Distributed Systems Programming

A.Y. 2021/22

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 *ToDo Manager* service, where users can keep track of the tasks they must do in the future. If you have attended the *Web Applications I* course delivered by Politecnico di Torino in the A.Y. 2020/2021, you may be already familiar with the main concepts behind the *ToDo Manager*, and you are invited to reuse what you did for the Labs of that course for carrying out this activity. Otherwise, you are invited to look at the documentation of the Web Applications I labs (https://github.com/polito-WA1-AW1-2021) and to use, as starting point for your activity, the solution of the lab activity called "BigLab 2" (https://github.com/polito-WA1-AW1-2021/BigLab2-solution).

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 two core data structures of the *ToDo Manager*, i.e., the *users* who want to manage their task lists by means of this application, and the *tasks* they must carry out. 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 in the *ToDo 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 *task* data structure is made of the following fields:

- *id*: unique identifier of the task in the *ToDo Manager* service (mandatory);
- *description*: textual description of the task (mandatory, maximum length: 160 characters);
- *important*: this Boolean property is set to true if the task is marked as important, false otherwise (default value: false);
- *private*: this Boolean property is set to true if the task is marked as private, false if the task is public (default value: true). A task is said private if only the user who created it can get information about it, public if every user can get information about it;
- *projects*: the names of the projects in which the task is inserted. In the *ToDo Manager* service, only a predetermined set of possible values for *projects* must be accepted (e.g., you can suppose that "Personal", "WA1_Project", "WA2_Project", "DSP_Project" are the only acceptable values for the *projects* field);
- *deadline*: the due date and hour of the task. The *ToDo Manager* service must accept only dates following January 1st, 2020. The date format must be compliant to the ISO 8601 standard;
- *completed*: this Boolean property is set to true if the task is marked as completed, false otherwise (default value: false);
- *owner:* the user who created the task. può essere l'id o l'intero oggetto?
- *assignedTo*: the users this task has been assigned to.

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 *ToDo 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 (e.g., you can suppose that each task is inserted into at most one project, for simplicity).
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 requested functionalities. The implementation of some of the necessary functionalities is already available in the solution of the WA1 Lab "BigLab 2". You are invited to reuse them in your implementation, while you will have to implement the other functionalities yourself.
In greater detail, the service has to be designed and implemented according to the following specifications.

The *ToDo Manager* service allows users to keep track information about the tasks they should carry out. A user who creates a task can assign it to other users, who should then complete it. Two key terms are task *owner* and task *assignee*. The former is the user who creates the task in the service, the latter is a user who is in charge of carrying out the task. The service maintains information about the users who are enabled to use it and their tasks 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 hash of the

password is computed and stored in the database. **Hint:** For the computation of hash, 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.

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

- the authentication operation itself (login);
- the operation to retrieve the list of all the tasks that are marked as public.

For authentication, the user sends the email and password to the service and, if these credentials are correct, the service returns a JSON Web Token (JWT) containing the user id, setting it as a cookie (the token should also contain an expiration date, but for simplicity this feature is omitted in this Lab).
The JWT has then to be sent as a cookie in future requests that target resources available only with authentication. If the client is a browser, or another application that manages cookies automatically (e.g. Postman), it sends cookies automatically and transparently. In the other cases, the client application has to send the JWT as a cookie in each request.
The management of the JWT as a cookie can be implemented by using the Passport middleware (http://www.passportjs.org/), and the *passport-jwt* module (https://www.npmjs.com/package/passport-jwt), as explained in the Guidelines below.
In order for JWT tokens to be secure enough, it is 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.

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

- The user can create a new task. If the creation of the task is successful, the service assigns it a unique identifier. Note that this operation is separated from the task assignment. The creator of the task becomes its owner. However, a task won't be necessarily assigned to its creator. The assignment of a task to the users is explained later in this document.
- The user can retrieve a single existing task, identified by the specified *id*, if at least one of the following conditions is satisfied:
    1) the task is public;
    2) the user is the owner of the task;
    3) the user is an assignee of the task.

The user can also retrieve the list of all the tasks that she created, and the list of all the tasks that are assigned to herself.

- The user can update an existing task, identified by the specified *id*, if she is the owner of the task. For example, this operation can be used to change its visibility from public to private (and vice versa). However, this operation does not allow changing the status of the *completed* property, which is subject to different rules.
- The user can delete an existing task, identified by the specified *id*, if she is the owner of the task.

A central feature of the *ToDo Manager* service is the assignment of the tasks stored in the database to the users who have access to the service. The main operations related to this feature are the following:

assegnatario : person entrusted with

- The owner of a task can assign it to a user (the assignee may be the owner herself).
- The owner of a task can remove its assignment from a user.
- The owner of a task can retrieve the list of all the assignees of that task.
- The assignee of a task can mark the task as completed.

It is suggested to organize the design and implementation of this feature in two subsequent steps. However, you can keep only the solution of the second step, as you will extend that in the next lab activities.

1. First, design and implement the service in such a way that a task can be assigned to one and only one user. Only this user has the possibility to mark this task as completed.
2. Then, modify the service design and implementation in such a way that a task can be assigned to multiple users at the same time. Each user the task has been assigned to can mark the task as completed. *Hint*: you might need to create a new table in the database, to represent this kind of relationship between tasks and users, and define some foreign keys so as to link this table with the tables storing the records related to users and tasks.

Finally, the service must offer an additional operation to assign automatically the non-assigned tasks to the users, in such a way that the assignments 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 tasks 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 *task* element to the service (e.g., for the creation of the task in the database, or for the update of an existing task), 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 and features should be self-describing. Therefore, when the *ToDo Manager* service sends back a JSON *task* or *user* object, this should include a *self* link referring to the URI where the resource can be retrieved with a GET operation. Moreover, a client should be able to perform all operations without having to build URLs.

Guidelines for the solution development

How to make the server stub run

After you automatically generate the server stub using 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.

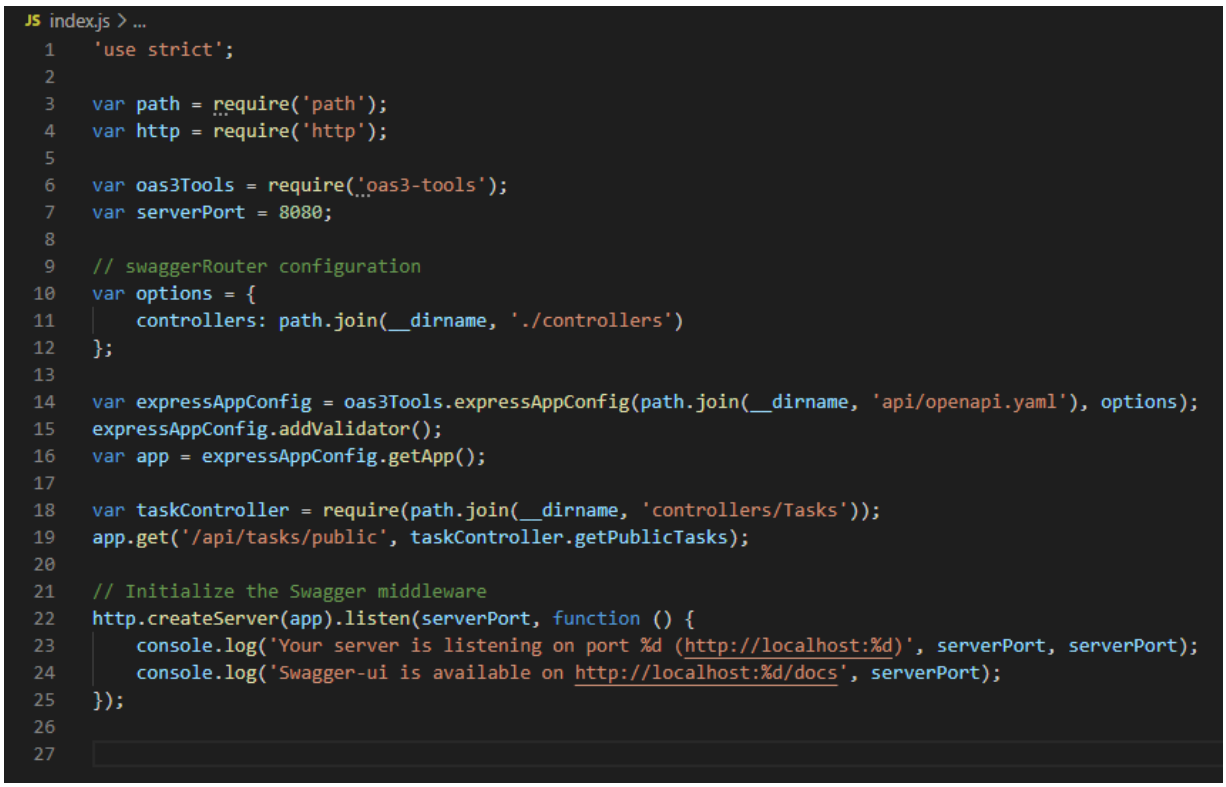
Unfortunately, routing is not automatically managed in the code generation mechanism provided by Swagger Editor. As such, before testing your stub server, you need to introduce the required route methods.

Let us suppose that you must map a GET method to the "/api/tasks/public" path; additionally, in the corresponding route method, the callback function that must be invoked is *getPublicTasks*, located in the "controllers/Tasks.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/Tasks.js” using the *require* function, and thus getting an object (*taskController*) which gives access to the exported functions of “controllers/Tasks.js”;
2) creating the routing method for the GET operation and mapping the path “/api/tasks/public” to the callback “taskController.getPublicTask”.

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.

How to manage an error occurring after the server stub generation

When the server stub is automatically generated with 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 JWT and Passport

The user authentication with JWT and Passport should be organized 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 should create a JWT containing the user id, and include it in the Set-Cookie header. To do so, you can use the following piece of code, where *user.id* is the id of the user who is being authenticated, while *jwtSecret* is a string representing an internal secret of the service;
```
const token = jsonwebtoken.sign({ user: user.id }, jwtSecret);
res.cookie('jwt', token, { httpOnly: true, sameSite: true});
return res.json({ id: user.id, name: user.name });
```
3. when a user requests an operation that requires authentication, an authentication middleware should check the JWT in the cookies of the request. To do so, first you should define a JwtStrategy (see below the corresponding piece of code). Second,

you must use the authentication middle in the route methods that require it. The middleware is specified as: *passport.authenticate('jwt', { session: false })*. If you have used the pieces of code shown here, and if the verification is successful, then the field *req.user* of the request will contain the id of the user taken from the JWT. You can use this information for internal operations (e.g., if the user is trying to update a task, you check if she is the real owner).

```
var cookieExtractor = function(req) {
    var token = null;
    if (req && req.cookies)
    {
        token = req.cookies['jwt'];
    }
    return token;
  };

var JwtStrategy = require('passport-jwt').Strategy;
var opts = {}
opts.jwtFromRequest = cookieExtractor;
opts.secretOrKey = '6xvL4xkAAbG49hcXf5GIYSvkDICiUAR6EdR5dLdwW7hMzUjjMUe9t6M5kSAYxsvX';
passport.use(new JwtStrategy(opts, function(jwt_payload, done) {
    return done(null, jwt_payload.user);
   })
);
```

Database management

You are free to create your own database for your personal solution, using *DB Browser for SQLite*. However, we provide two databases, already populated with some records, which you can use or extend for your implementation of the server:

1) "databaseV1.db" contains only the *users* and *tasks* tables, and it has been designed to be used for the *ToDo Manager* version where a single user can be assigned to each task;
2) "databaseV2.db" additionally contains the *assignments* table, and it has been designed to be used for the *ToDo Manager* version where multiple users can be assigned to each task.

If you use these databases, with the pre-installed data, you can use the following credentials to authenticate to the *ToDo Manager* service:

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