Express with Typescript (and MongoDB)

For most of what we will do in this period, I expect you to continue with TypeScript. I also expect you to use this template given below for everything involving TypeScript. It is has a Typescript-configuration file that allows you to use plain javascript (*.js) if you prefer.

Getting started

Clone or fork this project: https://github.com/fullStackJavaScript-dat/express-mongo-typescript-start-code.git, and **watch** these videos to familiarize you with what it offers (almost everything is in package.json). This is what you should use for the rest of these exercises (make sure to READ the README.md file)

Add a welcome page

Create a folder called public (or whatever you like) and in this folder place an index.html file with some info about the site.
Hint: https://expressjs.com/en/starter/static-files.html

Create a User-API for the Semester Case

In this part, you will build a simple API which could, but I'm not sure whether it will end like this, be used for the semester Case. You will build it following these steps:

- Create a simple facade with an in-memory data structure
- Build a REST API on top of the facade.
- Add error handling to the app.
- Add debugging and logging functionality to the app
- Add Authentication and authorization to the app.
- Add Tests to the app (unit as well as integration-test)
- Secure the app, using best practices provided by Express
- Add a real database to the app.
- Deploy the app on a droplet

This exercise is meant for 3-4 consecutive lessons, so skip highlighted parts for now

Facades for the User-API

a) In the source-folder, create a new folder, `facades`. In this folder create a file userFacade.ts and paste in the code below (userName below = email):

```
interface IGameUser { name: string, userName: string, password: string, role: string }

const users: Array<IGameUser> = [];
class UserFacade {
  static addUser(user: IGameUser): boolean {
    /*Info: Import bcryptjs and (npm install bcryptjs) and hash before you store */
    throw Error("Not Implemented")
  }
  static deleteUser(userName: string): boolean { throw Error("Not Implemented") }
  static getAllUsers(): Array<IGameUser> { throw new Error("Not yet implemented") }
  static getUser(userName: string): IGameUser { throw new Error("Not yet implemented") }
  static checkUser(userName: string, password: string): boolean {
    /*Use bcrypjs's compare method */
    throw new Error("Not yet implemented")
  }
}
```

b) Implement the five static methods

A REST-API that draws on the facade

Implement the following REST-API for GameUser manipulation
You decide for the in/out-JSON format
GET:
`/api/users/: id`
`/api/users  (changed from api/users/all)`

POST:

`/api/users`

DELETE:

`/api/users/:id`

Highlighted endpoints will eventually require authorization, but don't focus on this yet
Observe: there is NO *login endpoint*. Eventually, we will use HTTP basic auth, so we get most of this for free.

Using the debug package

Install the debug package (npm install debug)
Add the following declaration to your local .env-file: `DEBUG=game-case`
In all your future files for this project, DON't do `console.log` but import debug like this in the start of your file: `const debug = require("debug")("game-case")`

Then, whenever you have done `console.log("Hello")`, do `debug("Hello")`
Make sure you understand why this makes a BIG difference.

Before you Continue.

The task's given above were deliberately held simple, so you could follow the steps given by Mosh in his video.
With JavaScript however, we will probably NEVER operate on data in a synchronous way as you did above (possible only, since data was held in a simple array)

You should clone this project (remember to read README.md) for the rest of this exercise/semester and IMPORTANT, watch my video where I explain what I did and why.
🟥 A better strategy is to watch my video and change your own code according to what I explain.

Express and Middleware

It's essential to understand the concept of middleware when using Express. In the following, you will first write your own middleware, and after that use existing middlewares to do more detailed logging and add authentication and authorization to the app.

First middleware example - A Simple Application-logger for the app

In this exercise, you will create a simple logging middleware you will use for the remainder of this exercise . Obviously, for something as important as logging, existing packages already exist for Express, but before using that, we will first write our own in order to learn the concept middleware.

a) Create a simple middleware, which for ALL incoming API-request will log (console.log in this first version) the following details:

Time, The *method* (GET, PUT, POST or DELETE), the URL

b) Move your logger-middleware into the folder middlewares, in a file `simpleLogger.ts`, and export it from here. Import and use it in `app.ts`.

c) In the middlewares folder create a new file, my-cors.ts and add and export a middleware function that can add required CORS-headers. You should watch the full video, but at 4.48 in this video you should have seen before today's lesson, you can see how it can be done (not in a separate file, that's for you to do).

Using existing middleware

A more realistic logger middleware

a) Add a new file logger.ts to the middlewares folder. Add the required code using the Winston package: and the express-winston package so that all logging-info goes into a file `/logs/all.log` and errors go into a file: `/logs/error.log`

Hints:

For 🟨 🟥 students You can twist the example given at the express-winston link above to do exactly what you like.

For 🟩 students

A ready to use middleware for CORS

Leave your own my-cors.ts file for future reference, but change app.ts to use the cors-package.

Once you have installed this package this is as simple as importing it, and use it, that is two lines of code.

Info: Obviously when deployed, and for a real-life app, you should fine tune the cors-headers to you specific needs, but the two lines referred to above will do for now.

Error Handling

Provide the project with an error handling strategy, that will generate json-responses for errors related to the API, and just use the default error handler for other errors.

Basic Auth with middleware

Write your own middleware to handle basic authentication and authorization for the project.
Use this package to handle all details related to Basic HTTP Auth.
Unless you really like the challenge ;-) this is actually meant as a code-along exercise given in this video

Important: For a real life application you would not write your own security middleware, but use an existing (trusted) one. Passport would be a realistic candidate, but for this exercise, middleware, and writing middleware, was the main focus.

Add Tests to the app (unit as well as integration-test)

- Add a file with a test suite to test the User Facade
- Add a file with a test suite to test the User API

Add a real (MongoDB) database to the app.

- Refactor the User Facade to use a real MongoDB database and a **`users`** collection
- Refactor your test suite for the User Facade to use a test-database on your Atlas Account
- Refactor your API-test to use a test-database on your Atlas Account

Hints: Feel free to jump on this ongoing exercise using this repo:
https://github.com/fullStackJavaScript-dat/startcodeForP2Day4

To BE CONTINUED
Secure the app, using best practices provided by Express
Deploy the app on a droplet, using Express best practices.

logger.ts

```
//One way to implement the required logger functionality
import winston from "winston";
import * as expressWinston from "express-winston";
import path from "path"

let requestLoggerTransports:Array<any> =[
    new winston.transports.File({filename:path.join(process.cwd(),"logs","request.log")})
]
let errorLoggerTransports:Array<any> =[
    new winston.transports.File({filename:path.join(process.cwd(),"logs","error.log")})
]
if (process.env.NODE_ENV !== 'production') {
  requestLoggerTransports.push(new winston.transports.Console());
  errorLoggerTransports.push(new winston.transports.Console());
}
let requestLogger = expressWinston.logger({
    transports: requestLoggerTransports,
    format: winston.format.combine(
      winston.format.colorize(),winston.format.json()
    ),
    expressFormat:true,
    colorize: false
  })

let errorLogger = expressWinston.errorLogger({
    transports: errorLoggerTransports,
    format: winston.format.combine(
      winston.format.colorize(),winston.format.json()
    )
  })

 export {requestLogger,errorLogger};
```