Capítulo 5 – Desarrollo e implementación

En este capítulo

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.

Android

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

* Bluetotth: Se establecía al 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.

Firebase

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.

Base de datos

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.
* Angular CLI (Command Line Interface): Se intala a través de npm y es una herramienta oficial para habilitar los comandos de Angular por terminal, permitiendo la creación y configuración dl proyecto.

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.

A partir de este momento ya se puede ejecutar la aplicación, con el comando ‘ng serve’ que abre un puerto del PC en el host local y permite visualizar los cambios que hagamos en el código de manera dinámica.

Como se ha visto en el capítulo del estudio teórico, Angular funciona en base a componentes de este tipo, creando los necesarios y comunicándolos entre sí. En las siguientes secciones se describen otros ficheros especiales que son necesarios para este proyecto, y a los que se hará referencia cuando se hable de la implementación.

Módulos (app.module.ts)

Módulos (.module.ts)

Los módulos en Angular son elementos fundamentales que permiten organizar el código de forma estructurada y escalable. Un módulo agrupa componentes, directivas, pipes y servicios relacionados, y los declara en un único archivo para su uso conjunto. El módulo que se usa en este proyecto es ‘app.module.ts’, el cual sirve como punto de entrada del sistema, importa todos los componentes y los declara. También importa otros módulos servidos por Angular o librerías externas que se usan en la aplicación.

Enrutamiento (app-routing.module.ts)

Angular utiliza un sistema de enrutamiento basado en rutas que permite navegar entre diferentes vistas o componentes sin recargar la página. El archivo app-routing.module.ts define qué componente debe mostrarse cuando el usuario accede a una URL concreta. Este sistema permite estructurar la aplicación como una SPA (Single Page Application), facilitando una navegación fluida y moderna, a través de botones, entre secciones como inicio de sesión, listado de pacientes o visualización de pruebas.

3. Servicios (.service.ts)

Los servicios en Angular se encargan de gestionar la lógica de negocio, realizar operaciones de datos y facilitar la comunicación con fuentes externas, como, en este caso, la comunicación con Firebase. Son clases reutilizables que se inyectan en los componentes mediante el mecanismo nativo en Angular de inyección de dependencias.

5. Archivo de configuración (angular.json)

El archivo angular.json es un archivo de configuración global del proyecto Angular. Define cómo se compila la aplicación, cómo se estructuran los archivos, qué estilos y scripts se deben incluir y otras opciones del entorno de desarrollo. Este archivo es fundamental para adaptar el proyecto a necesidades específicas, como personalizar rutas de salida, añadir temas, o configurar arquitecturas específicas.

🌍 6. Entornos (environment.ts)

Angular permite definir variables y configuraciones específicas para distintos entornos (desarrollo, producción, etc.) mediante el archivo environment.ts. Esto resulta especialmente útil para proyectos que trabajan con servicios externos como Firebase, ya que permite configurar las claves y URLs correspondientes sin tener que modificarlas manualmente entre entornos. Además, mejora la seguridad y evita errores al desplegar la aplicación.

📦 7. Gestor de dependencias (package.json)

El archivo package.json es fundamental en cualquier proyecto Angular. Define todas las dependencias, scripts y metadatos del proyecto. Incluye información sobre las versiones de Angular, TypeScript, Firebase, y cualquier otra librería utilizada. Además, especifica los scripts para compilar, testear o desplegar la aplicación. Este archivo permite que el proyecto pueda ser clonado y configurado fácilmente en otros entornos mediante el comando npm install.

Estructura del proyecto:

El proyecto desarrollado en Angular presenta una estructura modular, limpia y organizada, pensada para favorecer la mantenibilidad y escalabilidad del sistema. A continuación, se describen las carpetas y archivos más relevantes dentro del directorio principal /src/app, que es donde se concentra la lógica de la aplicación.

components/

Este directorio contiene todos los componentes de la aplicación, agrupados por funcionalidades o pantallas. Cada subcarpeta representa un componente asociado a una vista o módulo específico del sistema, como por ejemplo dashboard, login-admin, pacientes, ver-paciente o equilibrio. Esta separación permite mantener el código segmentado y fácilmente localizable. Cada componente incluye sus propios archivos .ts, .html y .scss.

📁 services/firestore/

Aquí se encuentran los servicios de la aplicación, principalmente dos:

* firestore.service.ts que gestiona la lógica de conexión con Firebase, como, donde se definen las funciones necesarias para leer, escribir o actualizar datos en la base de datos en la nube.
* app.service.ts: donde se gestionan funcionalidades lógicas comunes en toda la aplicación, o variables que se pretende que sean accesibles desde todos los componentes

📄 app-routing.module.ts

Este archivo contiene la configuración de las rutas del sistema, es decir, define qué componente debe cargarse en función de la URL. Es esencial en una arquitectura SPA (Single Page Application), ya que permite la navegación sin recargar la página.

📄 app.module.ts

Es el módulo principal de la aplicación. En él se declaran todos los componentes, servicios, módulos externos e internos que forman parte del sistema. Sirve como punto de entrada de la aplicación y es cargado al iniciar la misma.

📁 assets/ y otros archivos

La carpeta assets se utiliza para almacenar archivos estáticos como imágenes o iconos. Además, archivos como environment.ts, main.ts, index.html o styles.scss definen configuraciones globales, estilos compartidos y el punto de arranque de la aplicación.

Esta organización modular facilita la escalabilidad del proyecto, ya que permite añadir nuevas funcionalidades sin interferir en otras partes del sistema, y mejora la comprensión del código por parte de otros desarrolladores o evaluadores.

Vinculación con Firebase

En primer lugar, es necesario instalar los paquetes necesarios desde el Angular CLI, con el comando

Ilustración 1- ff

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 4.50*).

Listado 4.50: Configuración archivo enviroment.ts

export const environment = {

  production: false,

  firebase: {

    apiKey: "AIzaSyCsmqP1VBBnYrYUhNVMFhVI3az\_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:db9439f911b0b63fe553b0",

    measurementId: "G-H7SG49SZGH"

  },

};

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

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 (firestore.service.ts, auth.service.ts)

Se crean 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.

En el directorio src/services/firestore.service.ts es donde se implementan estos métodos.

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.

En este servicio se incluyen funciones específicas para:

* Consultar y actualizar los datos de pacientes.
* Obtener los datos de las pruebas de sentadillas, velocidad y equilibrio
* Registrar y eliminar usuarios del tipo facultativo.
* Modificar los roles de los usuarios registrados.
* Eliminar un paciente junto con todas sus pruebas asociadas.

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 X) 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, reutilizable y mantenible**, separando claramente la lógica de negocio de las operaciones sobre la base de datos.

  public getPacienteTBA(numHistorial: string) {

    console.log('getPacientesTBA: ' + numHistorial)

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

  }

  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'

    });

  }

  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, });

      }

    });

  }

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)

GlobalService

Se ha implementado un servicio que será clave en toda la web denominado 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.

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() o updateName(). 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.

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
* Sing-up
* Login-admin
* Login-patient

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.

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

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() (Figura X) del authService.

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.

Administración de usuarios

La administración de usuarios se encarga de gestionar los roles de los usuarios (no pacientes) y de eliminarlos. Dolo es accesible si el rol del usuario que ha iniciado la sesión es ‘admin’.

Para su implementación se crea el siguiente componente

* Administración

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

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

* getUsuarios(): Para obtener la lista completa de usuarios 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 cada columna. 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. (Figura X)

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 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 al usuario y sus pruebas realizadas.

Para esta implementación se han creado los componentes:

* pacientes
* editar-paciente

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:

* Ver pruebas: redirige a la pantalla de visualización de paciente
* Editar paciente: redirige a la pantalla de editar paciente
* Eliminar paciente: 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() (Figura X) implementado en firestoreService.

Además, este componente añade un buscador de **filtrado dinámico de pacientes** a través de un campo de búsqueda. Esta funcionalidad se compone de dos métodos: filterList() y filter() (Figura X)

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.

  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)

  }

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) (Figura X) implementado en firestoreService.

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
* equilibrio
* Sentadillas
* Velocidad
* Histórico

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.

Esto se consigue con el código que se muestra en la figura X, que llama al método getPaciente() del servicio FirestoreService para **obtener 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**.

    this.firestoreService.getPaciente(this.parametroIdLargo.parametroIdLargo).subscribe((pacientesSnapshot) => {

      this.paciente = [];

      pacientesSnapshot.forEach((pacienteData: any) => {

        this.paciente.push({

          key: pacienteData.payload.key,

          data: pacienteData.payload.val()

        });

      })

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() y getPacienteTGS() de firestoreService.

      this.firestoreService.getPacienteTBA(this.parametroIdCorto.parametroIdCorto).subscribe((equilibrioSnapshot) => {

        this.equilibrio = [];

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

          this.equilibrio.push({

            key: equilibrioData.payload.key,

            data: equilibrioData.payload.val(),

            tipo: 'Equilibrio'

          })

          console.log('Equilibrio: ' + equilibrioData.payload.key)

        })

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.

    this.firestoreService.getPacienteTBA(this.usuario.idCorto).subscribe((equilibrioSnapshot) => {

      this.equilibrio = [];

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

        if (this.fecha.fecha == equilibrioData.payload.val().dateTimeStartTest) {

          this.equilibrio.push({

            key: equilibrioData.payload.key,

            data: equilibrioData.payload.val()

          })

          console.log('Sentadilla: ' + equilibrioData.payload.key)

        }

      })

    })