Web Applications [2024/2025]

Lab 3: APIs with Express

This week you will create a **basic back-end for your FilmLibrary**. To do so, you will use the Express framework. The back-end must implement a series of **APIs** to support the main features of the FilmLibrary you have developed so far: **create**, **read**, **update**, and **delete** the films. The data will be persistently stored in an SQLite database (handled by the server as a file).

1. API design

Design a set of APIs to support the main features of your FilmLibrary. Data *passed to* or *received from* the API should be in **JSON format**. Implement the following APIs (the most important ones are listed first):

- **Retrieve** the list of all the available films.
- **Retrieve** a film, given its "id".
- **Create** a new film, by providing all relevant information – except the "id" that will be automatically assigned by the back-end.
- **Mark** an existing film as favorite/unfavorite.
- **Change the rating** of a specific film by specifying a delta value (i.e., an amount to add or subtract to the rating, such as +1 or -1). Only ratings which are not null can be changed.
- **Delete** an existing film, given its "id".
- **Retrieve** a list of all the films that fulfill a given filter among these:
    - favorite: only films marked as favorite;
    - best: only films whose score is five out of five;
    - lastmonth: only films watched between today and the last 30 days;
    - unseen: only films without a watch date.

    Hint #1: the filter can be represented as the (string) value of a query parameter

    Hint #2: this API can be merged with the one that retrieves the list of all available films, using a suitable filter for the default case ("all") when no filter is specified.
- **Update** an existing film, by providing all the relevant information, i.e., all the properties except the "id" will overwrite the current properties of the existing film. The "id" will not change after the update. It must be possible to update only a part of the properties leaving the others unaltered.
    - **Hint**: in the API implementation, retrieve the film object before updating, overwrite the fields with the values coming from the request, them write the object to the database.

List the designed APIs, together with a short description of the parameters and the exchanged entities, in a *README.md* file. Be sure to identify which are the collections and elements you are representing. You might want to follow this structure for reporting each API:

```
[HTTP Method] [URL, optionally with parameter(s)]
[One-line about what this API is doing]
[Sample request, with body (if any)]
[Sample response, with body (if any)]
[Error response(s), if any]
```

Example in markdown (.md) format (for more info: https://www.markdownguide.org/getting-started/)
(Note that Visual Studio Code has a preview mode (right-click on file) to see how markdown is rendered)

````
#### Get all films

* `GET /api/films`
* Description: Get the full list of films
* Request body: _None_
* Response: `200 OK` (success)
* Response body: Array of objects, each describing one film. Note that absence
of values is represented as null value in json.

``` json
[
  { "id": 1, "title": "Pulp Fiction", "favorite": 1, "watchDate": "2023-03-11",
"rating": null, },
  ...
]
```

* Error responses:  `500 Internal Server Error` (generic error)
````

2. API implementation

Implement the designed HTTP APIs with Express. The films must be **stored persistently** in an SQLite database. To this end, you can use the provided database "films.db" (see hints below). The database contains one table: "films". The film entries have all the fields already described so far (id, title, favorite, watchDate, rating).

2.1 Validation

This will be one of the topics for next lab. However, you can already start adding some basic validation, such as checking if an id is an integer value, etc.

3. API testing

Test the implemented APIs with the **REST Client extension** for Visual Studio Code. To this end, you will have to write an API.http file according to the following syntax:

```
[HTTP Method] [URL, optionally with parameter(s)] HTTP/1.1
content-type: application/json (if needed)

[Sample request, with body (if any)]
###
```

Note the empty line before the sample request with body.

Example of an API.http file (Note that the REST Client extension will create a small "Send Request" link before the URL, that you can click to test the server.

```
## Retrieve all the films.
GET http://localhost:3001/api/films HTTP/1.1
###

## Create new film (without id)
POST http://localhost:3001/api/films HTTP/1.1
content-type: application/json

{
    "title": "Guardians of the Galaxy Vol.3",
    "favorite": 1,
    "watchDate": "2024-02-09",
    "rating": 4
}
###
```

Hints:

1. The file "**`films.db`**" is included in the repository available on GitHub, including a script to run in "*DB Browser for SQLite*" to recreate/modify it if needed: https://github.com/polito-WA-2024/labs-code/tree/main/lab03-express
2. As you saw in the lectures, you can use the SQLite database by means of the following module: **sqlite3** (https://www.npmjs.com/package/sqlite3) – the basic library.
3. To browse the content of the database, you can use one of the two following approaches:
    a. Download the Visual Studio Code *SQLite extension* (you can search for it in VSCode extension hub or browsing the following link): https://marketplace.visualstudio.com/items?itemName=alexcvzz.vscode-sqlite and/or the SQLite Viewer https://marketplace.visualstudio.com/items?itemName=qwtel.sqlite-viewer NOTE: In Linux, you may need to install the `sqlite3` native software package of your Linux distribution to make it work.
    b. Download the application *DB Browser for SQLite*: https://sqlitebrowser.org/dl/
4. Create a back-up copy of the database before testing your APIs.
5. Visual Studio Code **REST Client** extension is available at the following link: https://marketplace.visualstudio.com/items?itemName=humao.rest-client