Distributed Systems Programming

A.Y. 2024/25

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* service, where users can keep track of the films they have watched and the reviews that they have written for them. If you have attended the *Web Applications I* course delivered by Politecnico di Torino in the A.Y. 2023/2024, you may be already familiar with some of the main concepts behind the *Film Manager*, and you are invited to reuse / extend what you did for the Labs of that course for carrying out this activity. Otherwise, you can have a look at the documentation of the Web Applications I labs (https://github.com/polito-webapp1/lab-2024) and to use, as starting point for your activity, the solution of the last lab activity of the Web Applications I course (https://github.com/polito-webapp1/lab-2024/tree/main/lab11-authentication).

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://editor.swagger.io/) 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 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, 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* service (mandatory);
- *name*: username of the user;
- *email*: email address of the user, which must be used for the authentication to the service (mandatory, it must be a valid email address);
- *password*: the user's password, which must be used for the authentication to the service (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* service (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* service. 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 (the online version or the locally installed version), you can automatically generate a server stub, corresponding to the design, to be filled with the implementation of the requested functionalities. The implementation of some of the necessary functionalities is already available in the solution of the last lab activity of the Web Applications I course. You are invited to reuse them in your implementation, while you will have to implement the other functionalities yourself.
In greater detail, the service must be designed and implemented according to the following specifications.

The *Film Manager* service 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 service, 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 service 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. 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 service 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 service, which checks if these credentials are correct. Emails used for authentication must be unique in the service. The authentication mechanism that is used by the service is a session-based authentication, where the session data are saved server-side, and the client receives a session identified 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). In order for this authentication mechanism and the 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, in order to facilitate debugging.

Differently from the lab activities of the "Web Applications I" course, here an explicit API for logout must not be designed. Instead, logout must be managed in the following way:

- When a user logs in, the authenticated session must last at most 5 minutes. After that time, the user is automatically logged out.
- When a login request comes for a new user, the Film Manager service checks if the request has been sent by an already authenticated user. In that case, the previously authenticated user is logged out, and then the new user logs in. Otherwise, the new user directly logs in.

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 service assigns it a unique identifier. The creator of the film becomes its owner. If the film is marked as public, a review invitation is not automatically assigned to its creator or any other user. 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* service is issuing invitations to review public films to the users who have access to the service. 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 the review invitation has been issued to can mark the review for that film as completed. *Hint*: you might need to create a new table in the database, to represent this kind of relationship between films and users, and define some foreign keys so as to link this table with the tables storing the records related to users and films.

Finally, the service must offer an additional operation to automatically issue review invitations for films for which no invitation has been issued yet, in such a way that the invitations to the users are balanced. 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, in order to limit the size of the messages the service sends back.
- When the users send a JSON *film* element to the service (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. *Hint*: 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).
- The service 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* service 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

How to make the server stub run

After you automatically generate the server stub using the Swagger Editor, you need to perform some *routing* operations. 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.

Unfortunately, routing is not automatically managed in the code generation mechanism provided by Swagger Editor. As such, before testing your stub server, you need to introduce the required route methods.

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

These two operations are depicted in the following screenshot, where they appear in lines 18-19. Comparing this screenshot with your "index.js" file, you can note that these two lines are the only ones that have been modified.

After installing the required node.js modules, you can test the server stub with Postman. At this point, you can proceed to populate the stub with the code that is required to provide the functionalities previously described in this document.

```
'use strict';

var path = require('path');
var http = require('http');
var oas3Tools = require('oas3-tools');
var serverPort = 3001;

/*** Swagger configuration ***/

var options = {
    routing: {
        controllers: path.join(__dirname, './controllers')
    },
};

var expressAppConfig = oas3Tools.expressAppConfig(path.join(__dirname, 'api/openapi.yaml'), options);
var app = expressAppConfig.getApp();

var filmController = require(path.join(__dirname, 'controllers/FilmsController'));
app.get('/api/films/public', filmController.getPublicFilms);

// Initialize the Swagger middleware

http.createServer(app).listen(serverPort, function() {
    console.log('Your server is listening on port %d (http://localhost:%d)', serverPort, serverPort);
    console.log('Swagger-ui is available on http://localhost:%d/docs', serverPort);
});
```

How to manage an error occurring after the server stub generation

When the server stub is automatically generated with the Swagger Editor, the oas3-tools (https://www.npmjs.com/package/oas3-tools) is employed for the configuration of the Express application. Unfortunately, the latest release of this module has a bug, which makes the definition of the route methods invisible to the Express application. As this bug has not been fixed yet, we propose an easy workaround: to use a previous version (the 2.0.2) of oas3-tools.

In order to do so, you simply have to modify the dependency related to oas3-tools in the package.json file in the following way:

```
"dependencies": {
  "connect": "^3.2.0",
  "js-yaml": "^3.3.0",
  "oas3-tools": "2.0.2"
}
```

Be aware not to write “^2.0.2”, otherwise the dependency is solved with the most recent version.

How to manage user authentication with Passport and express-session

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

1. when a user tries to authenticate to the service, a Passport Local Strategy must be used to check if the user email exists, and the specified password is correct;
2. if the authentication is successful, the service creates an authenticated session with the *express-session* module. In the creation of this session, you should specify the following value for the three required parameters:
    a. you should set the *secret* parameter to a (possibly random) string that is used for signing the session id cookie (named *connect-sid*);
    b. you should set the *resave* parameter to false;
    c. you should set the *saveUninitialized* parameter 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 should verify if the requesting user has been previously authenticated. In this middleware, you can use the *req.isAuthenticated()* to verify user authentication.

How to manage the hash computation

For the hash computation, you can use the bcrypt module of node.js. 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.

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. If you use this database,

with the pre-installed data you can use the following credentials to authenticate to the *Film Manager* service:

- email address: “user.dsp@polito.it”;
- password: “password”.