TD 2 – Implementing a Rest Api in NodeJs

Introduction

Some students need to learn by heart many (thousands of) lessons. This is especially true for students in law, medicine, biology, languages, etc. They usually use tools like “Anki” to optimize their learning curves.

Anki is base on neuro-science experiments, proving that the best way to learn things is to repeat over time, with longer and longer spacing. Your brain starts saving information in the temporary memory, and then progressively fix it in the long-term memory over time.

Anki offer an interface not very user-friendly, but enough. The web version is not free, but the desktop one is, and you are invited to test it.

There are of course many other alternatives to Anki, for example some for children, with a simpler user-experience on mobile phones, using emotion, animations, blinking rewards, and games points to keep the child focused.

Objective Overview of the Hands-On

Our goal is to implement something similar to basic Anki features in web (NodeJs + Angular). Anki is highly extensible, we will focus on the main functionalities. The TD2 is focusing only on the Backend side: NodeJs + Express + Routes + Database, and only for 1 table.

Futures TDs later will bring you the Web FrontEnd in Angular, to connect to this backend over Http.

[Pre-requisites already from TD1]: Check installed

Ensure you have NodeJs + WebStorm correctly installed on your PC

[already from TD1] Step 1: Create/Import npm Project In WebStorm IDE

Create a new project (or reuse from TD1) using

C:\td2> npm init

Add npm dependencies to your generated package.json file for “express”, and “typescript”

From WebStorm IDE, open your Typescript project (open file package.json “as a Project”, not “as a File”).

Check you have a “tsconfig.json” file. If you do not have tsconfig.json file, then create a default one using WebStorm menu File > New > tsconfig.json

Step 2: main code for express server

Easy part. Ask ChatGPT to create a typescript skeleton Rest application code in nodejs for express, with a route GET “/api/liveness” that simply return “OK” message with a status code 200.

Just copy&paste from ChatGPT, save in file “app.js” (if using JavaScript or “app.ts if using TypeScript).

You may also copy&paste from the lesson, from Google, from Express documentation Getting-Started, etc.

Start your nodejs application from the command line, like

C:\td2> node app.js

Test that your you are able to view the “OK” message in your Web Browser, when opening it from

http://localhost:3000/api/liveness

Step 3: launching or Debugging your server app

Now stop (Ctrl+C) your server from the terminal, and restart it from your WebStorm IDE. Notice you may have to select “app.js” file (JavaScript generated) by expanding “app.ts” file (TypeScript).

Check that it still run “OK” in your browser when you refresh the browser page (F5).

Try to put some breakpoint in your debugguer, and check that you are able to execute code line by line, and inspect variables values.

Every time you press F5 in your browser, your breakpoint corresponding to the method handler of “GET /api/liveness” should be called. Do not forget to “Resume” the program from the debugger, not to freeze it. Maybe remove the breakpoint for later.

Step 4: Test using curl or Postman

Also test you server using curl (command line utility, mostly for Linux / Cmder developpers), or using Postman.

Install Postman app, and execute the same query “http GET /api/liveness”.

The advantage of Postman compared to the web browser, is that you can also test POST,PUT,DELETE calls, and save your requests along with request headers and request body, to be re-used later.

Step 5: Model the classes by drawing an UML classes schema

Here is a textual description of the model classes your have to draw in UML.

To start with, all items to be learned will be called “LearningFact”.

They are grouped in a “LearningPackage”. A package is basically a title, a description, a category, a targetAudience (text for pre-requisite, like age of the student), and a difficulty level (from 1 to 20).

As a user, you may choose packages to work on, let call this association “UserPackageLearning”. A UserPackageLearning has a startDate, an expectedEndDate, and minutesPerDayObjective (a number of minutes the user is willing to spend on it).

Each learning of a “LearningFact” for a user can be called “UserLearningFact”. It is caracterised by the number of times the user has already reviewed it, the confidence level the user thinks he remember it, and the last reviewed date.

From this point, you will have to implements all the API calls to manage such a database. You are guided for some of the api calls during the Hadns-On. You will have to finish yourself more calls at home. Notice that having all administrative operation available as apis is not mandatory, you may edit directly some values in a pgAdmin console.

Step 6: Declare your model entities as Typescript interfaces

For now, a learning package does not need to have fields for relationships (no LearningFacts list). Foreign keys may be declared as number type, and suffixed as “Id”.
Focus on “LearningPackage” for next Steps. You may finish the other interfaces later at home.

Step 7: hard-code an array of LearningPackage objects

Currently, we don’t have a database. So let’s start by hard-coding in typescript and array of 4 LearningPackage objects, each one with resp. title=”Learn TypeScript”, ”learn NodeJs”, “Learn Html”, ”Learn Angular”. Thoses object will have resp. id=1,2,3,4.

Step 8: Declare a route GET “/api/package”

Expose an http route for querying all the “LearningPackage” (hard-coded array from previous exercise).

Step 9: Declare a route GET “/api/package/:id”

Expose an http route for querying the data of a given “LearningPackage”, given its “id” provided in the request path.

When you find the package for the given id, your app must respond statusCode 200 and response body is the object in json, otherwise your app must respond statusCode 404, “entity not found for id:…”

Step 10: Declare a route POST “/api/package”

Declare an express route for POST “/api/package” that will create a new “LearningPackage” from json request body, and assign it a new id, then add it to the list of all packages.

Do not forget to add “express.json()” as a middleware feature in express (which is the replacement of old “body-parser” feature).

Your server should respond a statusCode 200 with the created object in response body (along with it id..), or else a statusCode 400 if some mandatory fields are not provided.

Step 11: Declare a route PUT “/api/package”

Similar to POST… but for updating an already existing object, finding it from its id.

When it was found and successfully modified, your server should respond a statusCode 200 with the modified object in response body, or else a statusCode 404.

Step 12: Declare a route GET “/api/package-summaries”

This is similar to “GET /api/package”, but should not return the complete objects, only the {id,title} fields. If you had millions of objects, and objects contain large data (nested list, detailed infos…), this would make a big performance difference.

[Optional] Step 13: Declare a route GET “/api/package-summaries/search?title=..&description=..&tag=..”

This is similar to “GET /api/package”, but would filter on the server only packages that match a given query parameter for “title” or other filtering condition.

By matching, you can consider that the query parameter is case incensitive, and should be found in the object text field (this is not an exact string equality comparison).

[Optional] Step 14: Setup swagger OpenApi

As seen in course, add npm dependencies for Swagger JsDoc and Swagger UI.

Configure it in express.

Then add comments for all your api operations, to describe them.

The Swagger JsDoc is like “JavaDoc” : just plain old /** comment */ , but contining “@openapi”, and describing the Http operation.

The Swagger UI is a web page for exposing in Html a user-friendly interface for testing the operations (text field for inputing values, then button to berform operations)

Step 15 : Install a PostgresQL database on your PC

Install on you PC a PostgresQL database server, if you do not already have one (would be surprising).

At this point, you have an admin username/password for connecting as admin.

Open pgAdmin, connect as admin to your localhost server, and create a new Database, named “LearningFact”.

You may also create a new user account learningDbUser with a password, that will be the owner of the database.

When you NodeJs backend connect to you database, you have to provide all 4 information:

- Network name of the server (“localhost”)
- Name of your database “LearningFact”
- Username: “learningDbUser”
- Password: ***

At this point you do not need to create the SQL Tables, it will be done after.

Step 16 : add npm dependency for Postgresql and Sequelize

Add NPM dependency in your project for Sequelize (an ORM library, easing the use of SQL Tables directly from code), and also for the database driver compatible with your postgresql server.

Configure Sequelize in Typescript to connect to your {server, database, user/password}.

Step 17 : Configure Sequelize for Table LearningPackage

Now configure Sequelize a class “LearningPackageTable” for corresponding SQL table “LearningPackage”.

You have to describe all the columns name and type of your SQL Table.

When it is done, you can synchronize the Typescript declaration code (class for the Table) to the real SQL Table in your database.

Step 18 : Check in pgAdmin … enter some rows

Open pgAdmin. This is a user-interface for exploring the content of your postgres server – database – tables.

From Step 17, you should be able to find a newly created table class “LearningPackage”, and it should have columns.

This table does not have rows yet, so add and save 4 rows, corresponding to the hardcoded objects.

Step 19: Re-implement you express api operation using the database

Rewrite all the express api methods, so that they now use the Sequelize table “LearningPackageTable”.

[To Finish at Home] Step 20 : repeat CRUD operations for “LearningFact”

Very similar to method GET,POST,PUT, now implement methods for managing LearningFact for a given LearningPackage

Knowing that a “LearningFact” is inside a “LearningPackage”, the naming conventions for the request URL on LearningFact could be

GET /api/package/:id/fact

Get all LearningFacts for a given package

POST /api/package/:id/fact

Create and Add a new Fact to a given package

PUT /api/package/:id/fact

Update an existing Fact of a given package

DELETE /api/package/:id/fact

Delete an existing Fact of a given package … or simply mark it as “disable=true”, because you usually never delete in a database!

[To Finish at Home] Step 21 : Design more needed APIs … not only CRUD !

What are the operation, that are not CRUD, that you might need to implement your Front-Back application next ?

Every action button (not navigation link) in you application should correspond to an API operation… not a pure “Rest” operation, but simply a POST or PUT taking some parameters, and responding some result..

Suggested action:

- Start a learning session => so give me the next learning fact to study today
- Answer to one LearningFact => save result associated to user – fact … and compute next day for reviewing it
- End a learning session => give a summary of you progress, and your remaining tasks
- Etc.

[To Finish at Home] Step 22: finish you server !!

Finish implementing the operations, because you will need them later to really connect your Angular web application to your server.

[Bonus Step, at Home] Step 23

Study how much time it would take to enrich the app with the following new features:

A “Tag” is an object corresponding to a unique keyword (or small phrase) in English and a French translation.

Tag may be associated to many different packages, and packages may have several tags (no need to repeat). We can call this association “LeaningPackageTag” in a relational model.

Draw the UML schema for it.

Evaluate how difficult it is to modify the code to add this feature. Does it fit without too much changes or risk?

Estimate how many days/hours of work it would take to implements such feature, in the backend nodejs, and in the frontend angular.