SC-T-213-VEFF - Web Programming I

Assignment 3 – RESTful Backend

Reykjavik University

Deadline: **20th March 2023, 23:59**

The topic of this assignment is: **Writing a RESTful Backend.**

This is a **group assignment**, which you can solve alone or in groups of **up to 3 students**.

1. Overview

In this assignment, you will develop a backend that can be used for keeping track of tunes and genres. The idea is the same as in assignment 2 and, in fact, your backend needs to be compatible with a modified solution of assignment 2. You will provide two kinds of resources with several endpoints that follow the REST principles and best practices. Please note that Postman can be a huge asset when developing solutions like this one.

Note: Similar to Assignment 2, this task might seem overwhelming at first. Start by designing the API following the lecture slides for L15/16 (writing down the HTTP methods and URLs for each endpoint). Then, implement the easy endpoints first (e.g., read requests are typically the easiest), use dummy data in the beginning (e.g., hard-coded arrays in the sample project).

2. Resources

For this assignment, we require two resource types: *tunes* and *genres*. Tunes represent a collection of notes and their timing. Genres represent a way to group tunes into different categories. A tune has the following attributes:

1. *id* - A unique number or string identifying each tune.
2. *name* - A string providing a heading/title for the tune.
3. *genreId* - An id of the genre this tune belongs to.
4. *content* - An array of objects containing the tune. Each object in this array has a *note* in string format, a *duration* in string format, and an integer *timing* relative to the start of the tune. The *note* and *duration* have to be compatible with the Tone.js library used in assignment 2.

A genre has the following attributes:

1. *id* - A unique number/string identifying each genre.
2. *genreName* - A string describing the genre.

You are free to implement these two resources as you like in your backend. However, the backends need to return the required attributes in the required format.

3. Endpoints

The following endpoints shall be implemented for the ***tunes***. In all but the first endpoint, a tune shall be treated as a sub-resource of a genre.

1. **Read** all tunes

   Returns an array of all tunes. For each tune, only the *id*, the *name*, and the *genreId* are included in the response. Additionally, providing the *filter* query parameter returns only the tunes that are in the genre with the provided name. If no tune is in the provided genre, or no genre with the provided name exists, an empty array is returned. Returning an empty array is important so the frontend can expect an array, no matter the result.

2. **Read** an individual tune

   Returns all attributes of the specified tune.

3. **Create** a new tune

   Creates a new tune. The endpoint expects the *name* and *content* of the tune. Duplicate names are allowed. The *genreId* is provided through the URL. The *content* attribute must be a non-empty array of objects, each of which has the *note*, *timing*, and *duration* attributes. The *id* shall be auto generated (i.e., not provided in the request body). The request shall fail if a genre with the given *id* does not exist. The request, if successful, shall return the new resource (all attributes, including *id* and the *full* content array).

4. **Partially update** a tune

   Partially updates an existing tune. All attributes but the *id* can be updated. All attributes that are provided in the request body are updated. If a *content* array is provided, it is always completely replacing the existing one. The request, if successful, returns the updated resource. To change the genre, the request expects the old *genreId* in the URL, and the new *genreId* in the request body.

The following endpoints shall be implemented for the ***genres***:

1. **Read** all genres

   Returns an array of all genres (with all attributes).

2. **Create** a new genre

   Creates a new genre. The endpoint expects only the *genreName* attribute in the request body. The unique *id* shall be auto-generated. Duplicate names for genres are not allowed. The request, if successful, shall return the new genre (all attributes, including *id*).

3. **Delete** a genre

   Deletes an existing genre. The request shall fail if any tunes exist in this genre. The request, if successful, returns all attributes of the deleted genre.

4. Requirements

The following requirements/best practices shall be followed:

1. The (unmodified) frontend code provided as supplementary material needs to work with the application. Note that this assignment uses different endpoint specifications, so the regular solution (and your solution) of assignment 2 will not work with this assignment's solution.

2. As in Assignment 2, you should opt to use arrow functions when possible, do not use "var" and use inbuilt JavaScript functions over generic for-loops etc.

3. The application shall adhere to the REST constraints.

4. The best practices from L15/16 shall be followed. This means that

- Plural nouns shall be used for resource collections
- Specific resources shall be addressed using their ids, as a part of the resource URL.
- Sub-resources shall be used to show relations between genres and tunes, as stated in Section 3.
- JSON shall be used as a request/response body format
- The HTTP verbs shall be used to describe CRUD actions. The safe (for GET) and idempotent (for DELETE and PATCH) properties shall be adhered to.
- Appropriate HTTP status codes shall be used for responses. 200 should be used for successful GET, DELETE and PATCH requests, 201 for successful POST requests. In error situations, 400 shall be used if the request was not valid, 404 shall be used if a resource was requested that does not exist. 405 shall be used if a resource is requested with an unsupported HTTP verb (e.g., trying to delete all tunes), or if a non-existing endpoint is called.
- You are not required to implement HATEOAS/Links.

5. The application/backend shall be served at http://localhost:3000/api/v1/ In case you have issues running your backend on port 3000, contact us - please do not change the port in the solution code.

6. The application shall be written as a Node.js application. A *package.json* file shall be part of the project, including the required dependencies.

7. The application shall be started using the command *npm start* (you must not remove the possibility to use npm start).

8. The application is only permitted to use in-built modules of Node.js, as well as Express.js, body parser, and cors[1].

9. You are not supposed to/allowed to add persistence (a database, file storage, or similar) to the assignment.

10. There are no restrictions on the ECMAScript (JavaScript) version.

[1]The *cors* module enables cross-origin resource sharing (CORS). This is not required for this assignment, but it makes sure that requests are not blocked in case you try out your backend using a browser.

5. Sample Project

In the supplementary material to this assignment, you will find a sample Node.js project (a package.json file and an index.js file). The index.js currently only includes two variables that contain examples for the resource format defined in Section 2. You may change the internal representation of the resource and/or add additional data structures as needed. You should extend the sample code to include your backend code, additional files are of course permitted. The dependencies to express, body-parser, and cors are already defined in the package.json - you can simply install the modules by running npm install in your project directory.

6. Starting the project in “development mode”

1. Navigate to the **prj03_suppl/prj03_starter** folder in a terminal window.
2. Install the NPM packages listed in **package.json** by running the **npm install** command (**npm i** does the same thing). This will fetch all the required packages needed to run the project.
3. To start up the process, you can run the “**npm run dev**” command to start the process **in development mode**, which in our case means that the process will be watching index.js for changes. When the code is updated, the process will automatically restart to reflect the changes made to the code. The process will print out a message telling you it has started:

```
> prj03_starter@0.0.1 dev
> nodemon index.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Tune app listening on port: 3000
```

4. Your backend is now running http://localhost:3000 and can be accessed via Postman, cURL or your website.

7. Submission

The assignment is submitted via Canvas. Submit a zip file containing your entire node project. Do **NOT** include the *node_modules* folder, doing so will result in a deduction.
No late hand-ins are accepted.