

UNIT 1

JAVASCRIPT



Final exercise – AssBook

Client-side Web Development
2nd course – DAW
IES San Vicente 2024/2025
Author: Arturo Bernal Mayordomo

Index

Introduction.....	3
Login page (login.html).....	4
Register page (register.html).....	5
Events page (index.html).....	6
Pagination, order and search.....	7
Optional: Attend an event.....	7
Add event page (add-event.html).....	8
Event details page (event-detail.html).....	9
Optional: attend an event.....	9
User profile page (profile.html).....	11
General aspects.....	13
Classes.....	15
Marks.....	16
Optional content (maximum 2 extra points).....	17

Introduction

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

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

- The necessary interfaces to type objects and server responses are implemented in the **src/interfaces** folder.
- The auxiliary classes **Http**, **MyGeolocation** and **MapService** (MapService is useful to create and manage maps) are also implemented.
- All needed dependencies are provided in the **package.json** file.
- HTML Templates (<template>) for events and people attending an event. If you prefer, create Handlebars templates and use that library.
- Also, you have all the necessary **HTML files**, and the **Eslint** and **Prettier** configuration to complete this project. Don't edit any HTML unless is very justified (**example**: optional parts like implementing CropperJS or modals with Bootstrap)

Web services

Web services are deployed in <https://api.fullstackpro.es/svtickets>. You can run them in your local computer along with the database to avoid many people inserting and deleting data at the same time.

Web Services source code can be found in this Github repository: <https://github.com/arturober/svtickets-services>. These services are properly **documented** in the Github page.

In that repository you'll also find (inside the **SQL** directory) the file to create the database in your local MySQL/MariaDB server (XAMPP).

To run the services API in you local computer. Start XAMPP and the database and in the services directory:

- Import **SQL/svtickets.sql** file. Edit **src/mikro-orm.config.ts** file and set the correct user and password ('root' and '')
- Run **npm i** (first time) and **npm start**
- Use the url <http://localhost:3000> in your client and Postman as the base.

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 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 (in the login, we also update the user's lat and lng):

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

Response: It returns an authentication token you must store (localStorage):

```
{
  "accessToken":
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NywiaWF0IjoxNjAzODIxNTcxLCJleHAiOiJlE2MzUzNTc1NzF9.XK1RY-5C4UvhQlb7d8ack2718IKrx71va1ukURz7_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 this token in other web services, or they will return an authentication error. This will be done automatically by the HTTP class as long as you have stored the token first.

After the login is successful, redirect to the [Events 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](#) or [Bootstrap](#) for example).

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)
- **avatar** → 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 message 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"
}
```

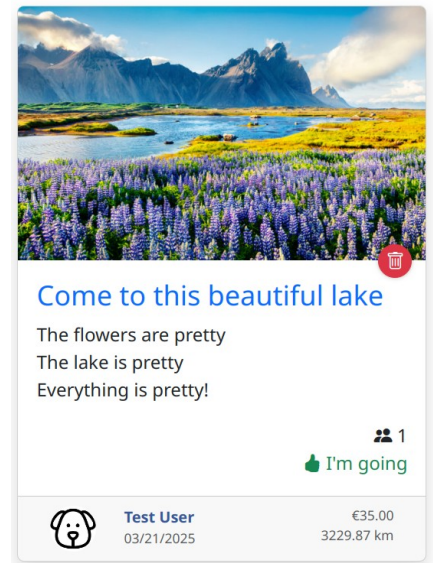
If you get any error messages, show all of them to the user. If everything went ok, redirect to the [login page](#).

Events page (index.html)

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

Every event will contain the field **mine : boolean**, that will indicate if the event was created by you or not. It will also contain new information like the user that created the event (**creator**), if you're attending the event or not (**attend**), or the distance from the event to your location in kilometers. Example:

```
{
  "id": 2144,
  "creator": {
    "id": 45,
    "name": "Another user",
    "email": "test3@email.com",
    "avatar": "http://SERVER/img/users/1633863254327.jpg",
    "lat": 38,
    "lng": -0.5
  },
  "title": "Event update",
  "description": "Description update",
  "date": "2021-12-03 00:00:00",
  "price": 25,
  "lat": 35.23434,
  "lng": -0.63453,
  "address": "Calle nada",
  "image": "http://SERVER/img/events/1633860306456.jpg",
  "numAttend": 0,
  "distance": 0,
  "attend": false,
  "mine": true
}
```



You can find the updated handlebars template in the **templates/** directory. You'll have to format the price, distance and date before calling the template.

Every event will contain the field **mine (boolean)**, that will indicate if the event was created by you or not. **Only** the event created by the logged user will have the "Delete" button. **Don't forget this!**

Now, the event has a creator, and the server returns that information inside the event (**creator**). Use that creator's avatar and name in the card.

When you click on the image or the event's title, it will redirect to the [Event Details page](#) with the id of the current event → Example: **event-detail.html?id=23**.

Pagination, order and search

Now, the search bar and the order buttons will make a call to the server to get the events filtered and ordered. The order buttons will send a param named **order** with one of these values ("distance", "price", "order").

The search bar has now a button to filter the events based on a text (by title only). Generate a param called **search** with the value and include it in the url to get the events.


Apart from the list of events, the server returns more properties in the response like **page** (current page) and **more** (if there's another page). When there are more events to load (more: **true**), the button#loadMore element will increase the current page number and include it in the URL. Loading more events must **append** them to the HTML container, and not replace.

To generate an URL containing these params, use an **URLSearchParams** object and concatenate it to the URL after the '?' character like this:

```
const params = new URLSearchParams({ page: String(page), order, search });
return await this.#http.get(`${SERVER}/events?${params.toString()}`);
```

Update the text below the nav bar, so the user knows the filters that are currently applied (default order="distance" and search=""):

Order: [Date](#) [Price](#) [Distance](#)



Ordering by: price. Searching by: party

Optional: Attend an event

At the card's right side , you will see the number of people attending to an event and text (with an icon) indicating if you're going to attend it or not. When clicking on this **div.attend-button** element, call the URL <http://SERVER/events/:idEvent/attend> using POST or DELETE, depending on the current **event.attend** value. If the call is successful change the icon class (bi-hand-thumbs-up-fill ↔ bi-hand-thumbs-down-fill) and the shown text (I'm going ↔ Not going), depending on the current **event.attend** value (change also this boolean). Update also the number of people attending (++ or --).

If you're not doing this optional part you can delete this HTML from the template.

Add event page (add-event.html)

This page will contain a form to create a new event. It will be similar to what we've done in previous exercises but adding these features.

- Show a map inside the **div#map** element. Geolocate the user and show a marker with the user's current position.
- Show an Autocomplete input in the **div#autocomplete** element so the user can write an address. When updating the position, show the marker and center the map in the new place. Also save the coordinates (lat, lng) in variables to send to the server.
 - To get the address value, you can do it like this:

`autocompleteDiv.querySelector("input").value`

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

```
{
  "title": "This is a new event",
  "description": "Description for the new event",
  "date": "2021-12-03",
  "price": 25.35,
  "address": "Nowhere",
  "lat": 35.23434,
  "lng": -0.63453,
  "image": "Image in BASE64"
}
```

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

Event details page (event-detail.html)

When we click an event's image or title, it should redirect here.

In this page we'll show the details of an event. We receive the event's id in the url like this: **event-detail.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/events/:id> service (:id will be the number → Example: <http://SERVER/events/4>), to get the event's info. It should return an event object (same as in the index.html page, but only one) → { **event**: {...} }.

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

Show the event's information on this page creating the same card as in the index.html page (the same template is present here). Put it inside **div#eventContainer** element. We don't want to repeat unnecessary code:

- Put the code to create the card and all the HTML inside in an external function. For example, you can create a method in the **EventService** class that returns the **div.col** element created. It could be something like **toHTML(event: MyEvent, template: HTMLTemplateElement): HTMLDivElement**.
 - The delete and attend buttons click handler won't go inside this method, but will be set outside. This is because actions after deleting or attending (optional) the event will be different in index.html (remove the card from the DOM), and here (redirect to index.html)

After deleting an event, redirect to the index.html page.

Load a map in the **div#map** element and show the position of the event with a marker (lat and lng properties). Show the address in the **div#address** element.

Optional: attend an event

If you implement the functionality to attend an event, you must show the list of users that are going to this event. There's a **template#attendTemplate** element that contains the HTML to show a user's information. Call the service <http://SERVER/events/:idEvent/attend> using GET and send the template an object with the list of users (like the server's response). Example:

```
{
  "users": [
    {
      "id": 48,
      "name": "Test User",
      "email": "test@test.com",
      "avatar": "http://arturober.com:5009/img/users/1634033447718.jpg",
      "lat": 38,
      "lng": -0.5,
      "me": true
    }
  ]
}
```

```
}  
],  
...  
}
```

Add these users to the list **ul#userList**. When clicking the button on the card that changes if you're attending or not the event (**div.attend-button**), reload the list of users below after calling the corresponding service (but do not reload the page in any case).

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:

```
{
  "user": {
    "id": 48,
    "name": "Test User",
    "email": "test@test.com",
    "avatar": "http://SERVER/img/users/1634033447718.jpg",
    "lat": 38.407608,
    "lng": -0.51634,
    "me": true
  }
}
```

Put the data inside the corresponding HTML elements (avatar → **#avatar**, name → **#name**, email → **#email > small**). The “me” property indicates if that profile is your profile or another user’s profile. When it’s not your profile, hide the corresponding buttons to edit the photo (label), profile and password (use class d-none or remove them from the DOM).

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

When it’s your profile, this is how the buttons work:

- **Edit avatar** → The <label> element next to the avatar is attached to the input type file (hidden). When the user clicks the label is like clicking the input to select a file.
 - When the user selects a file (change event in the input), transform it to base64 and send it to the server (PUT: <http://SERVER/users/me/photo>) in an object like this:

```
{
  "avatar": "Photo in base64"
}
```
 - If the server responds correctly, it will return a response with the same structure containing the avatar url. Set it in the profile image.
- **Edit profile** → When the user clicks on the **#editProfile** button, hide the **#profileInfo** element and show the **#profileForm** element instead to show the form which has 2 buttons (save and cancel).

- If the user presses cancel, hide **#profileForm** and show **#profileInfo**.
- When submitting the form, send an object to the server (PUT → <http://SERVER/users/me>) with the name and email to modify:


```
{
  "email": "test@email.com",
  "name": "Person Test"
}
```
- When the server responds without error, change the email and name in the HTML and hide the form, showing back the **#profileInfo** element.
- **Edit password** → Same as before, but showing the **#passwordForm** element instead.
 - Check that both passwords are equal (or show an error) before sending the new password to the server.
 - The service to call is PUT: <http://SERVER/users/me/password>, sending an object like this:


```
{
  "password": "1234",
}
```
 - When the server responds, hide the form and show back **#profileInfo**.

The profile and password forms will return just a confirmation code from the server (Status 204, no content).

General aspects

- When loading any page: check if there's a token stored in your localStorage. If there's a token, call GET → <http://SERVER/auth/validate>. Depending on the page you are, you will do this:
 - **login, register** → If it doesn't return an error (user is logged in), redirect to the [events page](#) (index.html) directly. If it throws an error, delete the token from the local storage (not valid).
 - **Other pages** → If it's not logged (you get an error), redirect to the login.html page
- The Http class methods are typed using generics <T>. This means that when calling a method, you should give it the response's type. In post and put methods, you must also indicate the type for the data sent to the server. There are interfaces for every response in the **src/interfaces/responses.ts** file. Example:
 - `this.#http.get<EventsResponse>(`${SERVER}/events`)`
 - `this.#http.post<SingleEventResponse, EventInsert>(`${SERVER}/events`, event)`
- Use classes to encapsulate functionality. For example, the **EventService** class will have the necessary methods to call web services that include **/events/**. A class named **UserService** will access the **/users/** services, and a class called **AuthService** will access the **/auth/** services.
- 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 AuthService class for example. Every page that has the logout link will need to handle the click event for it.
- As this exercise should be done in TypeScript. All variables and parameters, must include their **type** when it's not clear in the assignment (it's also recommended to specify the **return type** for functions and methods).
 - For example, if a method returns a Promise with an event inside, the return type will be `Promise<MyEvent>`. If it doesn't return anything, it should be **void**. All async methods return a Promise!, even if it's `Promise<void>`.
- By default, all HTML elements are returned from the DOM as generic `HTMLElement`. You'll have to cast them to the correct element if you want to use specific properties:

```
let image = document.getElementById("image") as HTMLImageElement;
img.src = ...; // OK
let value = (form.date as HTMLInputElement).value; // OK
```

- **Important:** The form to add a new event has a field named "title". The attribute **title** (string) is a property of any `HTMLElement`. If you try to cast it to `HTMLInputElement`, TypeScript won't let you do that using **form.title as HTMLInputElement**. Use the elements collection to access the input instead:
 - `(newRestForm.elements.getByName("title") as HTMLInputElement).value`

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 AuthService {
  ...
  async login(userLogin: UserLogin): Promise<void> {...}
  async register(userInfo: User): Promise<void> {...}
  async checkToken(): Promise<void> {...}
  logout(): void {...}
}

export class UserService {
  ...
  async getProfile(id?: number): Promise<User> {...}
  async saveProfile(name: string, email: string): Promise<void> {...}
  async saveAvatar(avatar: string): Promise<string> {...}
  async savePassword(password: string): Promise<void> {...}
}

export class EventService {
  ...
  async getEvents(page = 1, order = "distance", search = ""): Promise<EventsResponse> {...}
  async getEvent(id: number): Promise<MyEvent> {...}
  async post(event: MyEventInsert): Promise<MyEvent> {...}
  async delete(id: number): Promise<void> {...}
  async getAttendees(id: number): Promise<User[]> {...} // Optional
  async postAttend(id: number): Promise<void> {...} // Optional
  async deleteAttend(id: number): Promise<void> {...} // Optional
  toHTML(event: MyEvent, eventTemplate: HTMLTemplateElement): HTMLDivElement {...} // Create the Event
  card here (The delete event click outside)
}
```

I'm giving you the interfaces (MyEvent, User, etc..., and server responses) already **implemented** (check the project).

Marks

The final mark will be calculated according to these criteria (what is already implemented in the previous exercises does not count):

- Login page → 1 point
- Register page → 1 point
- Events page (index.html) → 2 points
- Add event page → 1 point
- Event details page → 1,5 points
- Profile page → 2,5 points
- Clean and structured code → 1 point

Even if everything works, there can be subtractions to the mark on each section if the code is not considered to be adequate. Format and organize well your code and avoid unnecessary repetitions.

Optional content (maximum 2 extra points)

These 3 extensions will raise each the final mark of the exercise, but only if the final mark without optional content is **7 or higher**:

- **(1 extra points)** Use SweetAlert or Bootstrap for showing error (and success) messages to the user (0,25), the confirmation to delete an event (0,25), and the forms to update profile info and password (0,25 each).
- **(1 extra point)** Implement the user's attending an event functionality in index and event-details pages.
- **(0,5 extra points)** Add a link in the user's profile to filter the events created by the current user. This link will redirect to the index.html page but including the id of this user like this: **index.html?creator=14**

- When the creator para is present in the url, show only the events created by these user, and include the name of the user in the text that shows the current filters applied. For example:

Creator: Peter.Ordering by: price. Searching by: am

- **(1 extra point)** Learn about the [CropperJS](#) library to crop images and use it for the event's photo (add event) and user's avatar (register and update profile).
 - Download the **cropperjs** dependency and its **@types/cropperjs** TypeScript definitions. Include it in your code with **import Cropper from 'cropperjs'**.
 - The event's image will have an aspect ratio of 16/9 with a width of 1024px. The user avatar image's aspect ratio is 1 and its width should be 200px.