**Capítulo 5: Desarrollo e implementación**

Este capítulo describe el proceso de implementación del sistema, tanto en su versión web como en su aplicación móvil, así como la estructura de la base de datos. Se detallan las funcionalidades principales desarrolladas, los componentes utilizados en Angular y Android, y los servicios que permiten la interacción con la base de datos en la nube mediante Firebase. Además, se incluyen fragmentos representativos de código para ilustrar cómo se han resuelto técnicamente los distintos requisitos funcionales definidos en fases anteriores del proyecto. La estructura modular del sistema ha permitido implementar cada parte de forma independiente, manteniendo una comunicación fluida entre interfaces.

# Antecedentes

Como se ha comentado en la introducción del proyecto, este trabajo contaba con una versión inicial con respecto a la tecnología Android y Firebase. Con el objetivo de enfocar este capítulo en el desarrollo propio, se dedica esta sección a enumerar y describir brevemente las funcionalidades con las que contaba este sistema previamente.

## Aplicación móvil

La versión inicial ya contaba con una versión Android operativa y básica, desarrollada con el lenguaje Java, y que contaba con las funcionalidades que se enumeran a continuación

* **Bluetooth**: Se establecía la conexión por Bluetooth con los tres sensores implicados en las pruebas.
* **Conexión con Firebase**: Estaba establecida la conexión con Firebase en modo escritura.
* **Pruebas**: Se ejecutaban las pruebas SPPB y se subía el resultado a Firebase.
* **Usuarios**: Un registro de usuarios básico
* **Interfaz**: Disponía de una interfaz de usuario con las pantallas funcionales, pero muy limitadas en diseño, estructura y experiencia de usuario.

Sin embargo, en esta versión previa, gran parte del procesamiento se realizaba en los sensores, que enviaban resultados de las pruebas ya procesados al móvil. Esto ha cambiado durante el desarrollo de este proyecto, implementando todo el procesamiento en el smartphone, limitando los sensores a solo emitir datos en crudo.

## Base de datos

Estaba creada una instancia de Firebase configurada con una estructura inicial de base de datos en tiempo real. Las funcionalidades incluían:

* Escritura de datos desde la app móvil
* Almacenamiento de usuarios y pruebas
* Proyecto Firebase correctamente enlazado al paquete de la app Android.

En este caso, la estructura no contemplaba a los facultativos registrados ni la gestión por roles de usuario.

## Web

El sistema contaba con una web muy básica y sin funcionalidad, por lo que se ha optado por hacer el desarrollo web desde cero.

# Base de datos

--

# Android

--

# Web

## Creación del proyecto Angular

Para la creación de un proyecto en Angular es necesario contar con una serie de herramientas que conforman el entorno de desarrollo, el primer paso se basa en la instalación de estos dos paquetes:

* **Node.js**: Es necesario para usar el sistema de gestión de paquetes que incluye, npm(Node Package Manager), que se usa para instalar las dependencias del proyecto y las bibliotecas necesarias. Node.js se puede descargar e instalar desde su sitio oficial https://nodejs.org/.
* **Angular CLI (Command Line Interface)**: Se instala a través de npm y es una herramienta oficial para habilitar los comandos de Angular por terminal, permitiendo la creación y configuración del proyecto. Se instala globalmente en el sistema utilizando el comando del listado 1 en la terminal, una vez instalado Node.js.

Listado 1 - Comando para la instalación de Angular CLI

*npm install -g @angular/cli*

Una vez instalada la CLI, se puede crear un proyecto nuevo en Angular con el comando ‘*ng new’*, que crea la estructura básica, con la configuración, módulos, y componentes principales necesarios para poder compilar y empezar a trabajar en Angular.

Se crea automáticamente un primer componente que se compone, como todos, por tres ficheros:

* **HTML**: Donde se define la estructura y contenido de la interfaz de usuario
* **TypeScript (TS):** Gestiona toda la lógica del componente.
* **CSS**: Se encarga de crear clases CSS que definen el diseño visual y el estilo de la interfaz, como los colores, fuentes y disposición de los elementos.

## Estructura y archivos clave del proyecto Angular

La aplicación presenta una estructura modular, limpia y organizada, pensada para facilitar la escalabilidad, el mantenimiento y la comprensión del código. A continuación, se describen las carpetas y archivos más relevantes que conforman la lógica del sistema, ubicados principalmente en el directorio */src/app*.

### Carpetas principales

* **components/**: Contiene todos los componentes de la aplicación, organizados por funcionalidades o pantallas. Cada subcarpeta agrupa un conjunto de archivos (.ts, .html, .scss) correspondientes a un componente específico, como dashboard, login-admin, pacientes, ver-paciente o equilibrio. Esta estructura permite segmentar el desarrollo y mantener cada unidad de interfaz separada y reutilizable.
* **Servicios**: Albergan los servicios que gestionan la lógica de negocio o la conexión con Firebase.
* **shared/ y guard/:** Carpeta auxiliar con elementos reutilizables, como servicios compartidos, control de autenticación o protección de rutas.

### Archivos clave

* **app.module.ts**: Es el módulo principal de la aplicación. Declara los componentes utilizados, importa los módulos de Angular necesarios y define los servicios globales. Funciona como punto de entrada de la aplicación.
* **app-routing.module.ts**: Configura el sistema de enrutamiento que permite navegar entre los distintos componentes sin recargar la página, característica propia de las SPA (Single Page Applications). Cada ruta está asociada a un componente concreto, como el login, la vista de pacientes o el panel de administración.
* **angular.json**: Archivo de configuración global del proyecto. Establece las rutas, estilos, scripts y parámetros de compilación. Permite personalizar el comportamiento del entorno según las necesidades del desarrollo.
* **environment.ts**: Contiene las variables de entorno del proyecto. Es especialmente útil para definir configuraciones distintas entre desarrollo y producción, como las claves de conexión a Firebase, sin necesidad de modificar directamente el código principal.
* **package.json**: Define las dependencias, versiones y scripts del proyecto. Incluye bibliotecas como Angular, Firebase y otras herramientas auxiliares. Gracias a este archivo, es posible instalar todo el entorno de trabajo mediante el comando npm install.
* **main.ts, index.html, styles.scss**: Archivos base del arranque de la aplicación. main.ts inicializa Angular, index.html define el punto de inserción de la vista, y styles.scss contiene los estilos globales de la interfaz.
* **assets/**: Carpeta destinada al almacenamiento de recursos estáticos como imágenes o íconos utilizados en la interfaz web.

Esta organización estructural no solo mejora la legibilidad y el orden del proyecto, sino que permite incorporar nuevas funcionalidades sin interferir con el código ya implementado, lo cual es fundamental para poder ampliarse en el futuro.

## Vinculación con Firebase

Para vincular la aplicación web con Firebase, en primer lugar, es necesario instalar los paquetes necesarios desde el Angular CLI, con el comando del listado 2.

Listado 2 - Instalación paquetes Angular

*npm install firebase @angular/fire*

Una vez instalados los paquetes, se configura el entorno definiendo un objeto de configuración con las claves del proyecto Angular, que se ha obtenido de la web de firebase. Esto se modifica en el fichero *environment.ts* (*Listado 3*).

Listado 3 - Configuración archivo enviroment.ts

export const environment = {

  production: false,

  firebase: {

    apiKey: "AIzaSyCsmqP1XXXXXXXXXXXVMFhVI3az\_yTq0ZzM",

    authDomain: "tvgi-8700f.firebaseapp.com",

    databaseURL: "https://tvgi-8700f.firebaseio.com",

    projectId: "tvgi-8700f",

    storageBucket: "tvgi-8700f.appspot.com",

    messagingSenderId: "30305207825",

    appId: "1:30305207825:web:XXXXXXXXXXXXXXXXX",

    measurementId: "G-H7SXXXXXXH"

  },

};

Además, hay que inicializar firebase en le module.ts del proyecto, tal como se muestra en el listado 4.

Listado 4 - Archivo module.ts

import { AuthService } from './shared/services/auth.service';

import { AngularFireModule } from '@angular/fire/compat';

import { AngularFireAuthModule } from '@angular/fire/compat/auth';

import { AngularFireDatabaseModule } from '@angular/fire/compat/database';

import { AngularFirestoreModule } from '@angular/fire/compat/firestore';

import { AngularFireStorageModule } from '@angular/fire/compat/storage';

@NgModule({

  imports: [

    BrowserModule,

    AngularFireModule.initializeApp(environment.firebase),

    AngularFireAuthModule,

    AngularFirestoreModule,

    AngularFireStorageModule,

    AngularFireDatabaseModule

  ],

  providers: [AuthService],

  bootstrap: [AppComponent],

})

export class AppModule { }

## Uso de servicios personalizados

Se han creado dos servicios para encapsular las operaciones más habituales, como leer/escribir en la base de datos o gestionar usuarios. Estos servicios se inyectan en los componentes que los necesiten mediante inyección de dependencias.

Se implementan en el directorio *src/services/*

### firestoreService

Este servicio es el encargado de centralizar la comunicación entre la aplicación y la plataforma Firebase y actúa como capa intermedia entre la lógica de la aplicación y la base de datos, permitiendo realizar operaciones de lectura, escritura y eliminación de forma estructurada.

A continuación, se describen los métodos utilizados en este servicio:

* **getPacientes():** Obtiene una lista reactiva de todos los pacientes registrados en la web. *(Listado 5)*
* **getPaciente(idPaciente: string):** Obtiene los datos de un paciente en particular.
* **getUsuarios**(): Obtiene una lista reactiva de todos los usuarios registrados en la web (no pacientes). *(Listado 5)*
* **modRolUsuario(uid: string, newrol: string**): Modifica el rol de un usuario *(Listado 6)*
* **updatePaciente(documentId: string, data: any)**: Modifica los datos de un paciente. *(Listado 7)*
* **deletePaciente(documentId: string, numHistorial: string**): Elimina completamente de la base de datos un paciente, así como todas sus pruebas realizadas. *(Listado 8)*
* **getPacienteTBA / TGS y TCS(numHistorial: string):** Obtiene las pruebas realizadas por un paciente en concreto, cada método obtiene un tipo de prueba. *(Listado 9)*
* **loginAdmin(username: string, password: string):** Realiza el inicio de sesión de un facultativo. *(Listado 10)*
* **createSanitario(username: string, password: string):** Crea en la base de datos una entrada para un facultativo nuevo. *(Listado 11)*
* **deleteSanitario(uid: string):** Borra una entrada de un facultativo concreto. *(Listado 12)*

Cabe destacar a nivel técnico, que este servicio hace uso de programación reactiva mediante herramientas nativas de Angular que ayudan a mantener un código limpio y escalable como el uso de suscripciones (subscribe) u operadores de transformación (pipe, map) para trabajar con datos en tiempo real. También se apoya en promesas (.then, .catch) para operaciones asíncronas en Firestore.

Un ejemplo de esto lo tenemos en el método deletePaciente() (Listado 12) donde se utiliza un subscribe, siguiendo una lógica tal que:

1. **Consulta reactiva**: snapshotChanges() crea un *observable* que escucha los cambios en tiempo real de los nodos con numHistorial igual al indicado.
2. **Subscribirse al observable**: con .subscribe(...), el código se ejecuta **solo cuando llegan los datos** desde Firebase.
3. **Eliminación individual**: por cada prueba encontrada, se obtiene su clave (payload.key) y se llama a .remove(...)

Esto es necesario porque las lecturas en Firebase son asíncronas, lo que significa que no se tiene claro cuándo va a llegar una respuesta. De esta forma el subscribe permite que no se ejecute el código hasta que no se ha recibido respuesta desde la base de datos, evitando fallos en la aplicación, bloqueos y mejorando el rendimiento.

La creación de este servicio permite mantener el código modular y reutilizable, separando la lógica de negocio de las operaciones sobre la base de datos.

Listado 5 - Método getPacienteTBA()

  public getPacienteTBA(numHistorial: string) {

    console.log('getPacientesTBA: ' + numHistorial)

    return this.db.list('tBA', ref => ref.orderByChild("numHistorial").equalTo(numHistorial)).snapshotChanges();

  }

  public modRolUsuario(uid: string, newrol: string) {

    const userRef: AngularFirestoreDocument<any> = this.afs.doc(

      `sanitarios/${uid}`

    );

    userRef.ref.get().then((documentSnapshot) => {

      if (documentSnapshot.exists) {

        console.log("Actualizamos entrada en Firebase")

        userRef.set({ role: newrol }, { merge: true, });

      }

    });

  }

  public createSanitario(username: string, password: string): Promise<void> {

    const userRef: AngularFirestoreDocument<any> = this.firestore.doc(`sanitarios/${username}`);

    return userRef.set({

      username: username,

      password: password,

      role: 'no-role'

    });

  }

Listado 6 - Método createSanitario()

Listado 7 - Método modRolUsuario()

  public getPacientes() {

    return this.db.list('usuarios/pacientes').snapshotChanges();

  }

Listado 8 - Método getPacientes()

Listado 10 - Método getPaciente()

  public getPaciente(idPaciente: string) {

    return this.db.list('usuarios/pacientes/' + idPaciente).snapshotChanges();

  }

  public deleteSanitario(uid: string): Promise<void> {

    return this.firestore.doc(`sanitarios/${uid}`).delete().then(() => {

      console.log(`Sanitario ${uid} eliminado`);

    }).catch(error => {

      console.error('Error al eliminar el sanitario:', error);

    });

  }

Listado 9 - Método deleteSanitario()

Listado 12 - Método deletePaciente()

Listado 13 - Método updatePaciente()

  public updatePaciente(documentId: string, data: any) {

    const pacientesRef = this.db.list('usuarios/pacientes')

    pacientesRef.update(documentId, data)

  }

Listado 14 - Método getUsuarios()

  public getUsuarios() {

    return this.firestore.collection('sanitarios').snapshotChanges();

  }

const equilibrioRef = this.db.list('tBA', ref => ref.orderByChild("numHistorial").equalTo(numHistorial))

equilibrioRef.snapshotChanges().subscribe((equilibrioSnapshot) => {

      equilibrioSnapshot.forEach((equilibrioData: any) => {

        equilibrioRef.remove(equilibrioData.payload.key)

Listado 11 - Método loginAdmin()

  loginAdmin(username: string, password: string): Observable<string | null> {

    return this.firestore.collection('sanitarios', ref =>

      ref.where('username', '==', username).where('password', '==', password)

    ).get().pipe(

      map(snapshot => {

        if (snapshot.empty) return null;

        const docData = snapshot.docs[0].data() as { role?: string };

        return docData.role || null;

      })

    );

  }

### GlobalService

El segundo servicio clave implementado es GlobalService, cuya función principal es definir variables que serán accesibles en toda la web.

El objetivo es que una vez se haya iniciado sesión, se conozca en todo momento el rol que tiene el usuario que se ha registrado, así como su userName. De esta forma podemos definir desde cualquier componente aquellas partes de la aplicación que son visibles o accesibles y aquellas que no, dependiendo del rol.

Este servicio utiliza la clase BehaviorSubject de la librería RxJS, que permite crear observables con valor inicial y actualizable, lo cual es especialmente útil para almacenar datos que cambian durante la ejecución de la aplicación, como los que se obtienen tras iniciar sesión.

Se definen los observables clave:

* role$
* name$

Se definen los métodos para modificar estos observables u obtener su valor:

* updateRole()
* getCurrentRole()
* updateName()
* getCurrentName()

De este modo, una vez que el usuario se autentica, el componente de login actualiza el valor del rol o del nombre mediante los métodos updateRole(newRole: string) o updateName(newName: string). Posteriormente, cualquier otro componente que necesite acceder a esta información puede hacerlo de dos formas:

* **De forma reactiva**, suscribiéndose a role$ o name$, que son observables que emiten automáticamente los cambios.
* **De forma directa**, mediante los métodos getCurrentRole() o getCurrentName(), que devuelven el valor actual sin necesidad de suscripción.

## 4.5. Funcionalidades

### 4.5.1. Inicio de sesión y registro

El inicio de sesión y el registro de usuarios son las primeras pantallas de la aplicación, y clave para cumplir con los requisitos funcionales basados en la seguridad.

Para su implementación, se crean los siguientes componentes:

* Sign-in *(Figura 1)*
* Sign-up *(Figura 2)*
* Login-admin *(Figura 3)*
* Login-patient *(Figura 4)*

Sign-in.component tiene como finalidad redirigir a uno de los tres componentes restantes, según el botón que el usuario pulse, haciendo uso de routerLink de Angular.

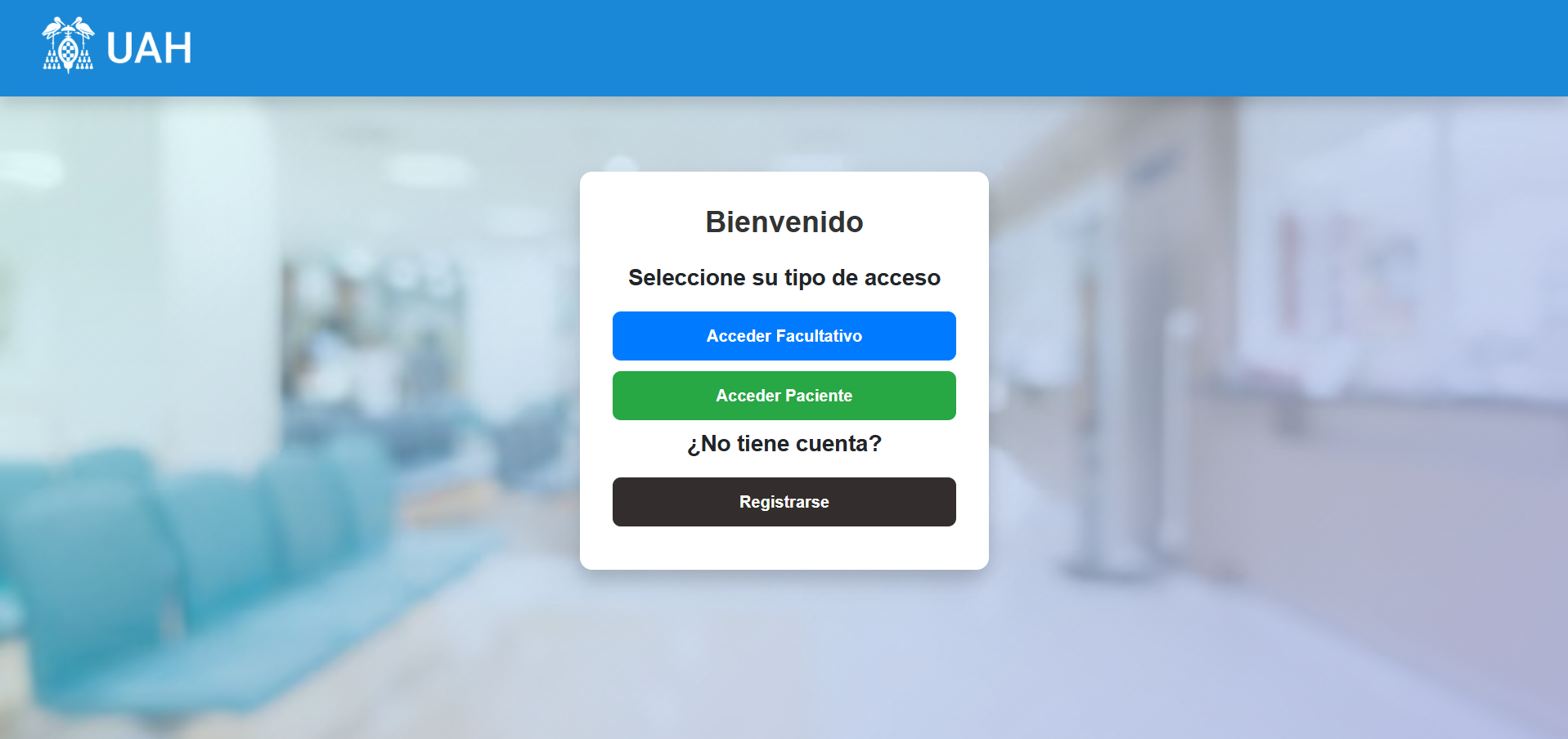


Figura 1 - Pantalla inicio de sesión

Sign-up.component muestra un formulario donde el facultativo debe registrarse, y llamando al método createSanitario() (Listado 9) implementado en el firebaseService, genera un campo en Firebase con el usuario y la contraseña introducidos, añadiendo el rol ‘no-role’ automáticamente.

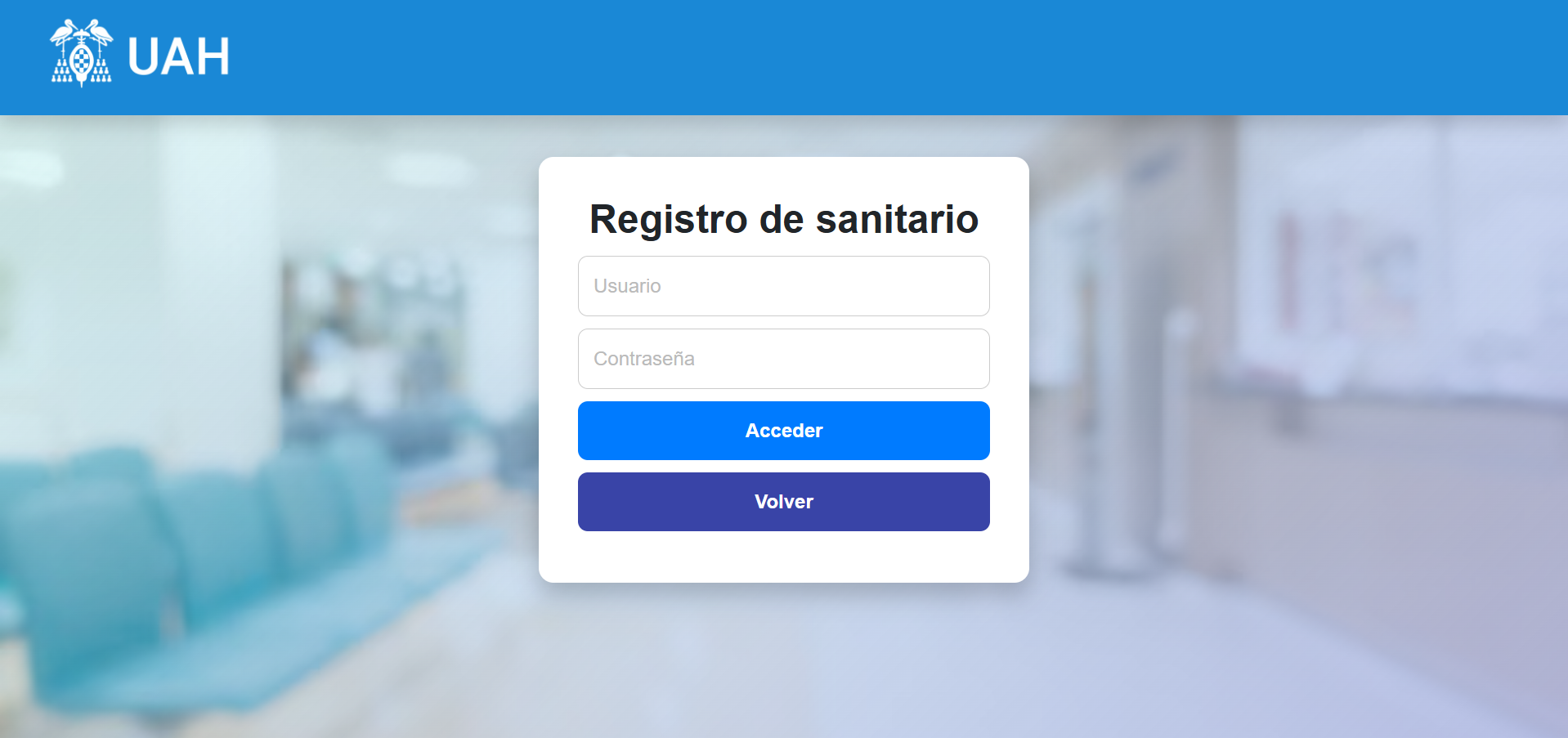


Figura 2 - Pantalla registro

Login-admin muestra un formulario donde se debe introducir un nombre de usuario y una contraseña. En este momento, se sigue la siguiente secuencia:

1. Se comprueba que el usuario existe en la base de datos
2. Se comprueba si su rol es válido (user o admin)
3. Se modifica el rol y el name del GlobalService
4. Se produce el inicio de sesión y se redirige a la pantalla del perfil de usuario.

En caso de que el usuario no exista o el rol no sea válido, se pinta un mensaje de error en la pantalla y no se inicia la sesión. Esto se obtiene llamando al método loginAdmin() (Listado 11) del firebaseService.

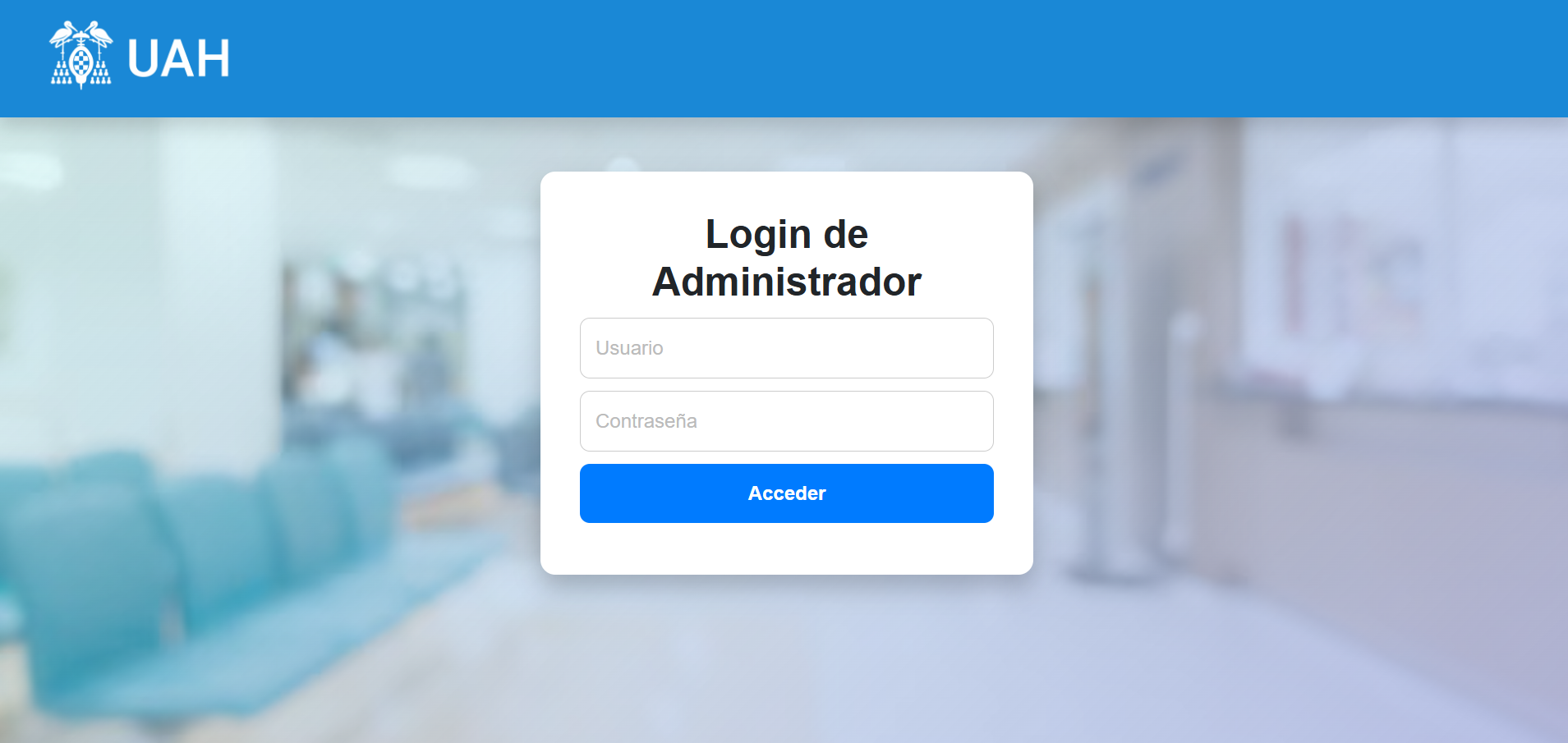


Figura 3 - Pantalla login admin

Login-patient.component muestra un formulario que pide al usuario introducir un apellido y un DNI, luego se sigue la siguiente secuencia:

1. Se comprueba que existe un usuario con ese DNI en la base de datos
2. Se comprueba si el DNI y el apellido introducido coinciden
3. Se produce el inicio de sesión y se redirige a la pantalla de datos de paciente.

En caso de que no exista el DNI o no coincida con el apellido, se muestra un mensaje de error por pantalla. Esta funcionalidad se ejecuta llamando al método loginAdmin() de firebaseService.



Figura 4 - Pantalla login paciente

### Perfil de usuario

Al iniciar sesión como facultativo o como administrador, se accede a la pantalla de perfil de usuario, que corresponde al componente:

* Dashboard *(Figura 6)*

Este componente presenta la información de usuario y da la bienvenida al sistema, indicando qué tipo de rol tiene el usuario y a qué tiene acceso dentro de la web.

En caso de que el usuario sea el administrador, se ve un mensaje en rojo que informa de cuántos usuarios sin rol asignado hay en el sistema, indicando al administrador que debe gestionar esos accesos.

Desde esta pantalla se puede acceder al resto de pantallas de la web, utilizando los botones de la columna vertical ubicada en el lado izquierdo de la pantalla, como se muestra en la figura 6. Si se pulsa el botón de salir, se cierra la sesión y se vuelve a la pantalla de inicio de sesión.



Figura 5 - Pantalla perfil de usuario

### Administración de facultativos

La administración de facultativos se encarga de gestionar los roles de los facultativos y de eliminarlos. Solo es accesible si el rol del usuario que ha iniciado la sesión es ‘admin’.

Para su implementación se crea el siguiente componente

* Administración *(Figura 7)*

Este componente muestra una tabla donde las filas son todos los facultativos registrados. En las columnas se muestra el nombre del usuario, su rol, y botones para cambiar el rol o eliminar la cuenta.

Esto se consigue llamando a los siguientes métodos, todos implementados en firestoreService:

* getUsuarios(): Para obtener la lista completa de facultativos registrados.
* hacerUser(): Para cambiar el rol a ‘user’
* hacerAdmin(): Para cambiar el rol a ‘admin’
* hacerNoRole(): Para cambiar el rol a ‘no-role’
* eliminarCuenta(): Para eliminar la cuenta de la base de datos

Además, este componente implementa un método que ordena las filas de la tabla pulsando en la cabecera de la columna ‘Nombre de usuario’ o ‘Rol’, ordenando así la tabla por nombre o por rol en orden alfabético. Esta función realiza una comparación entre las celdas de la columna indicada (por su índice n) y reorganiza las filas de forma ascendente o descendente, alternando la dirección si es necesario. (Listado 15)

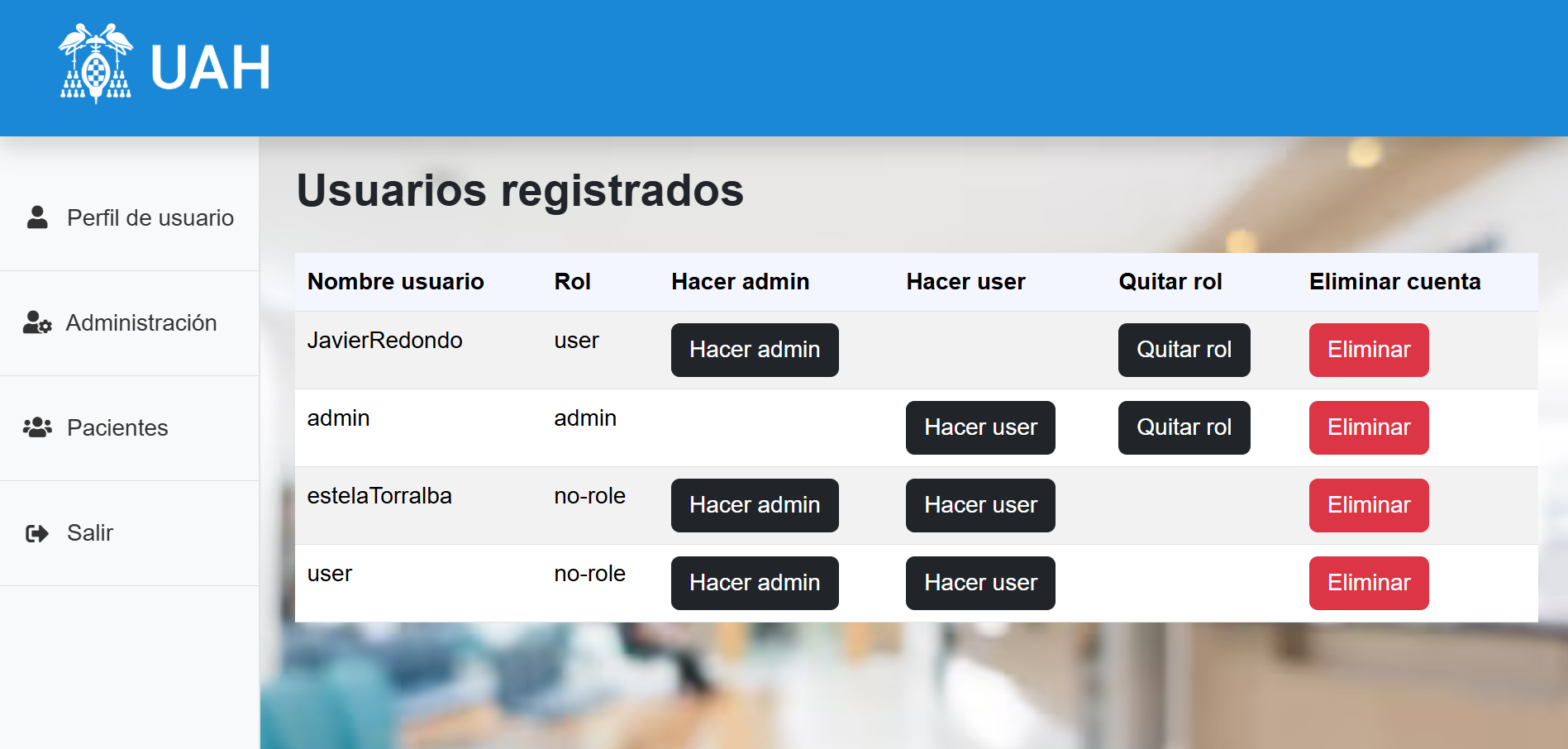


Figura 6 - Pantalla administración facultativos

Listado 15 - Método sortTable()

sortTable(n) {

    var table, rows, switching, i, x, y, shouldSwitch, dir, switchcount = 0;

    table = document.getElementById("myTable");

    switching = true;

    dir = "asc";

    while (switching) {

      switching = false;

      rows = table.rows;

      for (i = 1; i < (rows.length - 1); i++) {

        shouldSwitch = false;

        x = rows[i].getElementsByTagName("TD")[n];

        y = rows[i + 1].getElementsByTagName("TD")[n];

        if (dir == "asc") {

          if (x.innerHTML.toLowerCase() > y.innerHTML.toLowerCase()) {

            shouldSwitch = true;

            break;

          }

        } else if (dir == "desc") {

          if (x.innerHTML.toLowerCase() < y.innerHTML.toLowerCase()) {

            shouldSwitch = true;

            break;

          }

        }

      }

      if (shouldSwitch) {

        rows[i].parentNode.insertBefore(rows[i + 1], rows[i]);

        switching = true;

        switchcount++;

      } else {

        if (switchcount == 0 && dir == "asc") {

          dir = "desc";

          switching = true;

        }

      }

### Gestión de pacientes

El objetivo de la gestión de pacientes es poder visualizar y gestionar todos los pacientes que están registrados en la base de datos, pudiendo ver un listado con todos, y permitiendo editar sus datos personales, así como eliminar por completo el paciente y sus pruebas realizadas.

Para esta implementación se han creado los componentes:

* pacientes *(Figura 8)*
* editar-paciente *(Figura 9)*

#### Pacientes

El primer componente muestra un listado con todos los pacientes que han sido registrados en la base de datos (desde la app Android), y para cada uno se muestran tres botones:

* Pruebas: redirige a la pantalla de visualización de paciente
* Editar: redirige a la pantalla de editar paciente
* Eliminar: muestra un pop-up de confirmación y en caso afirmativo elimina al paciente de la base de datos

Para eliminar al paciente, al pulsar el botón y confirmar en el pop-up, se llama al método deletePaciente() (Listado 12) implementado en firestoreService.

Además, este componente añade un buscador de filtrado dinámico de pacientes a través de un campo de búsqueda, pudiendo buscar a un paciente por su número de historial o por su nombre. Esta funcionalidad se compone de dos métodos: filterList() y filter() (Listado 16)

El método filter() se activa al escribir en el input de búsqueda, enviando el valor introducido al observable searchTerm$. Por su parte, filterList() se suscribe a ese observable y aplica un filtro en tiempo real sobre la lista de pacientes (this.pacientes), devolviendo solo aquellos elementos cuyo nombre (ADname) o número de historial (ABnumHistorial) contienen el término introducido.

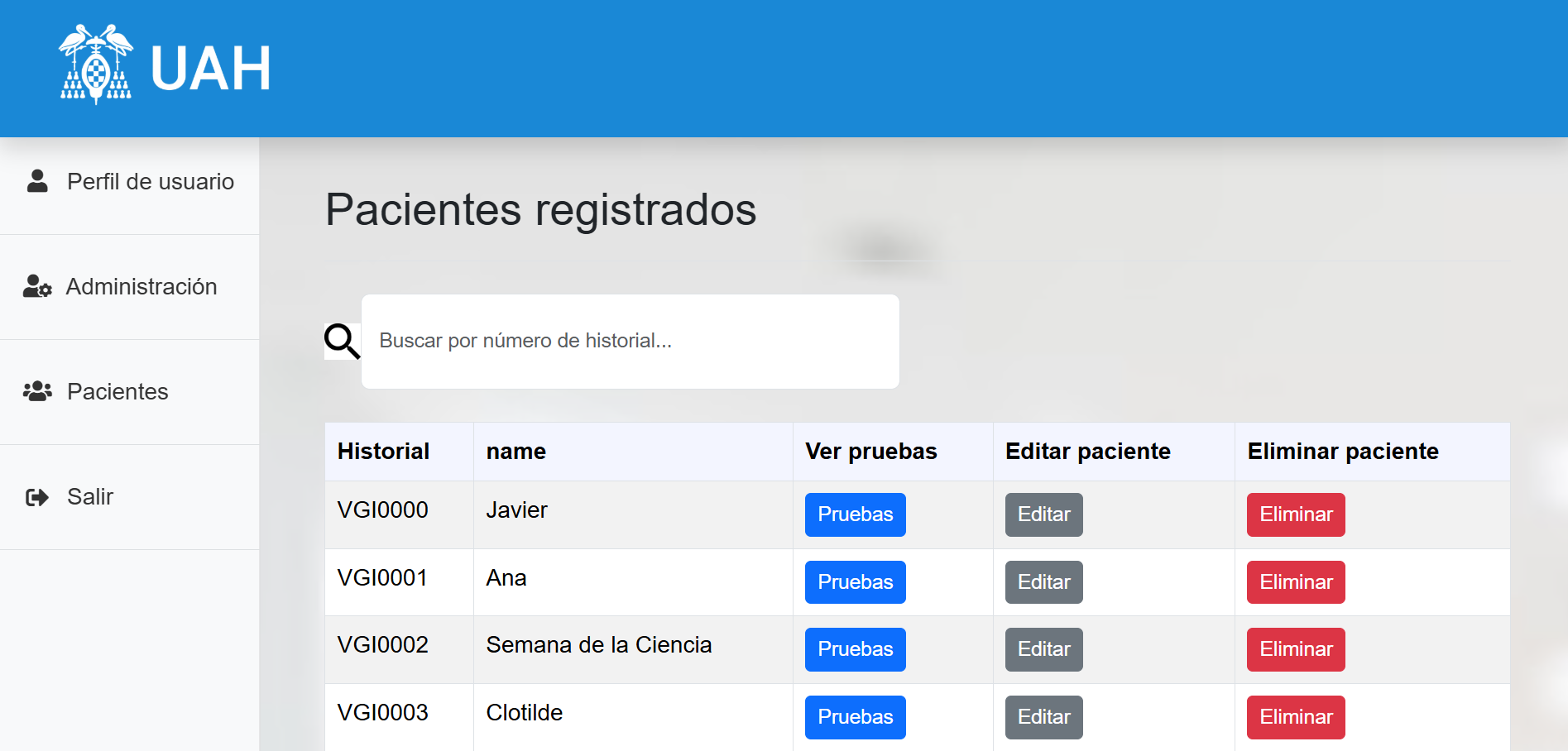


Figura 7 - Pantalla Listado pacientes

  filterList(): void {

    this.searchTerm$.subscribe(term => {

      const lowerTerm = term.toLowerCase();

      this.listFiltered = this.pacientes.filter(item =>

        item.data.ABnumHistorial.toLowerCase().includes(lowerTerm) ||

        item.data.ADname.toLowerCase().includes(lowerTerm)

      );

    });

  }

  filter($event) {

    this.searchTerm$.next($event.target.value)

  }

Listado 16 - Método filterList()

#### Editar paciente

Este componente muestra un formulario, donde se ven los datos personales asignados al paciente. Permite al administrador modificar los datos que quiera rellenando el campo correspondiente y pulsando en el botón ‘Guardar’.

Esto se consigue llamando al método updatePaciente(uidPaciente, this.checkoutForm.value) (Listado 13) implementado en firestoreService.

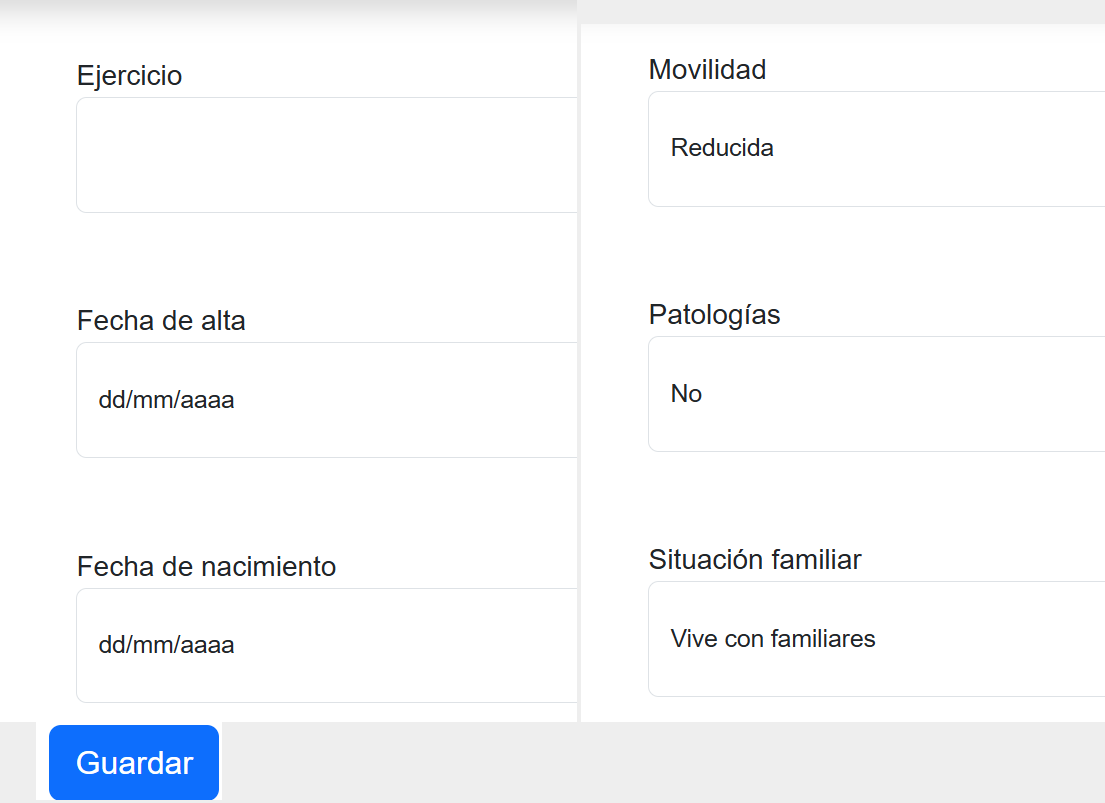


Figura 8 - Pantalla editar paciente

### Visualizar datos de paciente

El objetivo es poder visualizar los datos personales del paciente, ver un listado con las pruebas realizadas, poder acceder a cada una de las pruebas, ver el histórico de pruebas y poder descargar los datos en formato Excel.

Para esto, se han creado los siguientes componentes:

* Ver-paciente *(Figura 9)*
* equilibrio *(Figura 11)*
* Sentadillas *(Figura 12)*
* Velocidad *(Figura 13)*
* Histórico *(Figura 14)*

#### Ver paciente

Este componente, en primer lugar, muestra una lista con los datos personales del paciente. Esto es solo lectura, desde aquí no se puede modificar.

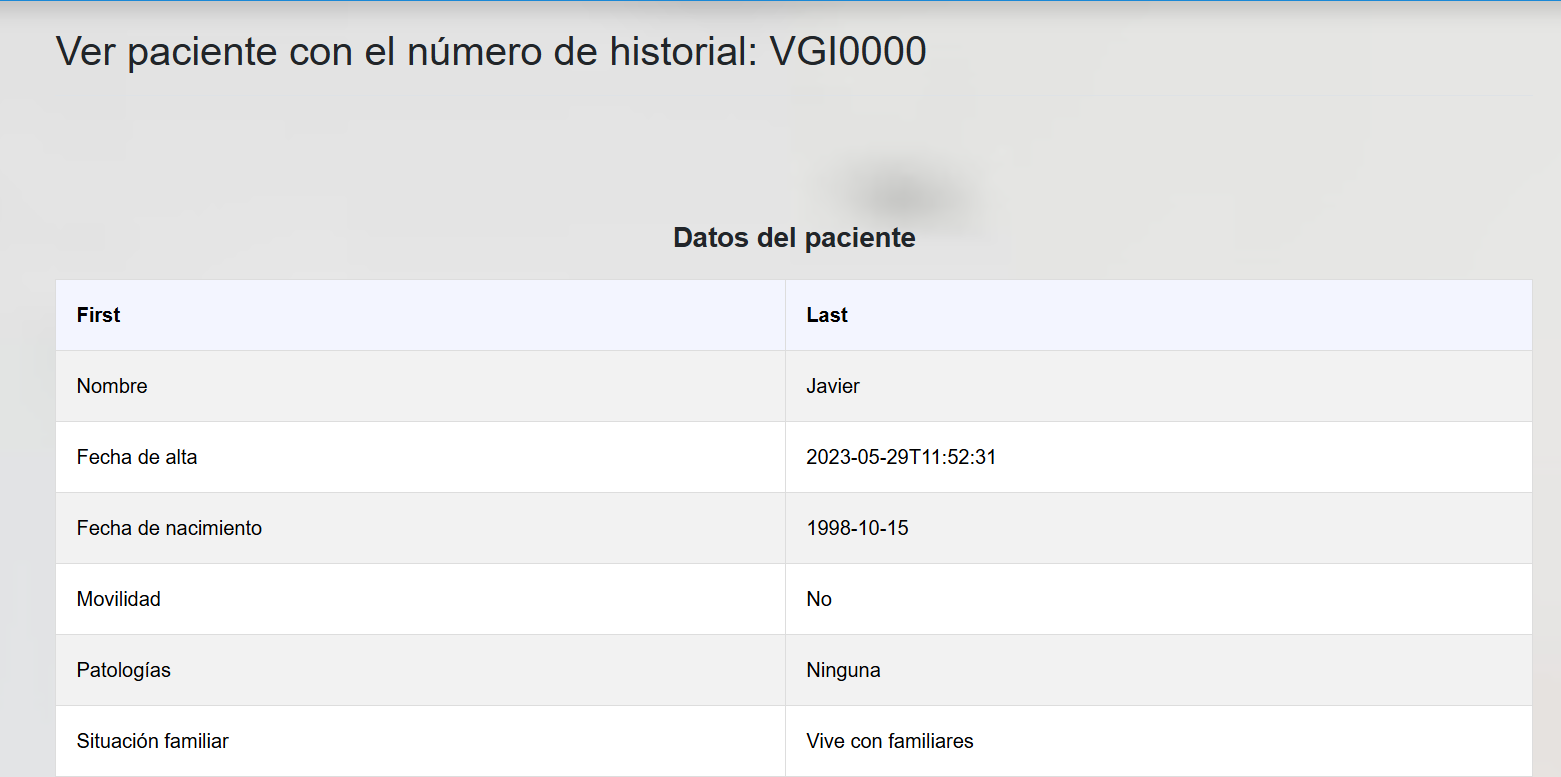


Figura 9 - Pantalla Datos paciente

Esto se consigue con el método getPaciente() (Listado 10) del servicio FirestoreService, que devuelve los datos de un paciente específico a partir de su identificador único, el cual ha sido previamente extraído desde la ruta del componente.

La función realiza una suscripción (subscribe) a los datos almacenados en Firebase y recorre el conjunto de resultados mediante un bucle forEach, construyendo un objeto que contiene tanto la clave (key) como los datos del paciente (val()). Estos datos se almacenan en el array this.paciente, que será utilizado posteriormente para mostrar la información del paciente en la vista HTML.

Después de los datos del paciente, el componente muestra un listado con las pruebas que este paciente tiene asignadas, de una manera similar a como se han obtenido los datos personales, pero esta vez llamando a los métodos getPacienteTCS(), getPacienteTBA() (Listado 5) y getPacienteTGS() de firestoreService.



Figura 10 - Pantalla pruebas realizadas

#### Equilibrio

Este componente muestra una sola prueba de equilibrio de un paciente concreto.

Se muestra la siguiente información:

* Fecha y hora
* Puntos paralelo
* Puntos semitandem
* Puntos Tandem
* Puntos Obtenidos
* Tiempo paralelo (s)
* Tiempo semitandem (s)
* Tiempo Tandem (s)

Esto se consigue de la misma forma que se obtuvo la tabla con el listado de las pruebas realizadas, pero esta vez, se filtra por fecha (dateTimeStartTest) dentro del componente, para recuperar exclusivamente los datos correspondientes a una sesión específica. Esto permite mostrar los resultados detallados de una única prueba de equilibrio en lugar de cargar un historial completo.

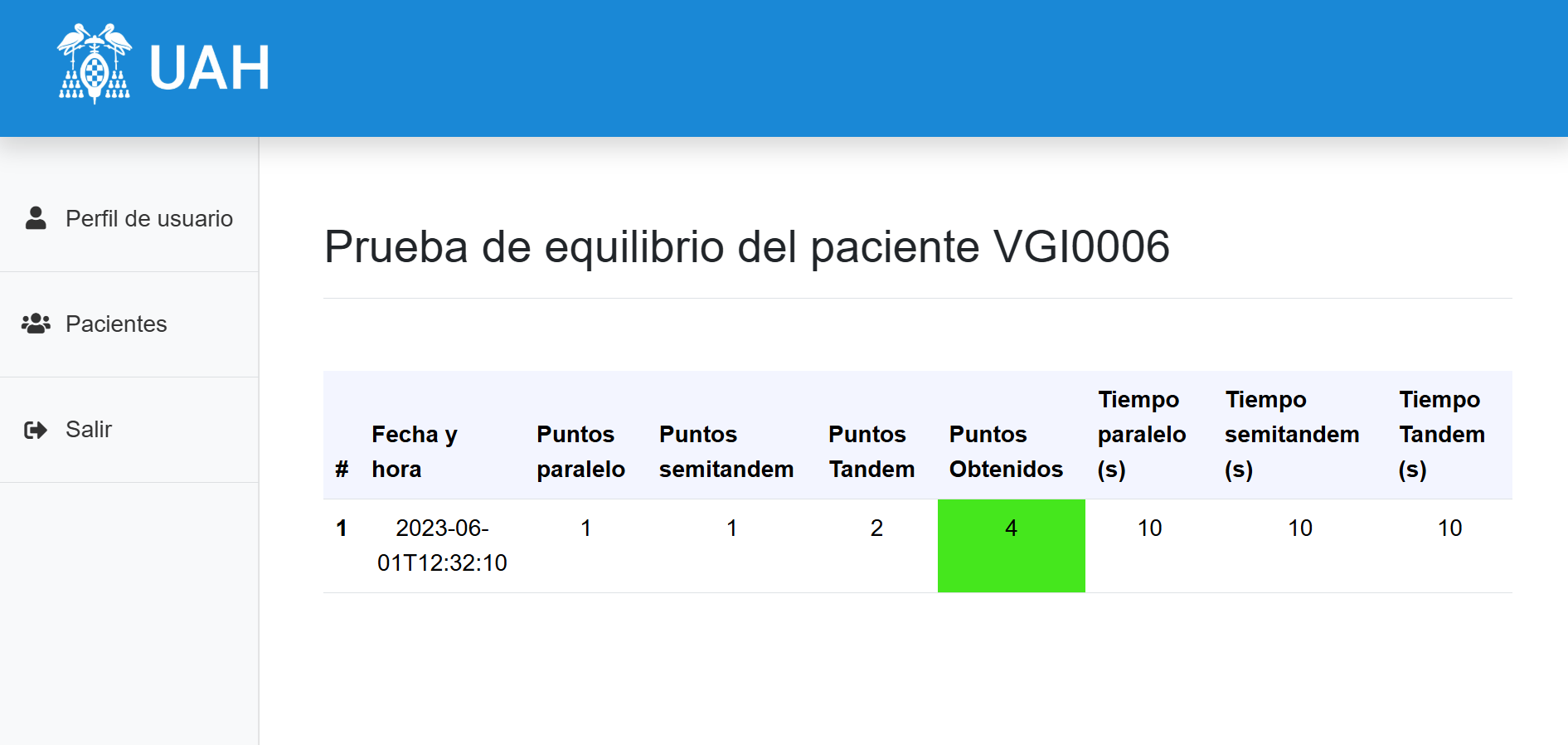


Figura 11 - Pantalla equilibrio

#### Sentadillas

Este componente, en primer lugar, al igual que en el componente equilibrio, muestra los siguientes datos de la prueba realizada:

* Fecha y hora
* Tiempo empleado (s)
* Número de sentadillas realizadas
* Puntos obtenidos

Además, muestra un gráfico donde se puede ver una representación de los datos del giróscopo obtenidos del sensor. Esto se consigue importando una librería llamada ‘ECharts’ y creando un objeto eChartsOpcion tal como se muestra en la figura X.

Este componente muestra también una tabla con los valores máximos, mínimos y la diferencia de cada una de las sentadillas realizadas. Para ello, se utiliza el método findPeaksAndValleys(data: number[]) que tiene como objetivo detectar picos (máximos locales) y valles (mínimos locales) dentro de un conjunto de datos numéricos, utilizando para ello una ventana deslizante.

Por último, se muestra un botón ‘Descargar Excel’ que llama al método exportarExcel() y permite generar y descargar un archivo Excel con los datos de las pruebas de sentadillas realizadas por un paciente. A partir de los registros cargados, se construye una tabla que incluye fechas, datos recogidos y valores estadísticos. Utiliza la librería xlsx para crear el archivo y saveAs para iniciar la descarga.



Figura 12 - Pantalla sentadillas



Figura 13 - Valores máximos y mínimos sentadillas

Listado 20 - Método exportarExcel()

  exportarExcel() {

    console.log(this.sentadillas);

    if (!this.sentadillas || this.sentadillas.length === 0) {

      console.warn("No hay datos de sentadillas para exportar.");

      return;

    }

    const datosParaExcel = this.sentadillas.map((item, index) => {

      return {

        ID: item.key,

        "Fecha de Inicio": item.data.dateTimeStartTest,

        "Fecha de Fin": item.data.dateTimeEndTest,

        "Datos Sentadillas": item.data.dataChart.join(", "),

        "Valores Máximos": this.maxValues.join(", "),

        "Valores Mínimos": this.minValues.join(", "),

        "Diferencia Máx - Mín": this.difMaxMinValues.join(", ")

      };

    });

    const ws: XLSX.WorkSheet = XLSX.utils.json\_to\_sheet(datosParaExcel);

    const wb: XLSX.WorkBook = XLSX.utils.book\_new();

    XLSX.utils.book\_append\_sheet(wb, ws, 'Datos Sentadillas');

    const excelBuffer: any = XLSX.write(wb, { bookType: 'xlsx', type: 'array' });

    const data: Blob = new Blob([excelBuffer], { type: 'application/octet-stream' });

    saveAs(data, `Datos\_Sentadillas.xlsx`);

  }

Listado 21 - Método findPeaksAndValley()

  findPeaksAndValleys(data: number[]): { maxValues: number[], minValues: number[] } {

    console.log(data);

    const peaks: number[] = [];

    const valleys: number[] = [];

    const windowSize = 30;

    data = data.map(value => Math.round(Number(value) \* 100) / 100);

    console.log("Datos procesados:", data);

    // Buscar el primer índice con valor >= 30

    let startIndex = -1;

    for (let i = 0; i < data.length; i++) {

      if (data[i] >= 30) {

        startIndex = i;

        break;

    }

    if (startIndex === -1) {

      console.warn("No hay valores mayores o iguales a 30 en el array.");

      return { maxValues: [], minValues: [] };

    let expectingPeak = true;  // Comenzamos buscando un pico (máximo)

    for (let i = startIndex; i < data.length - windowSize; i++) {

      const current = data[i];

      let isPeak = true;

      let isValley = true;

      if (expectingPeak) {

        for (let j = 1; j <= windowSize; j++) {

          if (data[i + j] >= current) {

            isPeak = false;

          }

        }

        if (isPeak) {

          peaks.push(current);

          expectingPeak = false; // Después de un pico, buscamos un valle

        }

      }

      if (!expectingPeak) {

        for (let j = 1; j <= windowSize; j++) {

          if (data[i + j] <= current) {

            isValley = false;

          }

        }

        if (isValley) {

          valleys.push(current);

          expectingPeak = true; // Después de un valle, buscamos un pico

    return {

      maxValues: peaks.slice(0, 5),

      minValues: valleys.slice(0, 4),

this.chartOption = {

        title: {

          text: 'Gráfico'

        },

        tooltip: {

          trigger: 'axis'

        },

        legend: {

          data: ['Sentadillas']

        },

        grid: {

          left: '3%',

          right: '4%',

          bottom: '3%',

          containLabel: true

        },

        toolbox: {

          feature: {

            saveAsImage: {}

          }

        },

        xAxis: {

          type: 'category',

          boundaryGap: false,

          data: this.x\_axis,

        },

        yAxis: {

          type: 'value'

        },

        series: [

          {

            name: 'Sentadillas',

            type: 'line',

            data: this.sentadillas[0].data.dataChart

          },

        ]

      };

#### Velocidad

Este componente sigue la misma estructura que le componente ‘Sentadillas’, mostrando, en este caso, los siguientes datos:

* Fecha y hora
* Distancia a recorrer (m)
* Puntos obtenidos
* Tiempo empleado (s)
* Velocidad media (m/s)

Muestra, al igual que el anterior componente, un botón para exportar los datos en Excel y un gráfico que, en este caso, muestra la distancia y la velocidad en cada muestra tomada por el sensor, tanto de la ida como de la vuelta.



Figura 14 - Pantalla velocidad

#### Histórico

Este componente tiene como objetivo unificar todas las pruebas de un paciente en una misma pantalla, facilitando así la interpretación de los datos por parte del facultativo.

Muestra tablas para cada tipo de prueba, indicando para cada prueba realizada el tiempo empleado y los puntos obtenidos. Desde este punto, pulsando en la fecha de cada prueba, se redirige a la pestaña donde se ve la prueba en detalle (componentes ‘Sentadillas’, ‘Equilibrio’ y ‘Velocidad’) (Figura 16)

Además, muestra un gráfico donde se ven los puntos obtenidos en cada prueba, representando una línea por cada tipo de prueba, como se muestra en la figura 15.

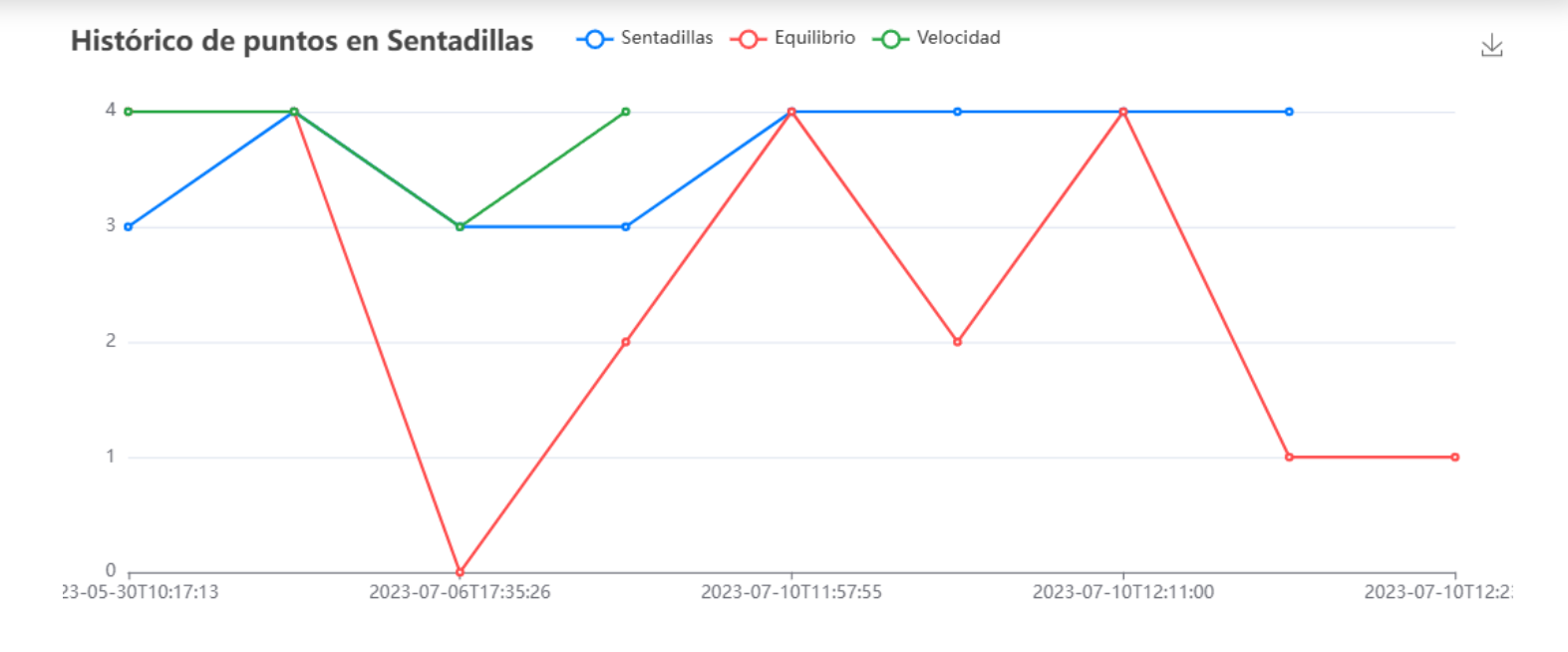


Figura 15 - Gráfico histórico



Figura 16 - Datos de pruebas histórico