    - Create route object.
    - Create the get route with path "/" (this is the "local" path inside the resource). The route calls the getAllUnits and sends the array
- Add the route you just defined to the app in the "app.js" file.
    - Import the route to a variable
    - Add the route to the App using "/api/units" as the URL for the resource

Test this rule in the browser

Remember Express Routes

- Each REST resource should have its own routes and model file with all the rules for the corresponding endpoints. All model files should be in the models directory and all route files in the routes directory.
- The model file should have the data and export functions that manipulate that data:

```
var products = [  { name: "Potatoes", price: 1.2 },
                  { name: "Onion", price: 0.65 }  ];

module.exports.getAllProducts = function() { return products; }
```

- The route file should import the model file and create the route object. After defining the endpoints for that resource it will export the route object

```
var express = require('express');
var router = express.Router();
var mProd = require("../models/productsModel");

router.get("/", async function(req,res,next) {
   let products = await mProd.getAllProducts();
   res.send(products);
}

module.exports = router;
```

Using async/await will prepare our routes to manage database calls

The function detaches from the main node process with async and waits for calls to the database

- Finally, you need to add the route object to the application in the app.js file. It is in in here that you will define the path for the resource (in this case /api/products) , to which all rule paths in the router will be concatenated

```
var productsRouter = require('./routes/productsRoutes');

app.use("/api/products",productsRouter);
```

3. Do the same thing for the get all students endpoint. This corresponds to the students resource.

- Create the “studentsModel.js” model file for the resource with the unit arrays and a method getAllStudents
    - The array should have students and their corresponding units and grades
      You can adapt the previous arrays for students and grades (student information in “students.js” and unit and grade information in “studentGrades.js”) or see the previous guide to create it
- Create the “studentsRoutes.js” routes file with the get rule that returns all studens (“local” path “/”)
- Add the route to the App using path “/api/students”

Test this rule in the browser

4. Create the endpoint to get one student given its position in the array. The file for this resource is already created, you only need to create the function in the model file and add a rule to the routes file.

- In the model file create a function called getStudent that receives a position and returns the student in the corresponding position of the array
- In the routes file create a rule which will receive the position as parameter. The URL will be “/:pos”:
    - Retrieve the parameter and call the getStudent function with it
    - Send the result back to the client

  The endpoint will be “/api/students/:pos” concatenation the path of the resource “/api/students” with the path of the rule “/:pos”

Test this rule in the browser

Remember Express Route parameters

- A rule can use parameters embedded in the path. Each part of the path that is a parameters uses the “:” and a name that will correspond to the property that will hold the value (express will process the path and create the variables for you)

- Imagine that in the previous example we want to retrieve one specific product given the position of the product in the list. We will need to define a path with a parameter for the positions “/api/products/:pos” and we can use that parameter to fetch the product.

    - In the models file we need to add another function to get the product on that position

```
module.exports.getProduct = function(pos) { return products[pos]; }
```

    - In the routes file we need to retrieve the parameter and call the previous function

```
router.get(“/:pos”, function(req,res,next) {
   let pos = req.params.pos;
   let product = await mProd.getProduct(pos);
   res.send(product);
}
```

All parameters in the path, every name with “:”, will correspond to a property of the object in “req.params”. So if we request the URL localhost:3000/api/products/1 the result will be the JSON of the second product:

```
‘{ “name”: “Onion”, “price”: 0.65 }’
```

5. Finally, we will create the endpoint to save the student grade for a unit.

This can be considered a post or put, post if we consider we are inserting a new grade, put if we considered the unit would already be in the student and we would be only editing its grade.

We will consider it a post rule, the unit will be added if the unit does not exist in the list, if it exists the rule will edit the grade to the new one.

Since it is a post rule we will consider all the unit with grade information will be received inside the body (an unit object equivalent to the ones that were already associated with the student in the previous rules), but to match the student we will require an URL parameter with its position in the list.

- Create the saveGrade function in the model file that receives a position and a unit object (with unit name and grade).
    - Obtain the student that is in the received position
    - If the unit as a name that is equal to any unit in the list change its grade to the new one and return an object:
      { msg: “Changed grade of unit <unit name>” }
    - If the unit does not exist add the unit object to the list and return the object:
      { msg: “Added grade for unit <unit name>” }

- Create a new post rule with path “/:pos” (notice this path is equal to the previous one but we can have equal paths if the method is different).
    - Retrieve the parameter pos
    - Retrieve the unit object from the body
    - Call the saveGrade function with both parameters
    - Return the result from saveGrade to the client

6. You cannot test this endpoint directly in the browser since it is a “post” endpoint and you can only make “get” requests in the browser URL.

You can either create a form to call the endpoint or you can use postman to test this endpoint like it was shown in classes.

(more information in the next page)

Remember Express Route body (post and put methods)

- Post and put endpoints will usually receive information in the body that can be retrieved from the request using req.body. Let's consider new endpoints to the previous example, one to add a product and another to change the product price.

- We will need to define both functions in the "productsModel.js":

```
module.exports.saveProduct = function(product) {
   products.push(product);
   return { msg: "Product inserted" };
}

module.exports.changePrice = function(pos,price) {
   products[pos].price = price;
   return { msg: "Product price changed" };
}
```

  For the saveProduct we only added the object to the list (we consider the object will be well formatted).

  For the changePrice we just change the price property of the product in the received position

- In the "productsRoutes.js" file we will need to add 2 more endpoints:

```
 router.post("/", function(req,res,next) {
   let newProd = req.body;
   let result = await mProd.saveProduct(newProd);
   res.send(result);
}

router.put("/:pos/price/", function(req,res,next) {
   let priceObj = req.body;
   let pos = req.params.pos;
   let result = await mProd.changePrice(pos,priceObj.price);
   res.send(result);
}
```

  The first endpoint is to add a new product, since it is new information we defined a post rule, retrieved the data from the body (a new product object) and called the saveProduct function.

  The second endpoint is to change the price of the product, since it is changing existing information, we used a put rule. We retrieved the position that corresponds to the object to change from the parameters and the information about the new price from the body. We consider that the information about the price will be in an object with the format { price: 12 } so we called the changePrice with the position and the price property of the object retrieved from the body.

The End