UNIT 1

JAVASCRIPT

Final exercise – InmoSanvi

Client-side Web Development
2nd course – DAW
IES San Vicente 2025/2026
Arturo Bernal / Rosa Medina

Index

Introduction

IMPORTANT: It is crucial to read this entire document, especially the "General Requirements & Important Notes" section, before you begin working on the project.

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 auxiliary classes **Http**, **MyGeolocation** and **MapService** (MapService is useful to create and manage maps)
- All the necessary **HTML files**. Don't edit any HTML unless is very justified (**example**: optional parts like implementing CropperJS)
- New icons (place them in the icons folder)
- A Postman collection to test web **services** before implementing them

Web services

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

Clone the repository and **always** start the services before working on the project:

- Run **npm i** (first time) and **npm start**
- Use the url http://localhost:3000 in your client and Postman as the base.

Initial Setup & Configuration

Follow these steps to set up your project environment:

1. **Create Project:** Initialize a new **Vite** project using the **TypeScript** template.
2. **Assets:** Copy the images from the public folder of the previous assignment into your new project's public folder.
3. **VS Code Extension (Recommended):** Install the "Material Icon Theme" extension in VS Code for better visual differentiation between file types.
4. **Install Dependencies:**
   - Install ol, tailwindcss, and @tailwindcss/vite as regular dependencies.
   - Install prettier as a **development dependency** (use the -D flag).
5. **ESLint Configuration:** Replace the default eslint.config.ts file with the one provided. This configuration will be used for grading.
6. **Prettier Configuration:** Create a **.prettierrc** file in the root of your project with the following configuration. **You must format your code before submission.**

```
{
    "trailingComma": "es5",
    "tabWidth": 2,
    "semi": true,
    "singleQuote": false,
    "bracketSpacing": true,
    "arrowParens": "avoid"
}
```

7. **Ignoring Files (Optional):** If you dislike Prettier's default HTML formatting, create a **.prettierignore** file and add the rule: ***.html**.
8. **Project Files:** Copy the provided HTML and CSS files into your project. The index.html and new_property.html files are nearly identical to the previous assignment; changes will be detailed in their respective sections.
9. **Vite Configuration:** Create and configure vite.config.js to correctly handle all your HTML files. **Verify that running the build command (npm run build) generates all HTML files in the dist folder before submitting your project.**
10. **Code Migration:**
    1. Delete the sample files inside the src folder.
    2. Copy the JavaScript files from the previous assignment into src.
    3. Rename all .js files to have a .ts extension.
11. **TypeScript Refactoring:** Refactor the existing JavaScript code to be fully compliant with TypeScript. The Http class will be provided to you along with other necessary project files like MapService.

Required Interfaces

Create the following TypeScript interfaces to model your data structures. It is recommended to organize them into logical files (e.g., property.interfaces.ts, user.interfaces.ts).

- **Coordinates**: Represents geographic coordinates with latitude and longitude.
- **LoginData**: Represents the data sent during login (email and password).
- **RegisterData**: Represents the data sent during registration. It should **extend** LoginData and add any necessary fields.
- **User**: Represents the user data returned by the server for a profile. It should **extend RegisterData but omit the password field** (e.g., extends Omit<RegisterData, "password">) and add any other required fields from the server response.
- **Province**: Represents province data as returned by the server.
- **Town**: Represents town data as returned by the server. **Note:** The province property can be either a number (ID) or a full Province object. Use a union type (number | Province) to handle this.
- **PropertyInsert**: Represents the data for a new real estate property that is sent to the server (POST request).
- **Property**: Represents a real estate property as returned by the server. It should **extend PropertyInsert**, omitting fields that are only for insertion (e.g., townId via Omit<PropertyInsert, "townId">), and add any extra fields. If some fields only appear occasionally in the response, mark them as **optional** (using ?).
- **PropertiesResponse & SinglePropertyResponse**: Represent the full server responses for a list of properties and a single property, respectively.
- **SingleUserResponse**: Represents the server response when fetching a single user's data.
- **TokenResponse**: Represents the server response after a successful login, containing the authentication token.
- **AvatarUpdate & PasswordUpdate**: Interfaces for editing the user's avatar and password. Include an interface for the server's response as well.
- **(Optional) RatingInsert & Rating**: For the optional comments feature, create an interface for submitting a new rating (RatingInsert) and another for a complete comment object (Rating).
- **(Optional) SingleRatingResponse & RatingsResponse**: Interfaces for the server responses when fetching one or multiple ratings.

Feel free to create any other interfaces you deem necessary.

Login page (login.html)

This page **login.html**, will have a login form. In this form a user will enter email and password.

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",
}
```

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

```
{
  "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NywiaWF0IjoxNjAzODIxNTcxLCJleHAiOjE2MzUzNTc1NzF9.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 this token in other web services, or they will return an authentication error (401). 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 **index.html**.

If there's an error, show it to the user (You can use normal alert or (optional) a library like sweetalert, for example).

On page load, check if the user is already logged in (call AuthService.checkToken). If so, redirect them to index.html.

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**
- **password**
- **avatar** → User's photo sent in base64 format.

Password Validation:

- Add an input event to both password fields and check if their value matches. If not, set a custom validation error on the second field.
- Create a separate function for this check and call it from the input events of both password fields. The error should always be associated with the second field.
- **Remember to clear the error** (setCustomValidity('')) when they match, otherwise the form cannot be submitted.

Image Preview: Implement the same image preview functionality as in the "Add New Property" form.

Send the : 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.

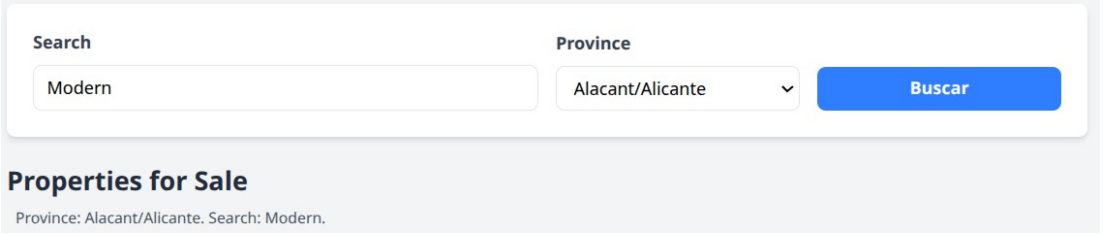
Properties page (index.html)

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

- **Data Display:** Property descriptions should **not** be displayed on the cards anymore; they will be shown on the detail page. The image and the title are inside **links** that should go to the details page. Don't forget to set the **href** attribute with the id (ex: property-detail.html?id=5)
- **Ownership Control:** The "Delete" button on a property card should **only be visible** if the property belongs to the logged-in user. The property object will include a **mine: boolean** field when you are authenticated.
- **Delete Confirmation:** Before deleting a property, use the confirm() dialog to ask the user for confirmation.
- **Pagination:** The server returns properties in pages of 12. Implement the "Load More" button to fetch the next page.
- **Filtering:**
  - Implement the text search bar and the province dropdown. Clicking the "Filter" button should update global state variables and fetch page 1 with the new filter parameters.
  - Clicking "Load More" should increment the current page number and fetch the next set of results, **appending** them to the existing list. The list should only be cleared when a new filter is applied (i.e., when fetching page 1).
  - Use URLSearchParams to build the request URL with search parameters (page, province, search, and optionally seller). Numeric parameters like page or province can be sent as "0" to indicate "no filter" if an empty string is not allowed.

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

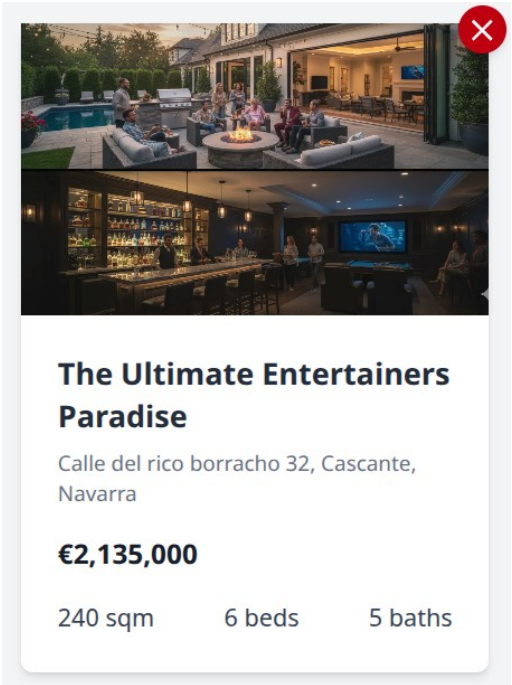
- **UI State:**
  - The "Load More" button should be **hidden** (by adding the hidden class) when the server response includes **more: false** in the response. It should be shown again when **more: true**.
  - Populate the province dropdown by fetching the provinces from the server.
  - Maintain the current search state (page, province, text) in global variables. Display a feedback message to the user below the search bar indicating the active filters. Example:



- **Authentication:**
  - On page load, check if the user is logged in using AuthService.checktoken().
  - If logged in, hide the element **.login-link** in the top menu and show the **.new-property-link**, **.profile-link**, and **.logout-link** elements.
  - The **.logout-link** button must have a click event that calls **AuthService.logout()** and redirects the user to the login page.
- **(Optional) Filter by Seller:** If the URL contains a seller query parameter (e.g., index.html?seller=5), include it in your API request to fetch only the properties belonging to that user. Update the filter feedback message to include the seller's name.

Example of a property object that the server returns when you're logged in:


```
{
    "id": 17,
    "address": "Calle del rico borracho 32",
    "title": "The Ultimate Entertainers Paradise",
    "description": "Host unforgettable gatherings in this home designed for entertaining. The open-concept living area flows effortlessly to a spacious outdoor patio with a built-in barbecue and fire pit.[1] A finished basement with a wet bar and home theater provides additional space for fun and recreation. With ample parking and a welcoming layout, this home is ready for your next celebration.",
    "sqmeters": 240,
    "numRooms": 6,
    "numBaths": 5,
    "price": 2135000,
    "totalRating": 5,
    "mainPhoto": "http://localhost:3000/img/properties/1760885974657.jpg",
    "createdAt": "2025-10-19T14:59:34.657Z",
    "status": "selling",
    "town": {
        "id": 1310689,
        "name": "Cascante",
        "longitude": -1.67903193,
        "latitude": 41.99909834,
        "province": {
            "id": 31,
            "name": "Navarra"
        }
    },
    "seller": 1,
    "mine": true
}
```

New property page (new-property.html)

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

- This page is for authenticated users only. Use AuthService.checktoken() on page load. If the check fails (in the catch block), redirect the user to index.html.
- **Logout:** Implement the logout button's functionality as described in the index.html section.
- **Optional (AI features)**: Add the necessary functionality to the **#translate-button** and **#generate-button** elements (click):
  - The translate button will translate the description text into english. **Detect** first the current language and **only translate** when it's not already in english (it will throw an error).
  - The generate button will generate a title ("headline") for the property based on the description text. You could provide a context like "A catchy title for selling a real state property in the market fast". Generate a title only when the description is at least 20 characters long or show an error message to the user (alert).
  - This is a new API, so it's **not supported by TypeScript** yet. You can declare the API classes as any and disable Eslint for this piece of code to avoid errors. You could put these functions in a separate file and export them.

```
/* eslint-disable */
declare const LanguageDetector: any;
declare const Translator: any;
declare const Summarizer: any;

async function detectLanguage(text: string): Promise<string> {...}

async function translate: Promise<string> (...}

async function summarize(text: string): Promise<string> {...}
/* eslint-enable */
```

Caution: There's a field named title in the form. This will give problems with TypeScript as when you type for example **form.title**, it will think that you're referring to the title atribute of the <form> element. To avoid that you could use a FormData object to parse the form or via the elements collection → **propertyForm.elements.namedItem("title")**.

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

Property details page (property-detail.html)

When we click a property's image or title, it should redirect here. This page displays the full details of a single property. Call GET → http://SERVER/properties/:id to get the property object with all its information.

- **Load Data:**
  - Get the property ID from the URL query string (e.g., property-detail.html?id=16). You can use location.search and URLSearchParams to extract the id.
    - If there's no id, redirect to index.html.
  - Fetch the property data from the server and populate the corresponding HTML elements (#property-title, #property-address, etc.). Do **not** use a template; directly manipulate the DOM elements.
  - The seller avatar and name are also links to his/her profile page. Update them with the corresponding user id (ex: profile.html?id=4).
- **Map Integration:** Once the property data is loaded, initialize the map centered on the property's location (latitude and longitude).
- **After** loading the property data (not before), check if the user is logged in
  - If logged in **and** the loaded property object (store it in a global variable) has the attribute **rated: false**, display the form for adding a new comment, update the top menu and add the click functionality to the logout button like you did in the index.html page.
- **Optional (User ratings)**:
  - **Star Ratings:** Ratings are represented by star characters (★ for filled, ☆ for empty). Create a helper function that takes a numeric rating and returns a string with the correct combination of stars.
  - Fetch and display existing comments for the property. Use the provided template and **prepend** each comment to the container to show the most recent ones first.
  - The star rating input is handled with CSS (don't worry about it). Your code only needs to read the selected rating value and the comment text, send them to the server, and then add the new comment to the top of the list upon success.
    - Along with the inserted rating, the server will return the new total rating of the property (update it in the page).
- **Mortgage Calculator:** Use this form to calculate the mensual rate the user would pay under certain conditions when getting a mortgage loan:
  - The first input field (read-only) should be pre-filled with the property's price.
  - The result will be shown in the **#monthly-payment** paragraph. Remove the hidden class from the **#mortgage-result** element first.

- Implement the mortgage calculation logic: After subtracting the down payment from the total price, apply the provided formula to the remaining amount.

$$M = P\frac{r(1+r)^n}{(1+r)^n - 1}$$

Definitions of Variables

- M = Total monthly payment
- P = Principal loan amount (the initial amount borrowed)
- r = Monthly interest rate (decimal)
  - *Note: This is your annual interest rate divided by 12. For example, if your annual rate is 5%, $r = 0.05/12 = 0.0041667$.*
- n = Total number of payments (months)
  - *Note: If your loan is for 25 years, $n = 25 \times 12 = 300$.*

User profile page (profile.html)

When we click on the profile icon link, or on a user's avatar or name, we'll go here.

This page displays the user's profile data. It can show the current user's profile or another user's profile if an ID is provided in the URL.

- This page is for authenticated users only. Call the checktoken method at the start. If the checktoken call fails (catch), redirect to index.html.
- 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.
- Populate the user's name, email, and avatar on the page and in the "Edit Profile" form fields.

  The server response will include a boolean field called **me**. Example:

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

  - If **me** is false (you are viewing someone else's profile), you must **hide** the "Edit Profile" and "Change Password" buttons, hide the avatar change overlay (.avatar-image-overlay), and **disable** the file input field (disabled = true).
- **Avatar Update:**
  - The user's image is a <label> for a hidden input[type=file].
  - When a file is selected, convert it to a **Base64** string and immediately call the server to update the avatar.
  - On success, update the avatar image on the page without a full reload.
- **Form Toggling:**
  - When the user clicks "Edit Profile" or "Change Password", hide the button and show the corresponding form.
  - If the form is submitted successfully **or** the user clicks "Cancel", hide the form and show the button again.
- **Data Updates:** When the user's profile data is successfully updated, reflect those changes on the page immediately without a reload.
- **User feedback**: Give the user feedback (alerts for example) when there's an error or the information has been updated successfully.
- **(Optional) View Properties Link:** Set the href of the "View User's Properties" button to index.html?seller=USER_ID, where USER_ID is the ID of the user whose profile is being viewed.

General Requirements & Important Notes

- **Version Control (important):**
  - You must create a **PRIVATE** GitHub repository for your project.
  - Share the repository with your teacher (read-only permissions are sufficient).
  - You must make **at least 2 meaningful commits per week**, starting from the week of November 10th.

- 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<PropertiesResponse>(`${SERVER}/properties`)
this.#http.post<SinglePropertyResponse, PropertyInsert>(`${SERVER}/properties`, property)
```

- Use classes to encapsulate functionality. For example, the **PropertiesService** class will have the necessary methods to call web services that include /**properties**/. A class named **UserService** will access the /**users**/ services, and a class called **AuthService** will access the /**auth**/ services.
- 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
```

- **Code Organization:**
  - Place all classes in a src/classes folder.
  - Place all interfaces in a src/interfaces folder. Do not put all interfaces in one file; create thematic files (e.g., user.interfaces.ts, property.interfaces.ts).
- **Imports:** With Vite, you no longer need to include file extensions in your import paths.
- **Type Safety: Avoid the use of any**. The provided ESLint configuration will enforce strict typing.
- **JS/TS Features:** If you wish to use modern language features, you can edit tsconfig.json and change the target from ES2022 to ES2024 or ESNext.
- **Async/Await:** When an asynchronous function is called from the top-level scope without using .then() or .catch(), ESLint will warn you. To fix this, simply chain a .catch(console.error) to log any potential errors.
- **HTML:** Do not modify the provided HTML files unless absolutely necessary, and only after consulting with the teacher.
- **Error Handling:** You must notify the user (at least with an alert()) when a form submission fails. For the profile page, also provide a success message when an

update is successful.

- **NPM Scripts:**
  - Rename the dev script to start.
  - Create a test script that runs ESLint on all files in the src folder.
  - Add the following scripts to your package.json:

```
"prestart": "npm run format && npm run test",
"format": "prettier --write --cache --parser typescript \"src/**/*.ts\""
```

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 {...} // Just remove the token from localStorage
}

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 ProvincesService {
  ...
  async getProvinces(): Promise<Province[]> {... }
  async getTowns(idProvince: number): Promise<Town[]> {...}
}


export class PropertiesService {
   ...
   async getProperties(urlSearch: URLSearchParams): Promise<PropertiesResponse> {... }
    async getPropertyById(id: number): Promise<Property> {...}
   async insertProperty(property: PropertyInsert): Promise<Property> {...}
   async deleteProperty(id: number): Promise<void> {...}
   async addRating(id: number, rating: RatingInsert): Promise<Rating> {... } // Optional
   async getRatings(id: number): Promise<Rating[]> {... } // Optional
}
```

Grading Breakdown

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

- Login Page: **1 point**
- Registration Page: **1 point**
- Main Listings Page (index.html): **2.5 points**
- Add Property Page: **1 point**
- Property Detail Page: **2.5 points**
- User Profile Page: **2 points**

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.

Point Deductions / Penalties

- **Failure to follow general requirements:** -0.25 to -0.5 points per rule.
- **Incorrect repository setup or missing commits:** -0.25 to -0.5 points.
- **ESLint warnings in the final submission:** -0.25 to -0.5 points.
- **ESLint errors in the final submission:** -0.5 to -1 point.
- **Poor Code Quality:** Up to -1 point deduction for issues such as:
  - Spaghetti code (lack of structure, everything in one file).
  - Poorly named variables/functions.
  - Using basic code (e.g., for loops) when more advanced, appropriate alternatives (e.g., filter, map) have been taught.
  - Excessive inline comments (JSDoc-style documentation comments above functions/classes are encouraged).
- **Functionality:** The application must be in a working state. If a core feature like login does not work, any dependent features (like the user profile) cannot be graded and will be considered incomplete.

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**:

- **(0.5 extra points)** Use SweetAlert or Bootstrap for showing feedback to the user in forms, asking them when to delete a property, etc.
- **(0.5 extra points)** Implement de Chrome AI API in the property-detail.html page to translate the description from other languages into english and to generate a title from the description.
    - Keep it mind that sometimes it takes a few seconds for the result.
- **(0,5 extra points)** Implement the logic to rate and comment a property (property-detail.html)
- **(0,5 extra points)** Add a link in the user's profile to filter the properties 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?seller=14**
    - When the seller id is present in the url, show only the properties created by these user, and **include the name** of the user in the text that shows the current filters applied. For example:

      Seller: Test user 2. Province: Alacant/Alicante. Search: Modern.
- **(1 extra point)** Learn about the CropperJS library to crop images and use it for the property's photo (new-property.html) 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 property's image will have an aspect ratio of 16/10 with a width of 1024px. The user avatar image's aspect ratio is 1 and its width should be 200px.
    - You can modify the HTML to include an area to crop the image (only visible when cropping) if you implement this.