Distributed Systems Programming

A.Y. 2025/26

Laboratory 1

The two main topics that are addressed in this laboratory activity are:

- design of JSON schemas;
- design of REST APIs.

In order to have a complete experience, an implementation of the designed REST APIs will also be developed, by completing an already existing implementation.

The context in which this laboratory activity is carried out is a *Film Manager* platform, where users can keep track of the films they have watched and the reviews that they have written for them.

The tools that are recommended for the development of the solution are:

- *Visual Studio Code* (https://code.visualstudio.com/) for the validation of JSON files against the schemas, and for the implementation of the REST APIs;
- *OpenAPI (Swagger) Editor*, extension of *Virtual Studio Code*, for the design of the REST APIs;
- *Swagger Editor* (https://github.com/swagger-api/swagger-editor) for the automatic generation of a server stub;
- *PostMan* (https://www.postman.com/) for testing the web service implementing the REST APIs;
- *DB Browser for SQLite* (https://sqlitebrowser.org/) for the management of the database.

The Javascript language and the Express (https://www.npmjs.com/package/express) framework of the Node.js platform are recommended for developing the implementation of the RESTful web service**.**

1. Design of JSON schemas

The first activity is about the design of JSON schemas for three core data structures of the *Film Manager*, i.e., the *users* who want to manage their film lists by means of this application, the *films* they have watched and/or reviewed, and the *review invitations* that

a user may issue to another user. All the design choices for which there are no specific indications are left to the students.

A *user* data structure is made of the following fields:

- *id*: unique identifier of the user data structure in the *Film Manager* platform (mandatory);
- *name*: username of the user;
- *email*: email address of the user, which must be used for the authentication to the platform (mandatory, it must be a valid email address);
- *password*: the user's password, which must be used for the authentication to the platform (the password must be at least 6 characters long and at most 20 characters long).

A *film* data structure is made of the following fields:

- *id*: unique identifier of the *film* data structure in the *Film Manager* platform (mandatory);
- *title*: textual title of the film (mandatory);
- *owner:* the id of the film data structure owner, i.e., the user who created it (mandatory);
- *private*: a Boolean property, set to true if the *film* data structure is marked as private, false if it is public (mandatory). A *film* data structure is said private if only its owner can access it, public if every user can access it;
- *watchDate*: the date when the film was watched by the owner, expressed in the YYYY-MM-DD format (YYYY is the year, MM is the month, DD is the day). This property can be included in the *film* data structure only if *private* is true;
- *rating*: a non-negative integer number (maximum 10) expressing the rating the owner has given to the film. This property can be included in the *film* data structure only if *private* is true;
- *favorite*: a Boolean property, set to true if that film is among the favourite ones of the user, false otherwise (default value: false). This property can be included in the *film* data structure only if *private* is true.

A *review* data structure is made of the following fields:

- *filmId*: unique identifier of the film for which a review invitation has been issued (mandatory);
- *reviewerId*: unique identifier of the user who has received the review invitation (mandatory);

- *completed*: a Boolean property, set to true if the review has been completed, false otherwise (mandatory)*;*
- *reviewDate*: the date when the review has been completed by the invited user, expressed as string in the YYYY-MM-DD format (YYYY is the year, MM is the month, DD is the day). This property can be included only if *completed* is true, and in that case it is mandatory;
- *rating*: a non-negative integer number (maximum 10) expressing the rating of the review the user has completed. This property can be included only if *completed* is true, and in that case it is mandatory;
- *review*: a textual description of the review (it must be no longer than 1000 characters). This property can be included only if *completed* is true, and in that case it is mandatory

The JSON Schema standard that must be used for this activity is the Draft 7 (http://json-schema.org/draft-07/schema#).

After completing the design of the schemas, it is suggested to write some JSON files as examples, and to validate them against the schemas in Visual Studio Code. In this development environment, validation errors are shown in the editor and in the "Problem" view. You can access this view in two alternative ways:

- following the path View -> Problems;
- pressing Ctrl+Shift +M.

2. Design and implementation of REST APIs

The second activity is about the design of REST APIs for the *Film Manager* platform. Specify and document your design of the REST APIs by means of the "OpenAPI (Swagger) Editor" extension of Visual Studio Code. For the design, you should reuse the schemas developed in the first part of the assignment, customizing them for being used in the REST APIs. Then, the resulting OpenAPI document can be used as the starting point to develop an implementation of the designed REST APIs in a semi-automatic way: after importing the OpenAPI file to the stand-alone Swagger Editor, you can automatically generate a server stub, corresponding to the design, to be filled with the implementation of the requested functionalities. Since the focus of this course is on the design of REST APIs, and not on the implementation of the application logic, the implementation of some of the necessary functionalities has been made available as part of the laboratory material. You

can find this implementation in the Lab01 repository in the polito-dsp-2025-2026 organization (you will have access to the repository starting from the second session of the laboratory activity, which is focused on the REST APIs). More details regarding implementations are given at the end of this document.

The platform's REST APIs and their implementation must respect the following specifications.

The *Film Manager* platform allows users to track information about the films they have watched, or for which they want to issue a review invitation or for which they must perform a review. Two key concepts are film *owner* and film *reviewer*. The former is the user who creates the *film* data structure in the platform; the latter is a user who is in charge of carrying out a review for the film, after having been invited by the owner to make it. The platform maintains information about the users who are enabled to use it and their films in a database. Each user is authenticated by means of a personal password, which, as common practice recommends, is not stored in the database as plain text. Instead, a salted hash of the password is computed and stored in the database. The salt must be random and at least 16 bytes long.

Most of the features of the platform can be accessed only by authenticated users. The only operations that can be used by anyone without authentication are:

- the authentication operation itself (login);
- the operation to retrieve the list of all the *film* elements that are marked as public;
- the operation to retrieve a single public *film* element;
- the operation to retrieve the list of the reviews of a public *film*;
- the operation to retrieve a single review of a public *film*.

For authentication, the user sends the email and password to the platform, which checks if these credentials are correct. Emails used for authentication must be unique on the platform. The authentication mechanism that is used by the platform is a session-based authentication, where the session data are saved server-side, and the client receives a session ID in a cookie. The management of this session-based authentication mechanism can be implemented by using the Passport middleware (http://www.passportjs.org/) and the *express-session* module (https://www.npmjs.com/package/express-session). For this authentication mechanism and authenticated communication to be secure enough, it would be necessary that the REST APIs are made available on HTTPS only. However, in

the implementation produced for this Lab, which is not for production, we will expose the APIs on HTTP to facilitate debugging.

An authenticated user has access to the CRUD operations for the *film* elements:

- The user can create a new film. If the creation of the film is successful, the platform assigns it to a unique identifier. The creator of the film becomes its owner. The film review invitation to the users is explained later in this document.
- The user can retrieve a single existing film, if one of the following conditions is satisfied:
  1) the film is marked as public;
  2) the user is the owner of the film.

  The user can also retrieve the list of all the films that she created, and the list of all the films that she has been invited to review.
- The user can update an existing film if she is the owner of the film. However, this operation does not allow changing its visibility from public to private (and vice versa).
- The user can delete an existing film, if she is the owner of the film.

A central feature of the *Film Manager* platform is issuing invitations to review public films to the users who have access to the platform. The main operations related to this feature are:

- The owner of a public film can issue a review invitation to a user (the reviewer may be the owner herself).
- The owner of a film can remove a review invitation if the review has not yet been completed by the reviewer.
- A reviewer invited for a film can mark the review as completed, also updating the review date, the rating and the textual description of the review.

Review invitations for a film can be issued to multiple users at the same time. Each user who has received a review invitation can mark their review for that film as completed.

Finally, the platform must provide an additional operation that automatically issues review invitations for all films without any existing invitations, ensuring that the invitations are distributed evenly among users (so that each user receives approximately the same number of invitations). While the design of this feature is mandatory, its implementation is optional.

Here are some recommendations for the design and development of the REST APIs:

- When a list of films is retrieved, a pagination mechanism is recommended, to limit the size of the messages the platform sends back.
- When the users send a JSON *film* element to the platform (e.g., for the creation of the film in the database, or for the update of an existing film), this input piece of data should be validated against the corresponding JSON schema.
- The platform should be HATEOAS (Hypermedia As The Engine of Application State) compliant. Hyperlinks should be included in responses to enable link-based navigation and features should be self-describing. For example, when the *Film Manager* platform sends back a JSON *film* or *user* object, this should include a *self* link referring to the URI where the resource can be retrieved by a GET operation. Moreover, a client should be able to perform all operations without having to build URIs.

Guidelines for the solution development

You can find a guide for implementing the second part of the laboratory in the README file found in the Lab01 repository in the polito-dsp-2025-2026 organization (you will have access to the repository starting from the second session of the laboratory activity). Here we explain some further details.

To simplify this task and allow you to focus on design-related activities, most of the implementation has already been provided through the uploaded laboratory materials. In particular, you are provided with:

- index.js – updated with most of the necessary preambles (e.g., module imports, Passport middleware initialization, CORS setup)
- passport-config.js – for managing authentication with the Passport middleware
- package.json – updated with the required dependencies
- database folder – containing the MySQL database with preconfigured tables
- components folder – containing object definitions for the main resources
- service folder – containing the documented implementation of several useful methods that can be used to populate the automatically generated controllers

The first step in the development of the implementation of the Film Manager platform is to generate the Node.js server stub using the guide in the README file or in the Installation Note document.

Next, you should integrate the content of the repository we provided you for this laboratory into the server stub you just generated. In particular, you should copy all files and folders from the repo into the generated server stub and replace duplicates in the stub with the ones from the repo (package.json, index.js, utils/ folder, service/ folder).

After integrating this repository into the generated server stub, you should complete the remaining parts of the implementation:

1. index.js – implement the API routes and set up the JSON validator according to the instructions in the TODO comments.
2. components/ – replace all occurrences of the placeholder "/change/me" with the correct API URLs.
3. controllers/ – implement your controller logic and connect it to the routes in index.js.

Routing

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on). More specifically, some *route methods* must be defined: a route method is derived from one of the HTTP methods and is attached to an instance of the Express class. Hence, you need to introduce the required route methods in order to run the server.

Let us suppose that you must map a GET method to the "/api/films/public" path; additionally, in the corresponding route method, the callback function that must be invoked is *getPublicFilms*, located in the "controllers/Apifilmspublic.js" file automatically generated by Swagger Editor.
To manage the routing of this GET method, the file that must be modified is "index.js". More specifically, after creating the Express *app* object, you need to perform two operations:

1) importing the module represented by "controllers/Apifilmspublic.js" using the *require* function, and thus getting an object (*filmspublicController*) which gives access to the exported functions of "controllers/Apifilmspublic.js";
2) creating the routing method for the GET operation and mapping the path "/api/films/public" to the callback "filmspublicController.getPublicFilms".

In the index.js file you can find a TODO comment where to put your route definition and some additional hints on how to proceed.

Validation

When the users send a JSON *film* element to the platform (e.g., for the creation of the film in the database, or for the update of an existing film), this input piece of data should be validated against the corresponding JSON schema. An express.js middleware suggested for validating requests against JSON schemas is *express-json-validator-middleware* (https://www.npmjs.com/package/express-json-validator-middleware), based on the *ajv* module (https://www.npmjs.com/ package/ajv).

Components' URI

In the components folder, there are the implementations of the classes present in the platform. Each one has the *self* attribute, as required by the HATEOS requirements, containing the URI to the resource.

You are invited to replace the current placeholder */change/me* with the correct URI for each resource. Beware to return the correct URI depending on the context in which the class is used.

Controllers Implementation

The generated server stub provides a list of controllers that you need to implement in order to make the server operational. While the controller is aware of the communication protocol used by the server (in this case, REST), the service layer is not. For this reason, the service functions return error messages as plain strings.

Your task is to connect the service layer with the REST controllers, ensuring that appropriate responses are sent to the client (returning the expected data on success and clear error messages on failure), that the correct HTTP status codes are returned, and that additional validation is performed when necessary (e.g., consistency between body fields and route parameters).

The functions available in the service/ folder can be used to perform various operations on the platform. Feel free to modify them if your design requires different logic, or to implement new services as needed.

User authentication

The user authentication with Passport and express-session is managed server side in the following way:

1. when a user tries to authenticate to the platform, a Passport Local Strategy is used to check if the user email exists, and the specified password is correct;
2. if the authentication is successful, the platform creates an authenticated session with the *express-session* module. In the creation of this session, the following value for the three required parameters are specified:
    a. the *secret* parameter set to a (possibly random) string that is used for signing the session id cookie (named *connect-sid*);
    b. the *resave* parameter set to false;
    c. the *saveUninitialized* parameter set to false;
3. the server includes the *connect-sid* cookie in the HTTP Response and sends it back to the client (this operation is automatically managed by Passport and it is transparent to the user);
4. when a user requests an operation that requires authentication, an authentication middleware verifies if the requesting user has been previously authenticated. In this middleware, *req.isAuthenticated()* is used to verify user authentication.

Within the provided material, most of these aspects are already configured. Your task is to understand this part of the code and use the authentication middleware as needed.

Database management

You are free to create your own database for your personal solution, using *DB Browser for SQLite*. However, we provide a database, already populated with some records, which you can use or extend for your implementation of the server. The credentials are stored in the /database/passoword_dabasbases.txt file of the laboratory repository. If you want to investigate the database, you can use the VS code extension SQLite Viewer and open the *.db* file with the IDE.

For the hash computation, the bcrypt module of node.js is used. If you want to create a new user, you can use the following website to generate the hash of a password, according to bcrypt: https://www.browserling.com/tools/bcrypt. If the generated hash starts with $2y$, replace that part with $2b$, as the node bcrypt module does not support $2y$ hashes.