PRACTICAL LESSON PLAN

Subject:	Web Applications Development	**Degree:**	Informatics Engineering
Topic:	Data brokering III	**Professor:**	Noélia Correia (ncorreia@ualg.pt)
Type:	⊠ Programming □ Setup	**Institution:**	Universiy of Algarve

Goals

An API proxy acts as intermediary between the consumer and backend services. It can be a small shim[1] of code, or more complex code that handles data transformations, security, and so on. It can expose an interface customized for the consumer, hiding complex calls to multiple backend services, and can build some cache. The goal of this lab is to use Express to create an API proxy server exposing a simplified API, which interacts with the main API [1]. A subset of data must be customized and returned to consumers. Also, middleware is used to optimize the API proxy server. You will be using TypeScript, Express and you will be reusing code from Lab3.

Required Resources

You need:

- ◇ Your laptop.
- ◇ Node and npm installed.

Tasks

Creating Modules

Modules avoid mixing together code (with functions of different kind) into one large file, or duplicating the code in different places. Here you should create the following modules:

- ◇ **externalAPI**: This will hold the code for retrieving and transforming data from the main API, {JSON} Placeholder as done in Lab3. Two async functions should be exported, `getAllUsers()` and `getSomeUsers(city: string)`. The first retrieves all users while the second retrieves just users from a given city, and both should use fetch API. Catch any errors that may occur. The function performing your transformation is also placed here (use the one from Lab3).
- ◇ **main**: The starting point of your API proxy. You should start by making all the necessary imports, and test if the calls to functions inside `externalAPI` are working (e.g., print any returned data to the console)

Set up Routes

Create module **routes.ts** holding a router and managing GET requests to endpoints `"/"` and `"/city/:nameCity"`. These should be mounted at the path `/users` in you `main.ts` file (after an Express app is created in there). When related routes and middleware are placed in a single file then a modular API is ensured.
At this point you can include `res.send("...")` (inside your routes) just to test (with your browser) if routes are working fine. Do not forget to create an Express app in `main` module.

Implement Actual Behaviour of Routes

Change the callback function at router endpoints so that the main API, {JSON} Placeholder, is questioned accordingly by your proxy API. That is, besides retrieving dada, transformation should be applied (the one implemented in Lab3) and a json output should be returned to the client. This should be done for all endpoints.

[1] Shim code is a small layer/library that transparently intercepts an API, changing any passed parameters or providing any compatibility between different interfaces or API.

Optimize an Express Server

The performance of your API can be improved with the introduction of compression and caching:

- ◇ **Compression**: Act of shrinking the payload sent to the consumer, reducing time transfer of data across the network. Gzip is supported by all modern browsers, so include the `compression` middleware in your Express app so that responses are compressed, as long as the client includes an `Accept-Encoding` header line. Check the performance improvement using the Web Development Tools provided by your browser.
- ◇ **Caching**: Act of storing previously requested data in order not to retrieve it from the source, reducing latency. Create an extra module for you to implement your own caching middleware, which should be passed into specific route handlers. Use `node-cache` API when developing such module. Check the performance improvement using the Web Development Tools provided by your browser.

Main References

[1] `https://jsonplaceholder.typicode.com/`