Music Library v2

The Music Library project has evolved into a Spring Boot project, running in a Tomcat server on port 8091, and now it needs a REST API.

Just like most other Spring Boot projects, it contains four layers:

- Controller : containing the endpoints
- Service : containing the business logic
- Repository : containing database actions
- Model : containing models of the data

The last three are already fully implemented, so your job is to define the endpoints.

All endpoints should be defined in LibraryController.java.

You should not modify any of the other files. You're free to do so if you want to experiment, but this task is possible to complete by only working on LibraryController.

You should only use the methods from LibraryService in your endpoint methods.

The LibraryController class should be annotated as a Rest Controller, and it should have a global (class-level) mapping to /api/v1, meaning all endpoints defined inside it will automatically get that prefix (hint: use @RequestMapping on the whole class).

Next, you should Autowire the LibraryService Service (the @Service annotation means it's a Spring Component). That way you will have a LibraryService object ready to use in your LibraryController.

Once you have that object, you will be able to call its methods inside your endpoint methods and thus implement their functionality.

Each method in LibraryService corresponds to an endpoint that needs to be created.

REST API SPECIFICATION

GET /artists

Returns a list of all artists.

Optional: if query parameter namesOnly=true, then return a list of all artists' names.

e.g. GET http://localhost:8091/api/v1/artists

e.g. GET http://localhost:8091/api/v1/artists?namesOnly=true

POST /artists

Creates a new artist.

Expects a JSON body representing an Artist object, containing only the artist's name.

body: { "name":"STRING" }

e.g. POST http://localhost:8091/api/v1/artists --body: { "name":"Kanye West" }

PUT /artists/{id}

Updates an existing artist's properties (which is only the name).

The path contains a variable which is the artist's ID.

Expects a JSON body representing an Artist object, containing only the artist's name.

body: { "name":"STRING" }

e.g. PUT http://localhost:8091/api/v1/artists/2 --body: { "name":"Ye" }

DELETE /artists/{id}

Removes an existing artist.

The path contains a variable which is the artist's ID.

e.g. DELETE http://localhost:8091/api/v1/artists/2

GET /tracks

Returns a list of all tracks.

```
POST /artists/{artistId}/tracks
```

Creates a new track for a specific artist.

The path contains a variable which is the artist's ID.

Expects a JSON body representing a Track object, containing name and year.

body: { "name": "STRING", "year": "NUMBER" }

e.g. POST http://localhost:8091/api/v1/artists/2/tracks --body: { "name":"Heartless", "year": 2008 }

```
PUT /artists/{artistId}/tracks/{trackId}
```

Updates an existing track for a specific artist.

The path contains two variables: the first one is the artist's ID, the second one is the track's ID (unique for each artist).

Expects a JSON body representing a Track object, containing name and year (both are optional).

body: { <"name">: "STRING", <"year">: "NUMBER" } (<> means optional)

e.g. PUT http://localhost:8091/api/v1/artists/2/tracks/0 --body: { "name":"Heartless (Remix)", "year": 2010 }

```
DELETE /artists/{artistId}/tracks/{trackId}
```

Removes an existing track from a specific artist.

The path contains two variables: the first one is the artist's ID, the second one is the track's ID (unique for each artist).

e.g. DELETE http://localhost:8091/api/v1/artists/2/tracks/0

Extras

Some functions in LibraryService return int values, depending on the success of the operation.

In case of complete success, 0 is returned; otherwise, they may return -1, -2, or -3, depending on the case. You can find all the details in each function's Javadoc.

You can utilize those return values to make your endpoints catch certain errors and return a specific status code and message. For example, if the client tried to create a new Artist, but another artist already had that name, the server could respond with 403 and the message "Artist with that name already exists".

You can also use my collection of ready-made Postman requests for testing your API, if you don't feel like creating your own requests.

I've included two JSON files in the project, *PracticeAssignment4.postman_collection.json* and *PracticeAssignment4.postman_environment.json*, which can be used to import the Postman Environment and Request Collection into your Postman.

Just open Postman,

click on **Import**,

select the two files,

click the other Import button that shows up,

and they'll be imported into your current Workspace.

Lastly, you need to select the newly imported Environment:

in the top-right corner there is a control that says "No Environment", so just click on that and select "ProjectAssignment4Environment".