UNIT 1

JAVASCRIPT

Final exercise – SanviPop

Client-side Web Development
2nd course – DAW
IES San Vicente 2020/2021
Author: Arturo Bernal Mayordomo

Index

Introduction

This final exercise will be an adaptation to TypeScript and also an expansion of the activities we've been doing in this unit (until week 4).

I'll give you the application skeleton, and you'll have to complete the necessary TypeScript files.

All needed dependencies are provided in the package.json file of the base project you need to download. Also, you have all the necessary HTML files, and the Webpack configuration to complete this project.

Dependencies for the optative CropperJS part: cropperjs, @types/cropperjs

Login page (login.html)

This page **login.html**, will have a login form. In this form a user will enter email and password. Also, geolocate the user, and if it's successful send the latitude and longitude of the user so it will be updated in the server.

The web service to call for login is:

POST → http://SERVER/auth/login

Request data example (lat and lng are optional):

```
{
    "email": "test@test.com",
    "password": "1234",
    "lat": 37,
    "lng": -0.5
}
```

Response example. It returns an authentication token you must save:

```
{
  "expiresIn": 31536000,
  "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NywiaWF0IjoxNjAzODIxNTcxLCJleHAiOjE2MzUzNTc1
NzF9.XK1RY-5C4UvhQlb7d8ack2718lKrx71va1ukURz7_NI"
}
```

Any error will result in a status different from 200, usually **401** (not authorized). Use **catch** to capture it. It will contain the following JSON:

```
{
  "status": 401,
  "error": "Email or password incorrect"
}
```

When the login is successful, save the token inside a LocalStorage variable (use the key '**token**' as the Http class is configured to get this value). You'll need to send this token in other web services, or they will return an authentication error. After the login is successful, redirect to the product's page.

If there's an error, show it to the user (You can show it in the p#errorInfo paragraph, or inside an alert message using sweetalert for example).

The first thing to do when loading this page: check if there's a token stored in your localStorage. If there's a token, call the service GET → http://SERVER/auth/validate, If it doesn't return an error (code 200, JSON: { ok : true}, redirect to the product's page directly. If it returns status **401** (error → catch), delete the token from the local storage (not valid).

Register page (register.html)

In this page, there will be a form to register a new user. In this form the following information must be sent:

- **name**
- **email** (and a field called **email2** to repeat the email information).
- **password**
- **lat, lng** (readonly fields)
- **photo** → User's photo sent in base64 format.

Geolocate the user and set the values for the lat and lng fields. If there's any error, use default values and show an error to the user.

When the user sends this form, check if both emails are equal or show an error and don't continue. If they're equal create an object with these properties and send it to the service: POST → http://SERVER/auth/register.

This service will return a status 200 (OK) containing the inserted user object or an error, usually status 400 (Bad Request) like this:

```
{
  "statusCode":400,
  "message": [
    "name should not be empty",
    "email should not be empty",
    "email must be an email",
    "password should not be empty",
    "password must be a string"
  ],
  "error":"Bad Request"
}
```

If you get any errors, show them to the user. If everything went ok, redirect to the login page. When there's an error response, you'll have to get the json content using the json() method (remembers it returns a Promise).

Products page (index.html)

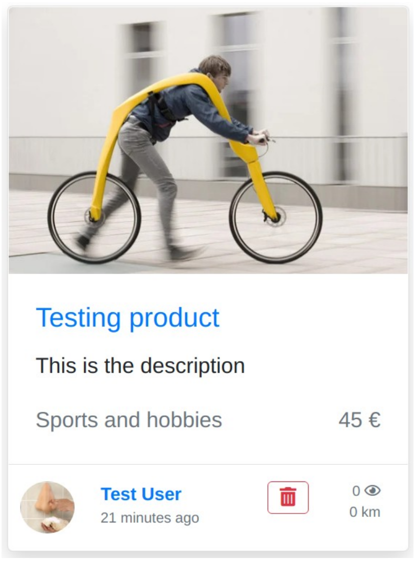
In this page, products will be shown, just like in previous exercises. Call the service GET → http://SERVER/products to get them all. These are the main differences:

When you load this page, check if the user is logged (see instructions in the login page section). If it's not logged (you get an error), redirect to the login.html page (do this in every other page except login and register).

Every product will contain the field **mine : boolean**, that will indicate if the product was created by you or not. It will also contain new information like the user that created the product (**owner**), the number of visits to the product (numVisits), or the distance from the product's owner to your location in kilometers.This is an example of a product the server will return to you:

```
{
  "products": [
    {
      "id": 169,
      "datePublished": "2020-10-27T10:31:00.000Z",
      "title": "la pulgaaa!!",
      "description": "Fichaje de ultima hora",
      "status": 1,
      "price": 99999999999,
      "owner": {
        "id": 1,
        "registrationDate": "2016-12-31T11:18:14.000Z",
        "name": "Prueba",
        "email": "prueba@correo.es",
        "lat": 38.410218199999996,
        "lng": -0.5163405,
        "photo": "http://localhost:3000/img/users/1603707241296.jpg"
      },
      "numVisits": 0,
      "category": {
        "id": 3,
        "name": "Sports and hobbies"
      },
      "mainPhoto": "http://localhost:3000/img/products/1603794660223.jpg",
      "soldTo": null,
      "rating": null,
      "photos": null,
      "bookmarked": false,
      "distance": 156.604,
      "mine": false
    },

  ]
}
```


Testing product
This is the description
Sports and hobbies 45 €
Test User
21 minutes ago
0
0 km

Only the products that are yours (mine: true) will show a button to delete them. Also, show user, visits and distance information in the card. You can find the updated handlebars template in the **templates/** directory.

Some properties (like status, soldTo, boormarked) won't be used in this project.

Add product page (add-product.html)

This page will contain a form to create a new product. It will be similar to what we've done in previous exercises.

This is an example of what you should send to the service: POST → http://SERVER/products:

```json
{
  "title": "Test product new",
  "description": "Product with\n2 lines",
  "category": 1,
  "price": 23.35,
  "mainPhoto": "IMAGE BASE64"
}
```

There can't be any empty field!

If everything goes well, the server will return the created product (status 201 → CREATED), like this:

```json
{
  "product": {
    "status": 1,
    "photos": [
      {
        "url": "http://localhost:3000/img/products/1603827095072.jpg",
        "id": 193
      }
    ],
    "title": "Test product new",
    "description": "Product with\n2 lines",
    "price": 23.35,
    "owner": {
      "id": 7,
      "registrationDate": "2020-10-25T14:28:14.000Z",
      "name": "Test User",
      "email": "test@test.com",
      "lat": 38.410218199999996,
      "lng": -0.1963405,
      "photo": "http://localhost:3000/img/users/1603823706448.jpg"
    },
    "category": {
      "id": 1
    },
    "id": 178,
    "mainPhoto": "http://localhost:3000/img/products/1603827095072.jpg",
    "mine": true
  }
}
```

When everything goes right, redirect to the products page . Or if anything goes wrong, show an error in the HTML (or an alert with the sweetalert2 library).

Product details page (product-details.html)

When we click a product’s image or name it’ll go to the details page.

In this page we’ll show the details of a product. We receive the product’s id in the url like this: **product-details.html?id=3**. Use **location.search** property to access the (id=3) part and extract the id from there. If there’s no id, redirect to **index.html**.

Call the service GET → http://SERVER/products/id service (**id** will be the number), to get the product’s info. It should return a product object (same as in the index.html page, but only one) → **{ product: {....} }.**

If the product doesn’t exist, it will return a 404 error (Not found).

Show the product’s information in that page (create the card). Use the toHTML method and put it inside **div#productContainer** element.

Load a map in the div#map element using Mapbox and show the position of the product (owner’s lat and lng properties).

User profile page (profile.html)

When we click on “My profile” link, or on a user’s avatar image, we’ll go here.

This page will show some user’s information. It can receive a user id in the url (profile.html?id=3). If it doesn’t receive any id,it will show the logged user’s profile. Call: GET → http://SERVER/users/me. Or if it receives an id, call instead: GET → http://SERVER/users/:id.

Both of these services will return a JSON object like this (or ok = false and error = “string” if anything goes wrong):

```
{
  "user": {
    "id": 7,
    "registrationDate": "2020-10-25T14:28:14.000Z",
    "name": "Test User",
    "email": "test@test.com",
    "lat": 38.410218199999996,
    "lng": -0.1963405,
    "photo": "http://localhost:3000/img/users/1603823706448.jpg",
    "me": true
  }
}
```

The “me” property indicates if that profile is your profile or another user’s profile. It will show the **Edit profile** button when it’s your profile. There’s a template to generate the profile HTML (profile.handlebars). Create a <div class="row mt-4"> element, put the handlebars result inside, and append it to the <div id="profile"> element.

Also create a map showing the user’s position in the div#map element.

Edit profile page (edit-profile.html) → Optative

In this page, we'll have 3 different forms to edit our user profile (call http://SERVER/users/me to get the logged user's info).

The first form will edit our name and email. Call this service by PUT → http://SERVER/users/me. And send this info to the server:

```
{
    "email":"test@email.com",
    "name": "Person Test"
}
```

The second form will edit the password. It will ask for the password twice. Send both passwords to: PUT → http://SERVER/users/me/password

```
{
    "password": "1234",
}
```

The third form will allow to edit your avatar. Send the new avatar in Base64 to: PUT → http://SERVER/users/me/photo

```
{
    "photo": "Photo in base64"
}
```

The name, email, and password forms will return just a confirmation from the server (Status 204, no content). The photo form will also return an object with the photo image url → {photo: “url”} (for more information see the video and the Postman services collection).

You can show errors and confirmation messages in the HTML o using SweetAlert.

General aspects

- All services except login, register and token checking require a header in the AJAX request called “**Authorization**” with the value “**Bearer token_value**”. The easiest way to do this is to check if the token exists in the local storage and send that header if it exists (Http class → ajax method).
    - Services that need the token will return an Unathorized HTTP code error (401) if there’s no token sent or if it’s not a valid token.
- As this exercise should be done in TypeScript. All variables, parameters must include their **type** when it’s not clear in the assignmet (and also the **return type** for functions and methods).
    - For example, if a method returns a Promise with a Product inside, the return type will be Promise<Product>. If it doesn’t return anything, it should be **void**. All async methods return a Promise!, even if it’s Promise<void>.
- Use classes to encapsulate functionality. For example, the **Product** class will have the necessary methods to call web services that include /**products**/. A class named **User** will access the /**users**/ services, and a class called **Authentication** will access the /**auth**/ services (and the token).
- The **logout** button will just remove the token from the Local Storage and redirect to the login page. You will want to include this functionality inside the Authentication class for example. Every page that has the logout link will need to handle the click event for it.

Classes

These are the recommended classes and methods you could implement. Of course, **you can add more functionality,** or do things in a different way (you don’t have to use **async** for example):

```
export class Http {
   static ajax(method: string, url: string, headers:any = {}, body:any = null): Promise<any>

   static get(url: string)

   static post(url: string, data)

   static put(url: string, data)

   static delete(url: string)
}

export class Geolocation {
   static getLocation(): Promise<Coordinates> //pos.coords
}

export class Auth {
   static async login(userInfo: IUser): Promise<void>

   static async register(userInfo: IUser): Promise<void>

   static async checkToken(): Promise<void>

   static logout()
}

export class User implements IUser {
   constructor(userJSON: IUser)

   static async getProfile(id?: number): Promise<User>

   static async saveProfile(name: string, email: string): Promise<void>

   static async saveAvatar(avatar: string): Promise<string>

   static async savePassword(password: string): Promise<void>

    toHTML(): HTMLDivElement
}

export class Product implements IProduct {
   constructor(prodJSON: IProduct)

   static async getAll(): Promise<Product[]>

   static async get(id: number): Promise<Product>

   async post(): Promise<Product>

   async delete(): Promise<void>

   toHTML(): HTMLDivElement
}
```

I’m giving to you the interfaces already **implemented** (check the project).

Marks

The final mark will be calculated according to these criteria:

- Register and login page → 2,5 points
- Products page (index.html) → 2 points
- Add product page → 1 points
- Product details page → 2 point
- Profile page → 1,5 points
- Clean and well structured code in general → 1 point

Optative content (up to 2 extra points)

These 3 extensions will raise each the final mark of the exercise:

- **(0,75 extra points)** Implement Edit profile page.
- **(0,25 extra points)** Use SweetAlert for showing most of the error (and success) messages to the user.
- **(1 extra point)** Learn about the CropperJS library to crop images and use it for the products photo (add product) and user's photo (register and update).
    - Download the **cropperjs** dependency and its **@types/cropperjs** TypeScript definitions. Include it in your code with **import Cropper from 'cropperjs'**.
    - The product's image will have an aspect ratio of 16/9 with a width of 1024px. The user photo image's aspect ratio is 1 and its width 200px.