



UNIDAD 1

JAVASCRIPT



Ejercicio final - AssBook

Desarrollo web del lado del cliente
2º curso - DAW
IES San Vicente 2024/2025 Autor:
Arturo Bernal Mayordomo

Índice

Introducción	3
Página de inicio de sesión (login.html)	4
Página de registro (register.html)	5
Página de eventos (index.html)	6
Paginación, orden y búsqueda.....	7
Opcional: Asistir a un acto.....	7
Añadir página de eventos (add-event.html)	8
Página de detalles del evento (event-detail.html)	9
Optativo: asistir a un acto	9
Página de perfil de usuario (profile.html)	11
Aspectos generales.....	13
Clases	15
Marcas.....	16
Contenido opcional (máximo 2 puntos adicionales).....	17

Introducción

Este ejercicio final será una adaptación a TypeScript y también una ampliación de las actividades que hemos estado realizando durante esta unidad (hasta la semana 4).

Te daré el esqueleto de la aplicación, y tendrás que completar los archivos TypeScript necesarios. Esto es lo que te doy:

- Las interfaces necesarias para los objetos de tipo y las respuestas del servidor se implementan en la carpeta **src/interfaces**.
- También se implementan las clases auxiliares **Http**, **MyGeolocation** y **MapService** (MapService es útil para crear y gestionar mapas).
- Todas las dependencias necesarias se proporcionan en el archivo **package.json**.
- Plantillas HTML (<template>) para eventos y asistentes a un evento. Si lo prefiere, cree plantillas Handlebars y utilice esa biblioteca.
- Además, tienes todos los **archivos HTML** necesarios, y la configuración de **Eslint** y **Prettier** para completar este proyecto. No edites ningún HTML a menos que esté muy justificado (**ejemplo**: partes opcionales como implementar CropperJS o modales con Bootstrap).

Servicios web

Los servicios web se despliegan en <https://api.fullstackpro.es/svtickets>. Puede ejecutarlos en su ordenador local junto con la base de datos para evitar que muchas personas inserten y eliminen datos al mismo tiempo.

El código fuente de los servicios web se encuentra en este repositorio de Github: <https://github.com/arturober/svtickets-services>. Estos servicios están debidamente **documentados** en la página de Github.

En ese repositorio también encontrarás (dentro del directorio **SQL**) el fichero para crear la base de datos en tu servidor local MySQL/MariaDB (XAMPP).

Para ejecutar la API de servicios en tu ordenador local. Inicia XAMPP y la base de datos en el directorio de servicios:

- Importe el archivo **SQL/svtickets.sql**. Editar el archivo **src/mikro-orm.config.ts** y establecer el usuario y contraseña correctos ('root' y ").
- Ejecutar **npm i** (primera vez) y **npm start**
- Utilice la url <http://localhost:3000> en su cliente y Postman como base.

Página de inicio de sesión (login.html)

Esta página **login.html**, tendrá un formulario de login. En este formulario el usuario introducirá su email y contraseña. Además, geolocalizará al usuario, y enviará la latitud y longitud del usuario para que se actualice en el servidor.

El servicio web al que se llama para el inicio de sesión es: [POST →](#)

<http://SERVER/auth/login> Ejemplo de solicitud de datos (en el inicio de sesión, también actualizamos la lat y lng del usuario):

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

Respuesta: Devuelve un token de autenticación que debes almacenar (localStorage):

```
{  
  "accessToken":  
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NywiaWF0IjoxNjAzODIxNTcxLCJleHAiOiJlE2MzUzNTc1NzF9.XK1RY-5C4UvhQlb7d8ack2718IKrx71va1ukURz7_NI"  
}
```

Cualquier error dará lugar a un estado distinto de 200, normalmente **401** (no autorizado). Utilice **catch** para capturarlo. Contendrá el siguiente JSON:

```
{  
  "status": 401,  
  "error": "Correo electrónico o contraseña incorrectos"  
}
```

Cuando el login sea exitoso, guarda el **token** dentro de una variable **LocalStorage** (usa la clave '**token**' ya que la clase Http está configurada para obtener este valor). Necesitarás este token en otros servicios web, o devolverán un error de autenticación. Esto lo hará automáticamente la clase HTTP siempre que hayas almacenado antes el token.

Una vez iniciada la sesión, rediríjase a la [página Eventos](#).

Si hay un error, muéstraselo al usuario (Puedes mostrarlo en el **p#errorInfo** párrafo, o dentro de un mensaje de alerta utilizando [sweetalert](#) o [Bootstrap](#) por ejemplo).

Página de registro (register.html)

En esta página, habrá un formulario para registrar un nuevo usuario. En este formulario se debe enviar la siguiente información:

- **nombre**
- **email** (y un campo llamado **email2** para repetir la información del email).
- **contraseña**
- **lat, lng** (campos de sólo lectura)
- **avatar** → Foto del usuario enviada en formato base64.

Geolocaliza al usuario y establece los valores de los campos lat y lng. Si hay algún error, utiliza los valores por defecto y muestra un mensaje de error al usuario.

Cuando el usuario envíe este formulario, comprueba si ambos emails son iguales o muestra un error y no continúa. Si son iguales crea un objeto con estas propiedades y envíalo al servicio: POST → <http://SERVER/auth/register>.

Este servicio devolverá un estado 200 (OK) que contiene el objeto de usuario insertado o un error, normalmente el estado 400 (Bad Request) como este:

```
{
  "statusCode":400, "message":
  [
    "el nombre no debe estar vacío", "el
    correo electrónico no debe estar
    vacío", "el correo electrónico debe
    ser un correo electrónico", "la
    contraseña no debe estar vacía", "la
    contraseña debe ser una cadena"
  ],
  "error": "Bad Request"
}
```

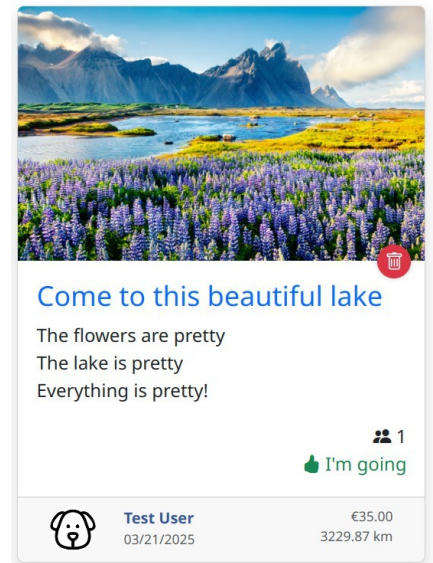
Si obtiene algún mensaje de error, muéstreselos todos al usuario. Si todo ha ido bien, redirige a la [página de inicio de sesión](#).

Página de eventos (index.html)

En esta página se mostrarán los eventos, igual que en los ejercicios anteriores. Llama a GET → <http://SERVER/events> para obtenerlos. Estas son las principales diferencias:

Cada evento contendrá el campo **mío : booleano**, que indicará si el evento fue creado por ti o no. También contendrá nueva información como el usuario que creó el evento (**creador**), si vas a asistir al evento o no (asistir), o la distancia del evento a tu ubicación en kilómetros. Ejemplo:

```
{
  "id": 2144,
  "creador": {
    "id": 45,
    "nombre": "Otro usuario",
    "email": "test3@email.com",
    "avatar": "http://SERVER/img/users/1633863254327.jpg", "lat":
    38,
    "lng": -0.5
  },
  "title": "Actualización de eventos",
  "description": "Actualización de la
  descripción", "date": "2021-12-03
  00:00:00",
  "precio": 25,
  "lat": 35.23434,
  "lng": -0.63453,
  "dirección": "Calle nada",
  "image": "http://SERVER/img/events/1633860306456.jpg",
  "numAttend": 0,
  "distancia": 0,
  "atender":
  false, "mina":
  true
}
```



Puedes encontrar la plantilla actualizada del manillar en el directorio **templates/**. En él tiene que formatear el precio, la distancia y la fecha antes de llamar a la plantilla.

Cada evento contendrá el campo **mío (booleano)**, que indicará si el evento fue creado por usted o no. **Sólo el** evento creado por el usuario registrado tendrá el botón "Eliminar". **No olvides esto!**

Ahora, el evento tiene un creador, y el servidor devuelve esa información dentro del evento (**creador**). Usa el avatar y el nombre de ese **creador** en la tarjeta.

Al hacer clic en la imagen o en el título del evento, se redirigirá a la [página Detalles del evento](#) con el id del evento actual → Ejemplo: **event-detail.html?id=23**.

Paginación, orden y búsqueda

Ahora, la barra de búsqueda y los botones de pedido harán una llamada al servidor para obtener los eventos filtrados y ordenados. Los botones de orden enviarán un parámetro llamado **orden** con uno de estos valores ("distancia", "precio", "orden").

La barra de búsqueda tiene ahora un botón para filtrar los eventos en base a un texto (sólo por título). Genera un parámetro llamado **search** con el valor e inclúyelo en la url para obtener los eventos.


Aparte de la lista de eventos, el servidor devuelve más propiedades en la respuesta como **page** (página actual) y **more** (si hay otra página). Cuando hay más eventos para cargar (more: **true**), el elemento `button#loadMore` aumentará el número de la página actual y lo incluirá en la URL. La carga de más eventos debe añadirlos al contenedor HTML, y no reemplazarlos.

Para generar una URL que contenga estos parámetros, utilice un objeto **URLSearchParams** y concaténalo a la URL después del carácter '?' de la siguiente manera:

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

Actualizar el texto debajo de la barra de navegación, para que el usuario sepa los filtros que se aplican actualmente (por defecto order="distancia" y search=""):

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



Ordering by: price. Searching by: party

Opcional: Asistir a un acto

En la parte derecha de la tarjeta, verá el número de asistentes a un evento y un texto (con un icono) que indica si va a asistir o no. Al pulsar sobre este elemento **div.attend-button**, llama a la URL <http://SERVER/events/:idEvent/attend> usando POST o DELETE, dependiendo del valor actual de **event.attend**. Si la llamada tiene éxito, cambia la clase del icono (dos pulgares hacia arriba ↔ dos pulgares hacia abajo) y el texto mostrado (Voy ↔ No voy), dependiendo del valor actual de **event.attend** (cambia también este booleano). Actualiza también el número de asistentes (++ o --).

Si no vas a hacer esta parte opcional puedes eliminar este HTML de la plantilla.

Añadir página de eventos (add-event.html)

Esta página contendrá un formulario para crear un nuevo evento. Será similar a lo que hemos hecho en ejercicios anteriores pero añadiendo estas características.

- Mostrar un mapa dentro del elemento **div#map**. Geolocaliza al usuario y muestra un marcador con la posición actual del usuario.
- Mostrar una entrada Autocompletar en el elemento **div#autocomplete** para que el usuario pueda escribir una dirección. Al actualizar la posición, mostrar el marcador y centrar el mapa en el nuevo lugar. Guarda también las coordenadas (lat, lng) en variables para enviarlas al servidor.

◦ Para obtener el valor de la dirección, puedes hacerlo así:

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

Este es un ejemplo de lo que debe enviar al servicio: POST → http://SERVER/events:

```
{
  "título": "Se trata de un nuevo evento",
  "descripción": "Descripción para el nuevo evento",
  "date": "2021-12-03",
  "precio": 25.35,
  "dirección": "Ninguna
parte", "lat": 35.23434,
  "lng": -0.63453,
  "imagen": "Imagen en BASE64"
}
```

Cuando todo vaya bien, redirige a la [página de eventos](#) . O si algo va mal, muestra un error en el HTML (o una alerta con la librería sweetalert2).

Página de detalles del evento (event-detail.html)

Cuando hacemos clic en la imagen o el título de un evento, debería redirigirnos aquí.

En esta página mostraremos los detalles de un evento. Recibimos el id del evento en la url así: **event-detail.html?id=3**. Usa la propiedad **location.search** para acceder a la parte (?id=3) y extraer el id de ahí. Si no hay id, redirige a **index.html**.

Llamar al servicio GET → <http://SERVER/events/:id> servicio (:id será el número → Ejemplo: <http://SERVER/events/4>), para obtener la información del evento. Debería devolver un objeto de evento (igual que en la página index.html, pero sólo uno) → **{ event: {...} }**.

Si el evento no existe, devolverá un error 404 (No encontrado).

Muestra la información del evento en esta página creando la misma tarjeta que en la página index.html (la misma plantilla está presente aquí). Ponla dentro del elemento **div#eventContainer**. No queremos repetir código innecesario:

- Pon el código para crear la tarjeta y todo el HTML que contiene en una función externa. Por ejemplo, puedes crear un método en la clase **EventService** que devuelva el elemento **div.col** creado. Podría ser algo como **toHTML(event: MyEvent, template: HTMLTemplateElement): HTMLDivElement**.
 - El manejador de clic de los botones de borrar y atender no irá dentro de este método, sino que se establecerá fuera. Esto es porque las acciones después de borrar o atender (opcional) el evento serán diferentes en index.html (eliminar la tarjeta del DOM), y aquí (redirigir a index.html)

Después de borrar un evento, redirige a la página index.html.

Cargar un mapa en el elemento **div#map** y mostrar la posición del evento con un marcador (propiedades lat y lng). Mostrar la dirección en el elemento **div#address**.

Opcional: asistir a un acto

Si implementas la funcionalidad de asistir a un evento, debes mostrar la lista de usuarios que van a este evento. Hay un elemento **template#attendTemplate** que contiene el HTML para mostrar la información de un usuario. Llama al servicio <http://SERVER/events/:idEvent/attend> usando GET y envía a la plantilla un objeto con la lista de usuarios (como la respuesta del servidor). Ejemplo:

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

```
}  
1,  
...  
}
```

Añade estos usuarios a la lista **ul#userList**. Al pulsar el botón de la tarjeta que cambia si se asiste o no al evento (**div.attend-button**), recarga la lista de usuarios de abajo tras llamar al servicio correspondiente (pero no recargues la página en ningún caso).

Página de perfil de usuario (profile.html)

Cuando hagamos clic en el enlace "Mi perfil", o en la imagen de avatar de un usuario, iremos aquí.

Esta página mostrará la información de algunos usuarios. Puede recibir un id de usuario en la url (profile.html?id=3). Si no recibe ningún id, mostrará el perfil del usuario registrado. Llamada: GET → <http://SERVER/users/me>. O si recibe un id, llama en su lugar: GET → <http://SERVER/users/:id>.

Ambos servicios devolverán un objeto JSON como éste:

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

Coloca los datos dentro de los elementos HTML correspondientes (avatar → **#avatar**, nombre → **#nombre**, email → **#email > pequeño**). La propiedad "me" indica si ese perfil es tu perfil o el de otro usuario. Cuando no es tu perfil, oculta los botones correspondientes para editar la foto (etiqueta), el perfil y la contraseña (usa la clase `d-none` o elimínalos del DOM).

Crea también un mapa que muestre la posición del usuario en el elemento

`div#map`. Cuando se trata de su perfil, así es como funcionan los botones:

- **Editar avatar** → El elemento `<label>` junto al avatar se adjunta al archivo de tipo de entrada (oculto). Cuando el usuario hace clic en la etiqueta es como hacer clic en la entrada para seleccionar un archivo.
 - Cuando el usuario selecciona un archivo (evento de cambio en la entrada), transformarlo a base64 y enviarlo al servidor (PUT: <http://SERVER/users/me/photo>) en un objeto como este:

```
{
  "avatar": "Foto en base64"
}
```
 - Si el servidor responde correctamente, devolverá una respuesta con la misma estructura que contiene la url del avatar. Colócalo en la imagen de perfil.
- **Editar perfil** → Cuando el usuario haga clic en el botón **#editProfile**, oculta el elemento **#profileInfo** y muestra en su lugar el elemento **#profileForm** para mostrar el formulario que tiene 2 botones (guardar y cancelar).

- Si el usuario pulsa cancelar, ocultar **#profileForm** y mostrar **#profileInfo**.
- En enviar el formulario formulario, envíe a un objeto objeto a al servidor servidor (PUT → <http://SERVER/users/me>) con el nombre un correo electrónico a modificar:

```
{
  "email": "test@email.com",
  "nombre": "Prueba de
persona"
}
```

- Cuando el servidor responda sin error, cambia el email y el nombre en el HTML y oculta el formulario, mostrando de nuevo el elemento **#profileInfo**.
- **Editar contraseña** → Igual que antes, pero mostrando el elemento **#passwordForm** en su lugar.
 - Comprueba que ambas contraseñas son iguales (o muestra un error) antes de enviar la nueva contraseña al servidor.
 - El servicio a llamar es PUT: <http://SERVER/users/me/password>, enviando un objeto como este:

```
{
  "contraseña": "1234",
}
```

- Cuando el servidor responda, oculta el formulario y vuelve a mostrar **#profileInfo**.

Los formularios de perfil y contraseña devolverán sólo un código de confirmación del servidor (Estado 204, sin contenido).

Aspectos generales

- Al cargar cualquier página: comprueba si hay un token almacenado en tu localStorage. Si hay un token, llama a GET → <http://SERVER/auth/validate>. Dependiendo de la página en la que te encuentres, harás lo siguiente:
 - **login, register** → Si no devuelve un error (el usuario ha iniciado sesión), redirige a la [página de eventos](#) (index.html) directamente. Si devuelve un error, elimine el token del almacenamiento local (no válido).
 - **Otras páginas** → Si no ha iniciado sesión (obtiene un error), redirija a la página login.html.
- Los métodos de la clase Http están tipados usando genéricos <T>. Esto significa que al llamar a un método, debes indicarle el tipo de la respuesta. En los métodos post y put, también debes indicar el tipo para los datos enviados al servidor. Existen interfaces para cada respuesta en el fichero **src/interfaces/responses.ts**. Ejemplo:
 - `this.#http.get<EventsResponse>(`${SERVER}/events`)`
 - `this.#http.post<SingleEventResponse, EventInsert>(`${SERVER}/events`, event)`
- Utilice clases para encapsular la funcionalidad. Por ejemplo, la clase **EventService** tendrá los métodos necesarios para llamar a servicios web que incluyan **/eventos/**. Una clase llamada **UserService** accederá a los servicios **/users/**, y una clase llamada **AuthService** accederá a los servicios **/auth/**.
- El botón de **cierre de sesión** eliminará el token del almacenamiento local y redirigirá a la página de inicio de sesión. Querrás incluir esta funcionalidad dentro de la clase AuthService por ejemplo. Cada página que tiene el enlace de cierre de sesión tendrá que manejar el evento de clic para ello.
- Como este ejercicio debe hacerse en TypeScript. Todas las variables y parámetros, deben incluir su **tipo** cuando no esté claro en la asignación (también se recomienda especificar el **tipo de retorno** para funciones y métodos).
 - Por ejemplo, si un método devuelve una Promise con un evento dentro, el tipo de retorno será `Promise<MiEvento>`. Si no devuelve nada, **será void**. Todos los métodos asíncronos devuelven una Promise, aunque sea `Promise<void>`.
- Por defecto, todos los elementos HTML son devueltos desde el DOM como `HTMLElement` genérico. Tendrás que convertirlos al elemento correcto si quieres utilizar propiedades específicas:

```
let image = document.getElementById("image") as HTMLImageElement;
```

```
img.src = ...; // OK
```

```
let value = (form.date as HTMLInputElement).value; // OK
```

- **Importante:** El formulario para añadir un nuevo evento tiene un campo llamado "title". El atributo title (string) es una propiedad de cualquier `HTMLElement`. Si intentas convertirlo a `HTMLInputElement`, TypeScript no te dejará hacerlo **usando form.title como HTMLInputElement**. Usa la colección de elementos para acceder a la entrada en su lugar:
 - `(newRestForm.elements.getName("title") como HTMLInputElement).value`

Clases

Estas son las clases y métodos recomendados que podrías implementar. Por supuesto, **puedes añadir más funcionalidad**, o hacer las cosas de una manera diferente (no tienes que usar **async** por ejemplo):

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

exportar clase UserService {
  ...
  async getProfile(id?: number): Promise<Usuario> {...}
  async guardarPerfil(nombre: cadena, email: cadena): Promise<void> {...}
  async saveAvatar(avatar: string): Promise<string> {...}
  async guardarContraseña(contraseña: cadena): Promise<void> {...}
}

export class ServicioDeEventos {
  ...
  async getEvents(página = 1, orden = "distancia", búsqueda = ""): Promise<EventsResponse> {...}
  async getEvent(id: número): Promise<MiEvento> {...} async
  post(evento: MiEventoInsertar): Promise<MyEvent> {...} async
  delete(id: número): Promise<void> {...}
  async getAttendees(id: number): Promise<Usuario[]> {...} // Opcional
  async postAttend(id: número): Promise<void> {...} // Opcional async
  deleteAttend(id: número): Promise<void> {...} // Opcional
  toHTML(evento: MyEvent, eventoTemplate: HTMLTemplateElement): HTMLDivElement {...} // Crear la tarjeta de
  evento aquí (El evento de borrado clic fuera)
}
```

Te doy las interfaces (MyEvent, User, etc..., y respuestas del servidor) ya **implementadas** (revisa el proyecto).

Marcas

La nota final se calculará en función de estos criterios (lo ya aplicado en los ejercicios anteriores no cuenta):

- Página de inicio de sesión → 1 punto
- Página de registro → 1 punto
- Página de eventos (index.html) → 2 puntos
- Añadir página de eventos → 1 punto
- Página de detalles del evento → 1,5 puntos
- Página de perfil → 2,5 puntos
- Código limpio y estructurado → 1 punto

Aunque todo funcione, pueden restarse puntos a la nota de cada sección si el código no se considera adecuado. Formatea y organiza bien tu código y evita las repeticiones innecesarias.

Contenido opcional (máximo 2 puntos adicionales)

Estas 3 ampliaciones subirán cada una la nota final del ejercicio, pero sólo si la nota final sin contenido optativo es igual **o superior a 7**:

- **(1 punto extra)** Utiliza SweetAlert o Bootstrap para mostrar mensajes de error (y éxito) al usuario (0,25), la confirmación para borrar un evento (0,25), y los formularios para actualizar la información del perfil y la contraseña (0,25 cada uno).
- **(1 punto extra)** Implementar la funcionalidad de asistencia del usuario a un evento en las páginas de índice y detalles del evento.
- **(0,5 puntos extra)** Añade un enlace en el perfil del usuario para filtrar los eventos creados por el usuario actual. Este enlace redirigirá a la página `index.html` pero incluyendo el id de este usuario así: **`index.html?creator=14`**
 - Cuando el par creador está presente en la url, mostrar sólo los eventos creados por este usuario, e incluir el nombre del usuario en el texto que muestra los filtros actuales aplicados. Por ejemplo

Creador: Peter. Ordenar por: precio. Búsqueda por: am

- **(1 punto extra)** Conoce la librería [CropperJS](#) para recortar imágenes y utilízala para la foto del evento (añadir evento) y el avatar del usuario (registrarse y actualizar perfil) .
 - Descarga la dependencia **cropperjs** y sus definiciones **@types/cropperjs** TypeScript. Inclúyelo en tu código con **`import Cropper from 'cropperjs'`**.
 - La imagen del evento tendrá una relación de aspecto de 16/9 con una anchura de 1024px. La relación de aspecto de la imagen del avatar del usuario es 1 y su anchura debe ser de 200px.