

# MARCO PRÁCTICO

## 1. ESTRUCTURA DEL PROYECTO

Para mantener el código ordenado y separado hemos seguido una estructura principal que separe en diferentes directorios las partes principales de la aplicación.

De esta forma, se consigue mantener una organización clara y eficiente del código, facilitando tanto su mantenimiento como su escalabilidad.

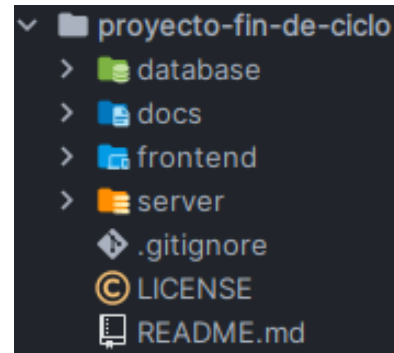


Fig 1 Estructura general del proyecto.

### A. Directorio *database*

En este directorio se almacenan todos los ficheros que están relacionados con la base de datos: diagramas, modelos, scripts SQL, etc.

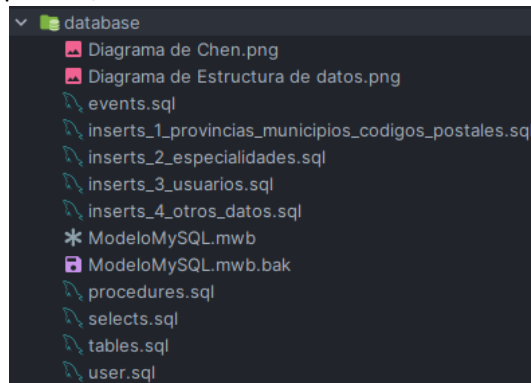


Fig 2 Estructura del directorio 'database'.

### B. Directorio *docs*

Este es el directorio de documentación donde hemos decidido guardar todos los archivos que sirven como documentación de este proyecto.

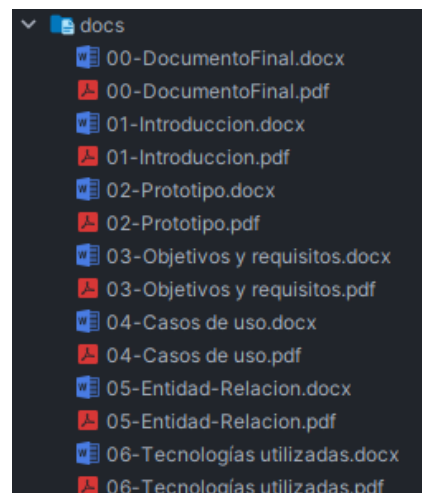


Fig 3 Estructura directorio 'docs'.

### C. Directorio *frontend*

Es el directorio donde se almacena todo aquello que tenga que ver con la programación orientada al cliente web.

La estructura elegida es una de las que se recomiendan para Angular basada en la organización de directorios por funciones. Cada carpeta tiene una responsabilidad específica y contiene los ficheros relacionados a esa funcionalidad para mantener el código modularizado.

En **src** encontramos los ficheros principales del proyecto de Angular y que son necesarios para su funcionamiento como el archivo HTML para el index, los estilos principales o el archivo del servidor sobre el que se ejecuta el proyecto (**main.ts**). De este directorio parten dos subdirectorios:

- a) **app**: En ella se almacenan todos los archivos principales de una aplicación Angular como los ficheros de plantilla, funcionamiento y estilado del componente principal de la aplicación (**app-root**) o los archivos de configuración y enrutado.

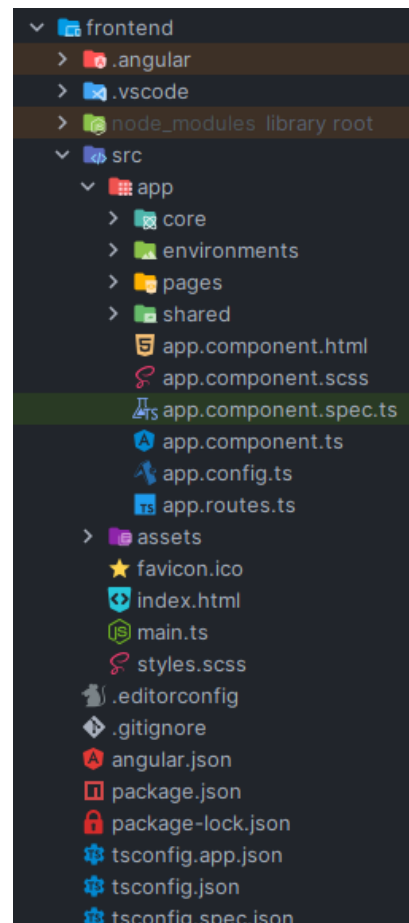


Fig 4 Estructura del directorio 'frontend'.

Este directorio queda subdividido de la siguiente forma:

1. **core**: Contiene los servicios y utilidades que son esenciales para el funcionamiento de la aplicación y que son utilizados en diferentes partes de esta. Dentro de este directorio encontramos otros subdirectorios encargados de funciones específicas: **classes**, contiene todas las clases que son utilizadas en múltiples puntos (p. e. una clase que contenga múltiples validadores personalizados para la entrada de datos

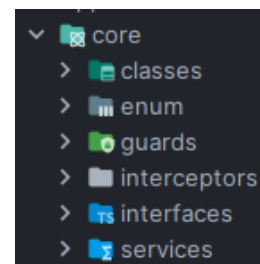


Fig 5 Subdirectorios del directorio 'core'.

en los formularios); **enum**, contiene enumerados que son utilizados en diferentes partes de la aplicación; **guards**, en este directorio se almacenan todos los guardias de ruta que, como veremos más adelante, son la forma que tiene Angular de controlar el acceso a rutas específicas; **interceptors**, en ella se contienen los interceptores HTTP de Angular que sirven para manejar las solicitudes y respuestas HTTP de forma global y que veremos en detalle más adelante; **interfaces**, en este

directorio se guardan las diferentes interfaces de TypeScript que definen las estructuras de los datos utilizados en la aplicación y, por último, **services** que almacena los servicios que contienen la lógica de negocio y que son utilizados para compartir datos entre componentes.

2. **enviroments**: Contiene los archivos de configuración de entorno para diferentes configuraciones de despliegue.
3. **pages**: En este directorio se organizan los diferentes módulos de la aplicación, cada uno correspondiente a una página o sección específica pudiéndose subdividir en secciones de una mayor especificidad. Dentro de estos subdirectorios se contienen los componentes relacionados a esa página.
4. **shared**: Contiene recursos reutilizables en toda la aplicación y se subdivide en **components** que contiene los componentes que son reutilizados como la navbar, la sidebar, los inputs específicos para contraseñas, etc. y en **pipes** que almacena las tuberías personalizadas y que se encargan de transformar los datos en las plantillas de Angular.

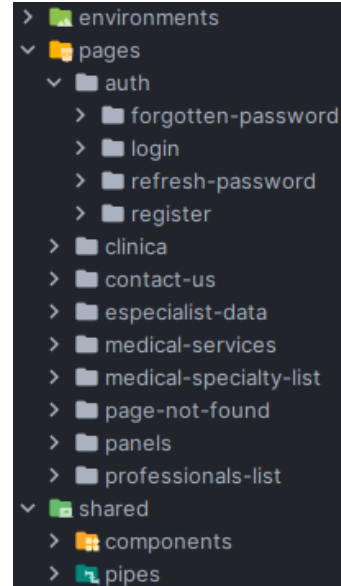


Fig 6 Subdirectorios 'enviroments', 'pages' y 'shared'.

- b) **assests**: En este directorio se almacenan los recursos estáticos de la aplicación tales como imágenes, iconos, etc.

## D. Directorio *server*

Es el directorio donde se almacena todo lo que tiene que ver con el trabajo de *backend*.

En la raíz de este directorio podemos ver los archivos **app.js** que es el archivo principal de la aplicación que inicia y configura el servidor y **.env** que es el archivo de configuración de entorno que contiene variables de entorno sensibles, como claves codificación de token, de contraseñas, configuraciones de base de datos, etc.

Este directorio principal se subdivide en diferentes subdirectorios:

- a) **controllers**: En esta carpeta se almacenan los controlares que en una aplicación Node.JS son los responsables de

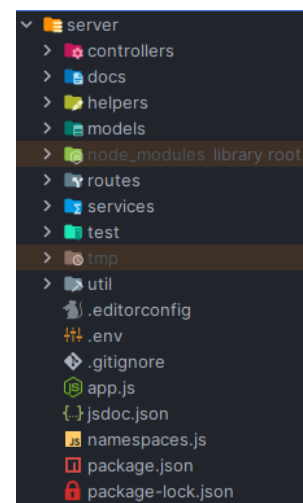
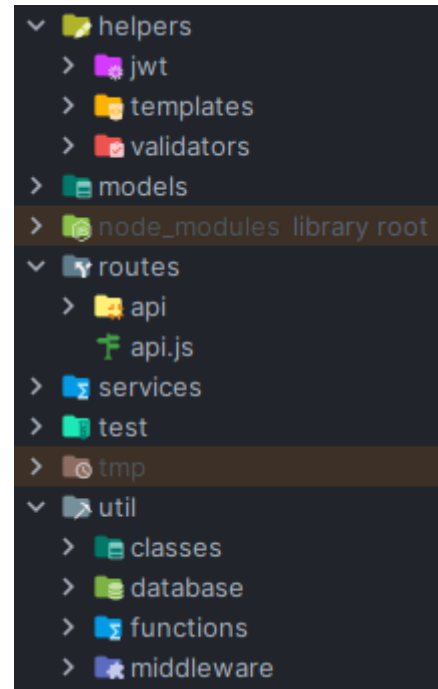


Fig 7 Estructura del directorio 'server'.

manejar las solicitudes entrantes, procesar los datos (generalmente cediéndoselos al servicio principal asociado a él) y devolver la respuesta correspondiente al cliente.

b) **docs**: Es el directorio que utiliza el servidor para la documentación automática generada con JSDoc y Swagger, dentro de él se describe el funcionamiento del sistema, el funcionamiento de las diferentes rutas, etc.

c) **helpers**: Contiene las utilidades y funciones auxiliares que apoyan la lógica principal del servidor. Dentro de este directorio encontramos 3 subdirectorios: **jwt** que contiene todo lo relacionado con JSON Web Tokens como la generación y verificación de tokens, **templates** que almacena las plantillas handlebars utilizadas para la generación de PDFs y correos electrónicos y **validators** que contiene los diferentes validadores utilizados para verificar y asegurar que los datos de entrada cumplen con los criterios necesarios antes de ser procesados.



d) **models**: En esta carpeta se almacenan los modelos de datos que representan la forma que tiene el servidor Node de comunicarse con el servidor de base de datos.

Fig 8 Estructura de los subdirectorios 'helpers', 'routes' y 'util'.

e) **routes**: Contiene los archivos relacionados con las rutas del servidor, definiendo cómo las solicitudes HTTP se asignan a los controladores. Se ha generado un subdirectorio específico api que define los puntos de entrada de la API.

f) **services**: Es el directorio para los servicios que son los elementos de Node.JS encargados de encapsular la lógica de negocio del servidor llevando a cabo la interacción con base de datos (a través de los modelos), la interacción servicios auxiliares, con funciones de utilidad o de ayuda, etc.

g) **test**: Contiene los archivos para las pruebas del servidor.

h) **tmp**: En este directorio se almacenan los archivos temporales, como se verá en la sección dedicada a la generación de PDFs este tipo de ficheros son creados de forma temporal en función de las solicitudes del usuario y una vez servida la respuesta de descarga, son eliminados del servidor.

i) **util**: En esta carpeta se almacenan las utilidades y funciones utilizadas a lo largo del proyecto y se subdividen en **classes** que contiene clases que encapsulas lógica

## MARCO PRÁCTICO. Estructura del proyecto

reutilizable, **database** que incluye las funciones y configuraciones específicas para la interacción con la base de datos, **functions** que contiene funciones auxiliares que son utilizadas en varias partes del proyecto y **middleware** que incluye middleware que se ejecuta en el ciclo de vida de las solicitudes HTTP para realizar tareas como la autenticación.

## 2. AUTENTICACIÓN, AUTORIZACIÓN Y CONTROL DE ACCESO

Debido a que nuestro objetivo es crear una aplicación médica funcional, es fundamental mantener un control de acceso riguroso a los datos de la aplicación garantizando que, por ejemplo, sólo los especialistas y el paciente en concreto pueda acceder a los datos médicos concretos impidiendo que el personal administrativo u otro paciente pueda acceder a ellos.

### A. Inicio de sesión

#### 1) Pantalla de login

Para comenzar a utilizar la aplicación de MediAPP lo primero que debe realizar un usuario es iniciar sesión con su cuenta previamente registrada bien por sí mismo (paciente) o por un administrador (administrador principal y especialistas), en esta información debe incluir su correo electrónico y su contraseña (este campo cuenta con la funcionalidad que permite mostrar y ocultar los caracteres).

El formulario de login tiene un fondo azul claro. En la parte superior, el título "LOGIN" está en negrita. Debajo, el campo "Email" es un input blanco. El campo "Contraseña" es un input blanco con un icono de ojo a la derecha para alternar la visibilidad. Debajo de la contraseña, hay un enlace azul "He olvidado mi contraseña". En el centro, hay un botón gris "Iniciar sesión". En la parte inferior, hay un enlace azul "¿No tienes cuenta?" seguido de un botón azul "Regístrate".

Fig 9 Formulario de login.

#### 2) Validación de datos en frontend

La validación de los datos en el lado del cliente se realiza en tiempo real, de forma que si los datos que se están introduciendo no son válidos se le comunica al usuario inmediatamente para tener un *feedback* más directo.


Este formulario es idéntico al de la Fig 9, pero muestra un mensaje de error en rojo: "El email ingresado no tiene un formato válido." justo debajo del campo de email, que contiene el texto "usuario@dominio.". El resto del formulario, incluyendo los campos de contraseña, los botones y los enlaces, permanece igual.

Fig 10 Error en la introducción del email.

Esto se consigue gracias a la posibilidad de crear formularios reactivos en Angular y al uso de validadores personalizados que comprueban, por ejemplo, a través de expresiones regulares que el contenido de los inputs cumple con ciertas normas.

```
initForm(): void {
  this.loginForm = new FormGroup<any>({ controls: {
    'email': new FormControl(
      value: null,
      validatorOrOpts: [
        Validators.required,
        CustomValidators.validEmail
      ]
    ),
    'password': new FormControl(
      value: null,
      validatorOrOpts: [
        Validators.required,
      ]
    )
  }});
}
```

Fig 11 Generación del formulario reactivo de Angular para el login.

```
static validEmail(control: FormControl): { [s: string]: boolean } | null {
  const value = control.value;
  const regex: RegExp = new RegExp( pattern: '^[\\w.%+-]+@[a-z\\d]+(\\.[a-zA-Z]{2,})+$' );

  if (!regex.test(value)) {
    return { 'isInvalidMail': true }
  }

  return null;
}
```

Fig 12 Ejemplo de validador: Validador para emails.

Mientras el formulario no sea válido el botón de inicio de sesión permanece deshabilitado, una vez completados ambos campos se habilita y es posible hacer un intento de inicio de sesión el cual pasa por una segunda validación por parte del componente que comprueba que si el formulario es invalido no realice ninguna acción y no llame al servicio encargado de comunicarse con la API para conseguir la autenticación.

### 3) Llamada al servicio de autenticación

Si todo es correcto se produce una llamada al servicio de autenticación, concretamente al método login que es el que realizará una petición HTTP a la API y generará un observable que contendrá la respuesta generada por el servidor.

```
onLoginAttempt(): void {
  if (this.loginForm.invalid) {
    return;
  }

  this.isLoading = true;

  const email = this.loginForm.get('email').value;
  const pass = this.loginForm.get('password').value;

  this.authService.login(email, pass)
    .subscribe( observerOrNext: {
      next: (response): void => {
        this.isLoading = false;
        this.loginForm.reset();
        this.router.navigate( commands: ['/mediapp'])
          .then((r: boolean): void => {});
      },
      error: (error: HttpErrorResponse): void => {
        this.error = error.error.errors[0]
          .toString()
          .replace(/Error: /g, '');
        this.isLoading = false;
      }
    });
}
```

Fig 13 Método encargado de gestionar el login en el cliente.

```
login(email: string, password: string): Observable<any> {  
  return this.http.post<any>({url: `${this.apiUrl}/usuario/login`, body: {  
    email: email,  
    password: password  
  }});  
}
```

Fig 14 Método encargado de comunicarse con el backend y llevar a cabo el login.

#### 4) Activación del interceptor para el login

Esta petición HTTP dispara un interceptor, concretamente del LoginInterceptor.

```
export class LoginInterceptor implements HttpInterceptor {  
  
  constructor(private authService: AuthService) {}  
  
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
    return next.handle(req).pipe(  
      tap((event: HttpEvent<any>) => {  
        if (event instanceof HttpResponse && req.url.endsWith('/usuario/login')) {  
          const accessToken = event.body.access_token;  
          const refreshToken = event.body.refresh_token;  
  
          this.authService.storeAccessToken(accessToken);  
          this.authService.storeRefreshToken(refreshToken);  
  
          this.authService.loggedInUser.next({ value: true });  
        }  
      }),  
      catchError((error: any) => {  
        return throwError(() => error);  
      })  
    );  
  }  
}
```

Fig 15 Interceptor para el login del usuario.

Este interceptor se encargará de permanecer a la escucha de la resolución del servidor y decidirá lo que se debe hacer con esta, si es positiva almacenará la respuesta del servidor en el LocalStorage del navegador, en cualquier otro caso devolverá el error recibido.

Completado esto, el servicio de autenticación completará la petición POST al servidor para intentar el inicio de sesión guardando en el cuerpo de la solicitud dos campos: email y password.

#### 5) Inicio de sesión en el servidor

Una vez hecho esto comienza el trabajo del servidor, cada vez que se recibe una petición esta es gestionada en primer lugar por el archivo app.js que es el núcleo central del funcionamiento del servidor.



## 6) Redirección al fichero de rutas

El fichero app.js se encargará de redirección la solicitud entrante a su fichero de rutas correspondiente, en este caso al correspondiente a las rutas de usuarios (usuario.routes.js) el cual se encargará de comprobar que ruta coincide con método y patrón de URL (en este caso el método sería POST y la ruta /usuario/login).

```
router.post(
  path: '/usuario/login',
  validateUserLogin,
  UsuarioController.postLogin
);
```

Fig 16 Ruta del servidor para el login.

Las rutas en el servidor se componen de una ruta, un conjunto de middlewares encargados de diferentes funciones como la validación de datos de entrada, la verificación de token, la verificación de roles, etc. y una llamada al método del controlador correspondiente. En este caso como se puede ver en la imagen hay una validación previa a la llamada al login.

## 7) Validación de datos en backend

Esta validación se realiza llamando a una función helper que comprueba que los campos recibidos en el cuerpo de la solicitud cumplen con los requisitos necesarios para poder ser utilizados en la lógica de negocio siguiente a la validación.

Para llevar a cabo esta validación se ha utilizado una la librería **express-validator** que proporciona una serie de funciones de validación y saneamiento para los datos de entrada.

En el caso de que uno o más de los

```
export const validateUserLogin = [
  body('fields: email')
    .trim()
    .notEmpty()
    .withMessage('El correo es requerido.')
    .custom(validator: (value) => {
      const regex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
      if (!regex.test(value)) {
        throw new Error('message: El correo debe ser un correo válido.');
      }
      return true;
    }),
  body('fields: password')
    .trim()
    .notEmpty()
    .withMessage('La contraseña es requerida.')
    .isString()
    .withMessage('La contraseña debe ser una cadena de texto.'),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      const errorMessages = errors.array().map((error) => error.msg);
      return res.status(400).json({ errors: errorMessages });
    }
    next();
  },
];
```

Fig 17 Función de validación del servidor.

datos no cumpla los requisitos se almacenan en un array de errores que es devuelto al cliente en una respuesta de tipo JSON junto con un código de estado de respuesta 400 o *Bad Request*.

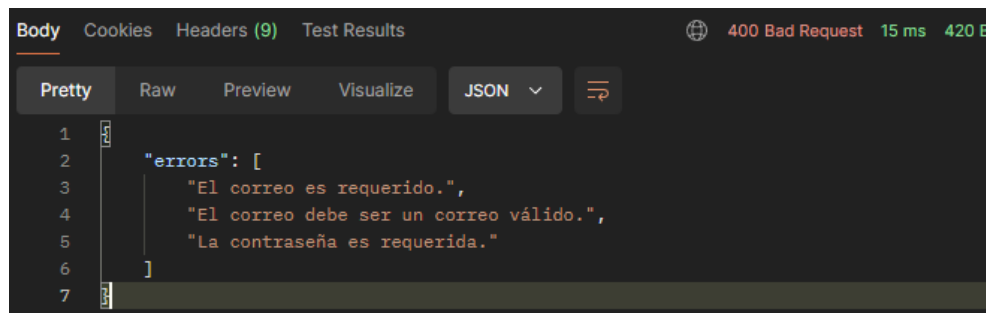


Fig 18 Ejemplo de una petición fallida utilizando Postman.

En el caso de que todo sea correcto se invoca a la función **next()** del validador que marca la finalización del middleware de validación y el comienzo del siguiente fragmento de código de la ruta.

### 8) Llamada al controlador de usuarios

Una vez comprobados los datos comienza el trabajo del controlador, en nuestro caso hemos simplificado los controladores para utilizar un patrón de diseño de software que divide la arquitectura de la aplicación del servidor en 3 capas lógicas principales: controlador, servicio y modelo. Cada una de estas capas tiene responsabilidades específicas y funciona de manera independiente de las otras.

- **Controlador:** Esta es la capa de presentación y es la que maneja las solicitudes HTTP, procesa las respuestas y dirige el flujo de la aplicación.
- **Servicio:** Esta es la capa de lógica de negocio. Contiene la lógica principal de la aplicación y las reglas de negocio. Esta capa interactúa con la capa de modelo para realizar operaciones CRUD (*Create - Read - Update - Delete*) y también puede interactuar con otros servicios de la aplicación.
- **Modelo:** Esta es la capa de acceso a datos. Define cómo interactuar con la base de datos u otras fuentes de datos. Esta capa se encarga de las operaciones de la base de datos como las consultas, las inserciones, las actualizaciones y las eliminaciones.

La ventaja de este enfoque es que proporciona una separación clara de las responsabilidades, lo que facilita la mantenibilidad y la escalabilidad del código. Además, permite que cada capa se desarrolle, se pruebe y se reutilice de forma independiente.

En este caso el controlador se encarga únicamente de recibir la solicitud (req) y provocar una respuesta (res) en base a los resultados que le comunica el servicio correspondiente que está asociado a él, de esta forma se simplifica la funcionalidad del controlador y se evita que, por ejemplo, pueda acceder a los datos de la base de datos.

```
static async postLogin(req, res) {
  const email = req.body.email;
  const password = req.body.password;

  try {
    const { accessToken, refreshToken } = await UsuarioService.userLogin(email, password);
    return res.status(200).json({
      message: 'Inicio de sesión exitoso.',
      access_token: accessToken,
      refresh_token: refreshToken,
    });
  } catch (err) {
    if (err.message === 'Correo o contraseña incorrectos.') {
      return res.status(403).json({ errors: [err.message] });
    }

    return res.status(500).json({ errors: [err.message] });
  }
}
```

Fig 19 Método de login del controlador de usuarios.

### 9) Llamada al servicio de usuarios

Este servicio se va a encargar en primer lugar de generar un pool de conexiones a la base de datos a través de una función de utilidad diseñada para ello y que es utilizada por todos los servicios de la aplicación.

```
// Importación de las librerías necesarias
import { createPool } from 'mysql2';

// Carga de las variables de entorno desde el archivo '.env'
import dotenv from 'dotenv';
dotenv.config();

/** @typedef {Object} DatabaseConfig ... */

/** @type {DatabaseConfig} ... */
const dbConfig = {
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
  database: process.env.DB_NAME,
  timezone: process.env.DB_TIMEZONE,
};

/** Crea un pool de conexiones a la base de datos y lo exporta. ... */
export const dbConn = createPool(dbConfig).promise();
```

Fig 21 Función de utilidad para la creación de un pool de conexiones a la base de datos.

```
static async userLogin(email, password, conn = dbConn) {
  try {
    const user = await UsuarioModel.findByIdByEmail(email, conn);

    if (!user) {
      throw new Error('Correo o contraseña incorrectos.');
```

Fig 20 Estructura del método de login del servicio.

## 10) Llamada al modelo de usuarios

Una vez completado esto el servicio se encarga de la comunicación con el modelo para las diferentes funciones necesarias para llevar a cabo su trabajo, en este caso, se encargará de solicitar una búsqueda del usuario en la base de datos.

En el caso de que no se encuentre ningún resultado se devolverá un nulo si encuentra uno, devolverá los datos correspondientes a ese usuario.

Con estos datos el servicio se encarga de comprobar si los datos introducidos por el usuario y que le han llegado son correctos, si alguno no lo es devolverá un error 'Correo o contraseña incorrectos.' que será gestionado por el servidor devolviendo una respuesta de estado 403 o *forbidden* junto con el mensaje de error y que será mostrado por Angular en la plantilla de login.

## 11) Firma de los tokens de acceso y refresco

En caso de que tanto email como contraseña sean correctos se generarán los tokens de acceso y refresco correspondientes, para ello el servicio de usuarios se comunica con el servicio de token para solicitar la generación y firma de estos tokens utilizando para ello la biblioteca jsonwebtoken de Node.

```
static async findByEmail(email, dbConn) {
  const query =
    'SELECT id, email, password, nombre, primer_apellido, segundo_apellido, dni, rol_id ' +
    'FROM usuario ' +
    'WHERE email = ?';

  try {
    const [rows] = await dbConn.execute(query, [email]);

    if (rows.length === 0) {
      return null;
    }

    return {
      usuario_id: rows[0].id,
      datos_personales: {
        email: rows[0].email,
        password: rows[0].password,
        nombre: rows[0].nombre,
        primer_apellido: rows[0].primer_apellido,
        segundo_apellido: rows[0].segundo_apellido,
        dni: rows[0].dni,
      },
      datos_rol: {
        rol_id: rows[0].rol_id
      }
    };
  } catch (err) {
    throw new Error('Error al obtener el usuario.');
```

Fig 22 Método de búsqueda de un usuario en la base de datos a través del email.

Fig 23 Inicio de sesión fallido.

```
static createAccessToken(user) {
  const payload = {
    user_id: user.usuario_id,
    user_role: user.datos_rol.rol_id,
    user_name: user.datos_personales.nombre
  };

  return sign(payload, process.env.JWT_SECRET_KEY, options: {
    expiresIn: '15m',
  });
}
```

Fig 24 Función encargada de la firma del token de acceso.

Fig 25 Función encargada de la firma del token de refresco.

```
static createRefreshToken(user) {
  const payload = {
    user_id: user.usuario_id,
    user_role: user.datos_rol.rol_id,
    user_name: user.datos_personales.nombre
  };

  return sign(payload, process.env.JWT_REFRESH_SECRET_KEY, options: {
    expiresIn: '1d',
  });
}
```

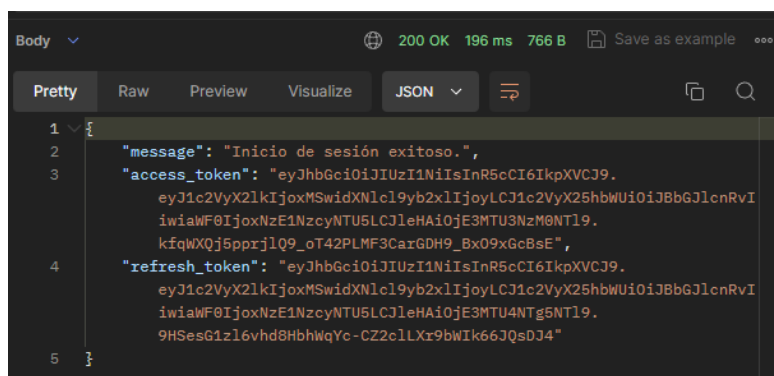
Cada uno de estos tokens tiene una función concreta:

- El **token de acceso** es un token corto que se utiliza para autenticar las solicitudes del usuario a la aplicación, contiene información sobre el usuario (rol, id y nombre) y se utiliza para verificar que el usuario tiene permiso para acceder a ciertos recursos. En nuestro caso el token tiene un tiempo de vida de 15 minutos, lo cual quiere decir que una vez pasado este tiempo, expirará y ya no será válido para autenticar solicitudes.
- Por su parte el **token de refresco** es un token largo que se utiliza para obtener un nuevo token de acceso cuando el token de acceso actual expira. A diferencia del anterior este no se envía en cada solicitud, sino que se almacena de forma segura en el lado del cliente y sólo se utiliza cuando es necesario obtener un nuevo token de acceso (este proceso será comentado más adelante cuando se explique el funcionamiento de otros interceptores del cliente). En nuestro caso el token de refresco tiene un tiempo de vida de 1 día, durante ese tiempo el token puede ser utilizado para obtener nuevos tokens de acceso.

Con esto se consigue un equilibrio entre seguridad y usabilidad. El token de acceso de corta duración minimiza el riesgo de que un atacante pueda utilizar un token robado, mientras que el token de refresco de larga duración permite al usuario mantener su sesión abierta sin tener que iniciar sesión de nuevo cada vez que el token de acceso expira.

### 12) Envío de los tokens al cliente

Una vez firmados los tokens, se almacenará el token de refresco asociado al usuario en la base de datos y si todo ha sido correcto se devolverán ambos al cliente a través del controlador por medio de una respuesta con estado 200 o OK.



```
Body
200 OK 196 ms 766 B Save as example
Pretty Raw Preview Visualize JSON
1 {
2   "message": "Inicio de sesión exitoso.",
3   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMSwidXNle19yb2x1IjoyLCJ1c2VyX25hbWUiOiJBbGJlcnRvIiwiaWF0IjoxNzcyNTU5LCJleHAiOjE3MTU3NzQ0NTl9.kfqWXQj5pprj1Q9_oT42PLMF3CarGDH9_Bx09xGcBsE",
4   "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMSwidXNle19yb2x1IjoyLCJ1c2VyX25hbWUiOiJBbGJlcnRvIiwiaWF0IjoxNzcyNTU5LCJleHAiOjE3MTU4NTg5NTl9.9HSesG1z16vhd8HbhWqYc-CZ2clLXr9bWIk66JQsDJ4"
5 }
```

Fig 26 Ejemplo de respuesta desde el servidor ante un login correcto.

Y esta respuesta será interceptada por el LoginInterceptor tal y como se detalló en el punto 4 del presente apartado. Este interceptor se encargará de almacenar a nivel local del navegador

tanto el token de acceso como el token de refresco utilizando los métodos **storeAccessToken()** y **storeRefreshToken()** del servicio de autenticación.

```
storeAccessToken(token: string): void {  
    localStorage.setItem('access_token', token);  
}  
  
storeRefreshToken(token: string): void {  
    localStorage.setItem('refresh_token', token);  
}
```

Fig 27 Métodos del servicio de autenticación para almacenar los tokens.

http://localhost:4200	
Origin http://localhost:4200	
Key	Value
access_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkjoxLCJ1c2VyX3JvbGUiOiJEsInVzZXJfYmFtZSI6Ikp...
refresh_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkjoxLCJ1c2VyX3JvbGUiOiJEsInVzZXJfYmFtZSI6Ikp...

Fig 28 Almacenamiento local del navegador.

Con esto el proceso de login por parte del usuario habría sido completado y sería redirigido a su zona de acción correspondiente.



Fig 29 Pantalla de opciones del usuario en función del rol.

## B. Protección de rutas en el cliente

Para evitar el acceso a rutas no autorizadas, por ejemplo, rutas propias para pacientes o para administradores hemos utilizado los guardias de ruta o *guards* de Angular que son middleware que se activa al intentar acceder a una ruta, por ejemplo, si estando logados como administrador intentamos acceder a una de las rutas de pacientes como por ejemplo /listado-medicacion el guardia correspondiente nos impedirá el acceso.

Para proteger una ruta en Angular utilizamos **canActivate** al que se le asocia el guardia correspondiente que queremos que se active cuando se intenta acceder a dicha ruta.

```
{  
  path: 'listado-medicacion',  
  component: ListadoMedicacionComponent,  
  canActivate: [patientGuard]  
},
```

Fig 30 Ejemplo de ruta protegida en Angular.

Cuando accedemos a una ruta protegida, el guardia correspondiente se activa y ejecuta el código que contiene en su interior. Por ejemplo, nuestro `patientGuard` lo que va a hacer es comprobar en primer lugar si el usuario está logado, en caso de no estarlo, se le redirigirá a la página de login y, en caso de estarlo, le solicitará al servicio de

```
export const patientGuard = () => {
  const router: Router = inject(Router);
  const authService: AuthService = inject(AuthService);
  let loggedInSubscription: Subscription;
  let isUserLoggedIn: boolean = false;
  let comprobarPatient: boolean = false;

  loggedInSubscription = authService.isLoggedInUser.subscribe(
    observerOrNext: (loggedIn: boolean): void => {
      isUserLoggedIn = loggedIn;
      if (isUserLoggedIn) {
        if (authService.getUserRole() === 2) {
          comprobarPatient = true;
        } else {
          router.navigate(commands: ['/auth/login']).then((r: boolean): void => {});
        }
      }
    }
  )

  return comprobarPatient;
}
```

Fig 31 Guardia de rol de paciente.

autenticación el rol del usuario, algo que se consigue gracias a la decodificación del token de acceso. Si el rol concuerda con el necesario para acceder a dicha ruta se le permite el acceso, en caso contrario no se realiza la redirección a la nueva ruta y se le impide el acceso haciendo que permanezca en su ruta actual.

```
getUserRole(): number | null {
  const accessToken: string = this.getAccessToken();

  if (!accessToken) {
    return null;
  }

  const decodedToken = this.jwtHelper.decodeToken(accessToken);
  const userRoleId = decodedToken.user_role;

  switch (userRoleId) {
    case 1:
      return UserRole.ADMIN;
    case 2:
      return UserRole.PACIENT;
    case 3:
      return UserRole.ESPECIALIST;
    default:
      return null;
  }
}
```

Fig 32 Método del servicio de autenticación para obtener el rol del usuario.

### C. Protección de rutas en el servidor

Así mismo se realiza una protección de rutas en el servidor para que en el caso de que Angular falle, Node responda y evite accesos no autorizados.

La protección de rutas en el servidor se realiza en dos fases:

- En primer lugar, toda ruta que necesite de autorización de acceso pasará una primera verificación que comprobará que el token de acceso es válido.
- En segundo lugar, si el token es válido, se verificará que el rol del

```
router.get(  
  path: '/prescripcion',  
  verifyAccessToken,  
  verifyUserRole(roles: [2]),  
  verifyUserId,  
  PacienteTomaMedicamentoController.getRecetas,  
);
```

Fig 33 Ejemplo de protección de rutas en Node.JS.

usuario que estaba contenido en el token se encuentra entre los posibles que pueden acceder. Esta fase puede tener en algunos casos una verificación extra que compruebe el ID del usuario contenido en el token.

### 1) Verificación del token de acceso

Para verificar el token de acceso hacemos uso de un middleware específico que recoge la cabecera *authorization* (en el siguiente punto se explicará cómo se consigue que esta cabecera esté presente en todas las solicitudes HTTP) y guarda en una variable el token recibido.

A continuación, se realiza una verificación del token para comprobar que no ha sido modificado y que mantiene la firma original de su creación, así mismo se comprueba si el tiempo de vida del token de acceso no ha vencido.

```
export const verifyAccessToken = (req, res, next) => {  
  const accessToken = req.headers['authorization'];  
  if (accessToken) {  
    const token = accessToken.split('Bearer ')[1];  
    try {  
      const decodedToken = verify(token, process.env.JWT_SECRET_KEY);  
  
      req.user_id = decodedToken.user_id;  
      req.user_role = decodedToken.user_role;  
  
      next();  
    } catch (error) {  
      if (error.name === 'TokenExpiredError') {  
        return res.status(401).json({  
          errors: ['El token ha expirado. Inicia sesión de nuevo.'],  
        });  
      } else {  
        return res.status(403).json({  
          errors: ['Token inválido.'],  
        });  
      }  
    }  
  } else {  
    return res.status(403).json({  
      errors: ['No se proporcionó ningún token.'],  
    });  
  }  
};
```

Fig 34 Funcionalidad para la verificación del token de acceso.

Si falla algo se devuelve una respuesta directa al cliente y no se continua con la ejecución en el servidor, en el caso de que el token expire, se devuelve un error 401 o *unauthorized* que en el cliente será procesado para solicitar el refresco de los tokens como se verá más adelante. En cualquier otro caso se devuelve un código de estado 403.

En el caso de que todo sea correcto se guardan dos variables a nivel de servidor:



- **req.user\_id** que servirá, por ejemplo, para utilizarla ya dentro de un controlador e impedir que un usuario de tipo paciente pueda acceder a los datos de otro paciente en las rutas que requieran de añadir un parámetro de búsqueda. De esta forma si el usuario con ID 3 intenta acceder directamente a una ruta que acepte un parámetro id se impedirá el acceso a estos, sino que se utilizará el **user\_id** almacenado en el objeto req.

```
if (req.user_role === 2) {  
  paciente_id = req.user_id;  
} else if (req.user_role === 3) {  
  paciente_id = req.params.usuario_id;  
}
```

Fig 35 Mecanismo que impide que un paciente (role 2) pueda acceder a datos ajenos a los de su cuenta.

- **req.user\_role** que se utilizará para comprobar, entre todas cosas, que un usuario puede acceder a unas determinadas rutas que estén bloqueadas a unos roles específicos.

## 2) Verificación del rol de usuario

En este caso se usa un middleware llamado **verifyUserRole** que acepta como parámetro un array de roles y se encarga de comprobar que el atributo **req.user\_role** tiene un valor establecido y si este role se encuentra entre la lista de roles pasadas por parámetro.

Si algo falla, bien porque el **req.user\_role** no esté establecido o bien porque el rol de este usuario no esté entre los indicados, se devolverá un error de estado 403 al cliente.

```
export const verifyUserRole = (roles) => {  
  return (req, res, next) => {  
    if (!req.user_role || !roles.includes(req.user_role)) {  
      return res.status(403).json({  
        errors: ['No tienes permiso para realizar esta acción.'],  
      });  
    }  
    next();  
  };  
};
```

Fig 36 Middleware para la verificación del rol del usuario.

## 3) Verificación del id del usuario

Por último, en caso de ser necesario verificar el identificador del usuario se utilizará otro middleware llamado **verifyUserId** que comprobará que el **req.user\_id** se ha establecido, en el caso de que esto falle se devolverá un error de estado 403 al cliente.

```
export const verifyUserId = (req, res, next) => {  
  if (!req.user_id) {  
    return res.status(403).json({  
      errors: ['Token inválido.'],  
    });  
  }  
  next();  
};
```

Fig 37 Middleware para la verificación del identificador del usuario.

## D. Manejo de la cabecera *authorization* por el cliente

Como se ha visto en el punto anterior para llevar a cabo la validación y verificación del usuario y de su rol por parte del servidor es necesario que el cliente mande una cabecera HTTP en específico, la cabecera *Authorization*, la cual se compone de la palabra clave **Bearer** seguida del token de acceso que se ha generado en el inicio de sesión (o en el refresco del token).

Para que esto sea posible vuelve a ser necesaria la intervención de los interceptores de Angular, en este caso el interceptor encargado de esta función es el **AuthInterceptor**.

Este interceptor se va a encargar de interceptar cualquier petición HTTP que se

```
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const authToken: string = this.authService.getAccessToken();

    if (authToken) {
      const authReq: HttpRequest<any> = req.clone({ update: {
        headers: req.headers.set('Authorization', `Bearer ${authToken}`)
      }});

      return next.handle(authReq);
    }

    return next.handle(req);
  }
}
```

Fig 38 Generación de la cabecera *Authorization* en el cliente.

haga desde el cliente y a través de **req.header.set()** se encargará de añadir la cabecera *Authorization* con el valor correspondiente en las cabeceras de la solicitud.

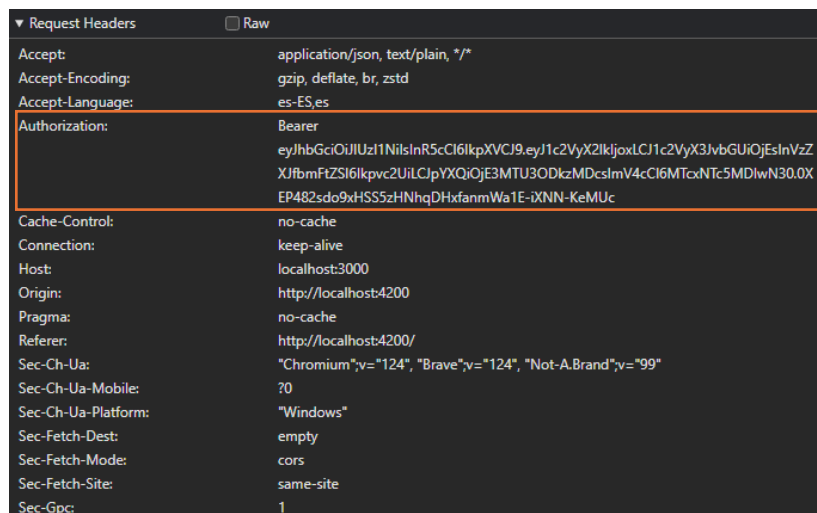


Fig 39 Captura de las herramientas de desarrollador donde se puede ver la cabecera *Authorization*.

## E. Tratamiento del token de refresco

Un punto importante de todo el proceso de autenticación del usuario es el mantenimiento de su sesión para ello se utiliza el token de refresco, un token que como se vio en el apartado de inicio de sesión es largo y que en nuestro caso tiene una vida útil de 1 día, con esto se consigue que cuando el token de acceso expira, la sesión del usuario pueda continuar activa de forma

completamente transparente para este y sin sufrir un cierre de sesión continuo que le obligue a iniciar la sesión cada poco tiempo.

### 1) Intercepción del error 401 desde el servidor.

Para conseguir que este proceso funcione de forma correcta vuelve a ser importante la acción de los interceptores de Angular que, en este caso, lo que harán será manejar las respuestas 401 respondidas por el servidor cuando se produzca una expiración del token de acceso tal y como se explicó en la sección correspondiente a la verificación del token de acceso por parte del servidor.

El interceptor que se utiliza en este caso es el **RefreshTokenInterceptor** que es un interceptor más complejo que los anteriores que se han visto en el documento.

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
    return next.handle(req).pipe(catchError( selector: error => {  
        if (error instanceof HttpErrorResponse && error.status === 401) {  
            return this.handle401Error(req, next);  
        } else {  
            return throwError( errorFactory: () => error);  
        }  
    }));  
}
```

Fig 40 Método principal del interceptor encargado del refresco de token.

Cuando se recibe una respuesta HTTP de tipo error con un estado 401 este interceptor se activa invocando al método privado de su clase **handle401Error()**.

```
private handle401Error(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
    if (!this.isRefreshing) {  
        this.isRefreshing = true;  
        this.refreshTokenSubject.next( value: null);  
  
        return this.authService.refreshToken().pipe(  
            switchMap( project: (tokens: any) => {  
                this.isRefreshing = false;  
                this.refreshTokenSubject.next(tokens.access_token);  
                return next.handle(this.addToken(req, tokens.access_token));  
            }  
        ),  
            catchError( selector: (error) => {  
                this.isRefreshing = false;  
                this.authService.removeTokens();  
                location.reload();  
                return throwError( errorFactory: () => error);  
            }  
        )  
    );  
    } else {  
        return this.refreshTokenSubject.pipe(  
            filter( predicate: token => token != null),  
            take( count: 1),  
            switchMap( project: jwt => {  
                return next.handle(this.addToken(req, jwt));  
            }  
        ));  
    }  
}
```

Fig 41 Método encargado de gestionar el error 401.

Este método se va a encargar de comprobar en primer lugar de si existe ya una solicitud de refresco del token, si no existe se activa el método, el cual llamará al método **refreshToken()** del servicio de autenticación que, como veremos a continuación, será el encargado de llamar al *backend* para solicitar un reseteo del token.

Si esta solicitud es respondida de forma exitosa, se ejecuta el bloque de código que se encuentra dentro del **switchMap** emitiendo un nuevo token de acceso a través del sujeto **refreshTokenSubject**, a continuación, se maneja la solicitud original que falló debido al error 401 utilizando el método **addToken()**.

```
private addToken(req: HttpRequest<any>, token: string) : HttpRequest<any> {  
    return req.clone( update: {  
        setHeaders: {  
            Authorization: `Bearer ${token}`  
        }  
    });  
}
```

*Fig 42 Método encargado de reciclar la petición original que provocó el 401.*

Este método se encarga de clonar la solicitud original utilizando el nuevo token para completar la solicitud.

Si la solicitud de refresco es respondida de forma fallida (por ejemplo, si el tiempo de vida del token de refresco ha expirado también) se ejecutará el bloque **catchError** provocando la eliminación de los tokens que se encuentran actualmente en el navegador y recargando la página actual lo que provocará el cierre de sesión.

En el caso de que cuando se invoque el método **handle401Error()** ya exista una solicitud de actualización de token en curso, se realiza una suscripción a **refreshTokenSubject** y espera hasta que se emita un nuevo token de acceso. Una vez que se emite dicho token, se maneja la solicitud original que fallo con el error 401 pero esta con el nuevo token que se ha generado.

## **2) Llamada al servicio de autenticación**

En el punto anterior se dijo que el interceptor se comunicaba con el servicio de autenticación, concretamente lo hace con el método **refreshToken()** que será el encargado de lanzar la petición POST al servidor para el refresco del token.

En primer lugar, lo que comprobará es que el token de refresco existe solicitándoselo al *localStorage* del navegador, en caso de que no haya un token almacenado se lanzará un error que, como hemos visto más arriba, provocará el cierre de sesión y la vuelta del usuario a la página de login para que vuelva a iniciar sesión.

```

refreshToken(): Observable<any> {
  const refreshToken: string = this.getRefreshToken();

  if (refreshToken) {
    return this.http.post(
      url: `${this.apiUrl}/usuario/refresh-token`,
      body: {refresh_token: refreshToken}
    )
    .pipe(
      tap( observerOrNext: (tokens: any) : void => {
        this.storeAccessToken(tokens.access_token);
        this.storeRefreshToken(tokens.refresh_token);
      }),
      catchError(this.handleError)
    );
  }

  return throwError( errorFactory: () =>
    new Error( message: 'No hay token de refresco almacenado' ));
}

```

Fig 43 Método del servicio de autenticación encargado de solicitar un token de refresco.

Si el token está presente realizará la petición al servidor, si esta es respondida de forma correcta almacenará los nuevos tokens en el almacenamiento local del navegador sobrescribiendo los anteriores, en caso de que se produzca un error en la solicitud, se producirá el cierre de sesión debido al error subsiguiente que sería lanzado.

### 3) Comunicación con el servidor

Una vez que la solicitud es recibida por el servidor y procesada por el fichero app.js es redirigida al enrutador correspondiente que será el encargado de comunicarse con el método correspondiente de su controlador.

```

router.post(
  path: '/usuario/refresh-token',
  UsuarioController.postRefreshToken
);

```

Fig 44 Enrutador para el refresco del token.

#### 4) Manejo del token por el controlador

Al igual que sucedió en el controlador para el login del usuario, el método del controlador encargado del refresco del token se encarga de pasarla el token de refresco al servicio correspondiente y permanecer a la espera de una respuesta por parte de este.

En caso de que todo haya ido de forma correcta se devolverá una respuesta OK con los nuevos tokens, en cualquier otro caso se devolverá un error 403 o 404 y una vez que sea manejado por el cliente se producirá el cierre de sesión.

```
static async postRefreshToken(req, res) {
  const refreshToken = req.body.refresh_token;

  try {
    const { new_access_token, new_refresh_token }
      = await UsuarioService.updateRefreshToken(refreshToken);

    return res.status(200).json({
      message: 'Token de acceso renovado exitosamente.',
      access_token: new_access_token,
      refresh_token: new_refresh_token,
    });
  } catch (err) {
    if (err.message === 'El token no es valido.') {
      return res.status(403).json({
        errors: ['Token de actualización inválido.'],
      });
    }

    if (err.message === 'No se ha proporcionado un token de actualización.') {
      return res.status(403).json({
        errors: ['Token de actualización no proporcionado.'],
      });
    }

    if (err.message === 'El usuario no existe.') {
      return res.status(404).json({
        errors: ['Usuario no encontrado.'],
      });
    }

    return res.status(403).json({
      errors: ['Token de actualización inválido.'],
    });
  }
}
```

Fig 45 Método del controlador encargado del refresco del token.

#### 5) Funcionamiento del servicio

El servicio se va a encargar en primer lugar de verificar el token de refresco y comprobar que es real y que no ha expirado, si sucediera alguno de estos errores se devolvería el error al controlador y finalizaría la ejecución.

En caso de que el token sea correcto se extraerá el ID del usuario de él y se solicitará al modelo de usuario el token de refresco de este usuario para comprobar que coincide con el almacenado actualmente en base de datos, en caso de que no coincida o no exista se devolverá el error correspondiente.

```
static async getRefreshTokenById(id, dbConn) {
  const query =
    'SELECT refresh_token ' +
    'FROM usuario ' +
    'WHERE id = ?';

  try {
    const [rows] = await dbConn.execute(query, [id]);

    if (rows.length === 0) {
      return null;
    }

    return {
      refresh_token: rows[0].refresh_token
    };
  } catch (err) {
    throw new Error({ message: 'Error al obtener el token de refresco.' });
  }
}
```

Fig 46 Método del modelo encargado de devolver el token de refresco.

Si todo ha sido correcto se generarán los nuevos tokens y se actualizará el token de

refresco en el registro del usuario, por último, los nuevos tokens serán devueltos al controlador el cual los devolverá al cliente que se encargará de almacenarlos de forma segura en el navegador.

```
static async updateRefreshToken(refreshToken, conn = dbConn) {
  try {
    if (!refreshToken) {
      throw new Error('message: 'No se ha proporcionado un token de refresco.');
```

Fig 47 Método del servicio encargado de realizar el proceso de refresco del token.

De esta forma se consigue refrescar el token de forma transparente y sencilla para el usuario.

## F. Reinicio de contraseña

Como se vio en la figura 47 una de las opciones que tienen los usuarios de la plataforma es recuperar la contraseña en caso de olvido, este proceso comienza completando un sencillo formulario en el que se le pide el correo al usuario.


The image shows a web form titled "FORMULARIO RECUPERACIÓN CONTRASEÑA" (Password Recovery Form). Below the title, it says "Correo electrónico" (Email). There is a text input field with the placeholder "Introduce tu correo electrónico" (Enter your email). Below the input field is a blue button labeled "Confirmar" (Confirm). The entire form is set against a light blue background with rounded corners.

Fig 48 Formulario de recuperación de contraseña.

En el caso de que el usuario exista en la base de datos se le envía un email con la información para el reinicio de contraseña.

La funcionalidad de creación y envío de correos electrónicos desde el servidor será tratada en mayor profundidad en su apartado correspondiente, en este nos centraremos en el proceso de creación del token de reinicio y el proceso de actualización de contraseña.



## Enhorabuena

Se ha enviado un enlace a su email, porfavor pinche en él para renovar su contraseña

OK

Fig 49 Mensaje de confirmación de envío del email.

### 1) Servicio del cliente

Una vez que se produce la solicitud de reinicio de contraseña se realiza una llamada al servicio y método correspondientes que se encargan de ponerse en contacto con el servidor para solicitar el reinicio.

```
export class ForgottenPasswordService {

  private apiUrl: string = environment.apiUrl;

  constructor(private http: HttpClient) { }

  enviarCorreoRenovacion(email: ForgottenPassswordModel): Observable<any> {
    return this.http.post( url: `${this.apiUrl}/usuario/contrasena-olvidada`, email)
      .pipe(catchError(this.handleError));
  }

  private handleError(errorRes: HttpResponse) : Observable<never> {
    let errorMessage: string = errorRes.error.errors?? 'Ha ocurrido un error durante el proceso';
    return throwError( errorFactory: () => new Error(errorMessage));
  }
}
```

Fig 50 Servicio del cliente encargado de la funcionalidad de reinicio de contraseña.

### 2) Manejo por el servidor

Al igual que en casos anteriores una vez que la solicitud es recibida por el servidor es procesada por el enrutador correspondiente el cual es el encargado de comunicar al controlador los datos que han sido recibidos.

```
router.post(
  path: '/usuario/contrasena-olvidada',
  validateUserPasswordForgot,
  UsuarioController.postForgotPassword
);
```

A su vez el controlador será el encargado de pasarle los datos al servicio que será el verdadero protagonista de llevar a cabo el proceso de reinicio de contraseña.



```
static async postForgotPassword(req, res) {
  const email = req.body.email;

  try {
    await UsuarioService.forgotPassword(email);

    return res.status(200).json({
      message: 'Correo enviado exitosamente.',
    });
  } catch (err) {
    if (err.message === 'Correo no encontrado.') {
      return res.status(404).json({ errors: [err.message] });
    }

    return res.status(500).json({ errors: [err.message] });
  }
}
```

Fig 51 Controlador encargado del reinicio de contraseña.

Lo hará a través de solicitar al modelo de usuarios el correo electrónico que ha recibido en el cuerpo de la solicitud, si no existe se devolverá un error, en caso de encontrarlo se creará y firmará un token con el servicio de token el cual tendrá un tiempo de vida de 1 hora.

```
static createResetToken(user) {
  const payload = {
    email: user.datos_personales.email,
  };

  return sign(payload, process.env.JWT_RESET_SECRET_KEY, { options: {
    expiresIn: '1h',
  }});
}
```

Fig 52 Método del servicio de Token encargado de crear el token de reinicio.

```
static async forgotPassword(email, conn = dbConn) {
  try {
    const user = await UsuarioModel.findByEmail(email, conn);

    if (!user) {
      throw new Error({ message: 'Correo no encontrado.' });
    }

    const resetToken = TokenService.createResetToken(user);

    await TokenService.createToken(user.usuario_id, resetToken, conn);
    await EmailService.sendPasswordResetEmail(email, user, resetToken);
  } catch (err) {
    throw err;
  }
}
```

Fig 53 Servicio encargado del reinicio de contraseña.

Si todo ha sido correcto se produce el envío del correo electrónico el cual contiene el enlace correspondiente a la plataforma de Angular junto con el token de reseteo:

```
static async sendPasswordResetEmail(to, user, resetToken) {
  const transporter = EmailService.#createTransporter();
  const compiledTemplate = EmailService.#compileTemplate(
    {
      templateName: 'reset-password.handlebars',
      data: {
        user,
        resetLink:
          `${process.env.ANGULAR_HOST}:${process.env.ANGULAR_PORT}/auth/reset-password/${resetToken}`,
      },
    }
  );

  const mailDetails = EmailService.#createMailDetails(
    {
      from: 'clinicamedicacoslada@gmail.com',
      to,
      subject: 'Recuperar contraseña - Clínica Médica Coslada',
      compiledTemplate,
    }
  );

  try {
    return await transporter.sendMail(mailDetails);
  } catch (err) {
    throw err;
  }
}
```

Fig 54 Método encargado de enviar un correo electrónico para continuar el proceso de reinicio.

### 3) Continuar el proceso tras el correo electrónico.

En el caso de que no haya habido ningún error el usuario recibirá un correo en que le permitirá continuar con el proceso de reinicio de contraseña.

Haciendo click en el botón de recuperar contraseña será redireccionado a la aplicación Angular para continuar el proceso de reinicio de contraseña.

Concretamente será redireccionado a una ruta que recibe un parámetro de la ruta.

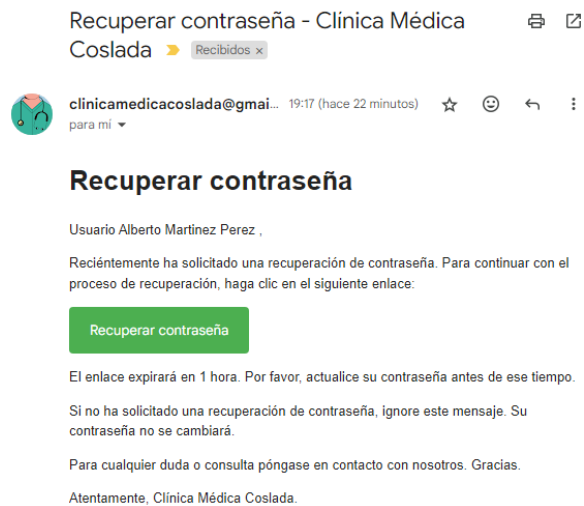


Fig 55 Email de recuperación de contraseña.

```
{
  path: 'auth/reset-password/:token',
  component: RefreshPasswordComponent
},
```

Fig 56 Ruta del enrutador de Angular para continuar el proceso.

Esto es importante ya que este parámetro será capturado por el componente para almacenarlo en una variable que posteriormente será enviada al servidor para verificar la autenticidad del token:

```
this.suscripcionRuta = this.activatedRoute.params.subscribe( observerOrNext: params : Params => {
  this.token = params['token'] || null;
});
```

Fig 57 Captura del parámetro del token.

La vista correspondiente a este componente se trata de un nuevo formulario que solicitará el introducir la nueva contraseña del usuario y repetirla.

El formulario tiene un fondo azul claro con un recuadro central de color más oscuro. En la parte superior, el título "FORMULARIO RECUPERACIÓN CONTRASEÑA" está en mayúsculas y centrado. Debajo, se encuentran dos secciones de entrada de texto. La primera se titula "Nueva contraseña" y contiene un campo de texto con el placeholder "Introduce la nueva contrase" y un ícono de ojo para alternar la visibilidad. La segunda se titula "Repite la contraseña" y contiene un campo de texto con el placeholder "Repite la contraseña" y otro ícono de ojo. En la parte inferior del formulario, hay un botón rectangular de color azul con el texto "Confirmar" en blanco.

*Fig 58 Formulario para la recuperación de contraseña.  
Introducción de nueva contraseña.*

Una vez que el usuario completa el formulario y pulsa en confirmar se llama al servicio correspondiente que se pondrá en contacto con el servidor.

```
export class RefreshPasswordService {  
  
  private apiUrl: string = environment.apiUrl;  
  
  constructor(private http: HttpClient) { }  
  
  renovarContrasena(passwordRefresh: RefreshPasswordModel): Observable<any> {  
    return this.http.post(<url>: `${this.apiUrl}/usuario/contrasena-reset`, passwordRefresh)  
      .pipe(catchError(this.handleError));  
  }  
  
  private handleError(errorRes: HttpResponse) : Observable<never> {  
    let errorMessage: string = errorRes.error.errors??'Ha ocurrido un error durante el proceso';  
    return throwError( errorFactory: () => new Error(errorMessage));  
  }  
}
```

*Fig 59 Servicio de Angular encargado de continuar el reinicio de contraseña.*

#### 4) Finalización del proceso en el servidor

Una vez completado el paso anterior se vuelve de nuevo al servidor para completar el proceso de actualización de contraseña.

```
router.post(  
  '/usuario/contrasena-reset',  
  validateUserPasswordChange,  
  UsuarioController.postResetPassword,
```

*Fig 60 Ruta del servidor que continua el reinicio de contraseña.*

Al igual que en casos anteriores el controlador simplemente se encargará de pasarle los nuevos datos al servicio que será el encargado de completar el proceso.

```
static async postResetPassword(req, res) {
  const newPassword = req.body.password;
  const userEmail = await verifyResetToken(req, res);

  try {
    await UsuarioService.resetPassword(userEmail, newPassword);

    return res.status(200).json({
      message: 'Contraseña actualizada exitosamente.',
    });
  } catch (err) {
    if (err.message === 'Correo no encontrado.') {
      return res.status(404).json({ errors: [err.message] });
    }
    return res.status(500).json({ errors: [err.message] });
  }
}
```

Fig 61 Método del controlador para continuar el proceso de reinicio de contraseña.

```
static async resetPassword(email, password, conn = dbConn) {
  try {
    const user = await UsuarioModel.findByEmail(email, conn);

    if (!user) {
      throw new Error('Correo no encontrado.');
```

Fig 62 Método del servicio para continuar el proceso de reinicio de contraseña.

Lo hará solicitando a la base de datos el correo que se conseguirá a través de la verificación y decodificación del token de reinicio y si todo es correcto, realizará una

```
import pkg from 'bcryptjs';
const { genSalt, hash } = pkg;

/** @name createEncryptedPassword ... */
export const createEncryptedPassword = async (password) => {
  const salt = await genSalt(10);
  return await hash(password, salt);
};
```

Fig 63 Método de utilidades para la encriptación de contraseñas.

encriptación de la contraseña (para ello utilizará la librería de **bcryptjs**) utilizada por el usuario y actualizara los datos del usuario en la base de datos a través del modelo.

## G. Política de *Cross-Origin Resource Sharing* (CORS)

Para terminar con este punto de autorización, autenticación y control de acceso debemos mencionar las CORS, las cuales son una especificación implementada por la mayoría de los

navegadores que permite compartir recursos entre diferentes orígenes. Un origen se define como una combinación de esquema (protocolo), host (dominio) y puerto. Por defecto, por razones de seguridad, un navegador restringe las solicitudes HTTP de origen cruzado iniciadas dentro de un script.

```
import cors from 'cors';

const corsOptions = {
  origin: process.env.CORS_ORIGIN,
  methods: process.env.CORS_METHODS,
  allowedHeaders: process.env.CORS_ALLOWED_HEADERS,
};

export const app = express();

app.use('/api', cors(corsOptions), apiRoutes);
```

Fig 64 Configuración de CORS del servidor.

En resumen, CORS es una forma segura de permitir que un dominio acceda a recursos de otro dominio.

En nuestro caso, toda petición que se realiza contra el servidor pasa una primera evaluación por parte del fichero `app.js` que verificará en

```
CORS_ORIGIN=http://localhost:4200
CORS_METHODS="GET,POST,PUT,DELETE"
CORS_ALLOWED_HEADERS="Content-Type,Authorization"
```

Fig 65 Variables de entorno para las CORS.

primer el origen de la petición, el método de petición recibido y las cabeceras pasadas con esta para comprobar que es coincidente con los valores del archivo de variables de entorno, en el caso de que no lo sea se lanzará un error de CORS al navegador para evitar el acceso no autorizado (bien por origen, bien por método) a los datos contenidos en él.

```
✖ Access to fetch at 'http://localhost:3000/api/usuario/login' from
origin 'http://localhost:4201' has been blocked
by CORS policy: Response to preflight request
doesn't pass access control check: The 'Access-
Control-Allow-Origin' header has a value '
http://localhost:4200' that is not equal to the
supplied origin. Have the server send the header
with a valid value, or, if an opaque response
serves your needs, set the request's mode to 'no-
cors' to fetch the resource with CORS disabled.
```

Fig 66 Error de CORS si se realiza una petición desde un origen no permitido.

### 3. CONSUMO, INTRODUCCIÓN, ACTUALIZACIÓN Y ELIMINACIÓN DE DATOS

En cuanto a las distintas formas de manipulación de los datos , hemos utilizado los cuatro métodos http por excelencia ,de tal forma que el uso de cada uno de ellos conllevará a una acción distinta sobre los datos.

#### Método GET

Se utiliza para solicitar datos de un recurso específico al servidor. Como por ejemplo , obtener las citas solicitadas y acudidas por un paciente.

```
351 router.get(  
352   '/cita',  
353   verifyAccessToken,  
354   verifyUserRole([2]),  
355   verifyUserId,  
356   validateQueryParams,  
357   CitaController.getCitas,  
358 );
```

Fig 67 Ejemplo Ruta GET.

#### Método POST

Se utiliza para registrar datos nuevos al servidor. Un ejemplo de esto es que el administrador dé de alta a un especialista.

```
router.post(  
  '/usuario/registro-especialista',  
  verifyAccessToken,  
  verifyUserRole([1]),  
  validateEspecialistaRegister,  
  UsuarioController.postRegistro,  
);
```

Fig 68 Ejemplo Ruta POST.

#### Método DELETE

Se utiliza para eliminar un recurso existente en el servidor. Un ejemplo de esto es la eliminación de una especialidad por parte del administrador.

```
450 router.delete(  
451   '/especialidad/:especialidad_id',  
452   verifyAccessToken,  
453   verifyUserRole([1]),  
454   validateEspecialidadIdParam,  
455   EspecialidadController.deleteEspecialidad,  
456 );  
457
```

Fig 69 Ejemplo Ruta DELETE.

## Método PUT

Se utiliza para actualizar un recurso existente en el servidor o si este no llegase a existir , puede llegar a crearlo.

```
router.put(  
  '/usuario/actualizar-especialista/:usuario_id',  
  verifyAccessToken,  
  verifyUserRole([1]),  
  validateUsuarioIdParam,  
  validateEspecialistaRegister,  
  UsuarioController.putUsuario,  
);
```

Fig 70 Ejemplo Ruta PUT

## A. Ruta Post

A continuación, veremos más a fondo el funcionamiento de cada una de estas rutas mediante la explicación de los ejemplos anteriores , tanto en el lado del front como su funcionamiento en el lado del back.

En el panel de administración del usuario administrador , entre las distintas opciones que este presenta , tenemos la de crear dos tipos de usuarios : especialistas y pacientes.

Para ello hemos diseñado un componente formulario para cada uno de ellos donde dependiendo de la opción que se seleccione ( ya sea crear paciente o crear un especialista) mediante el uso del enrutado de Angular, este nos redirigirá hacia uno u otro.

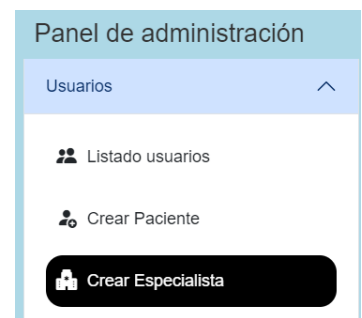


Fig 71 Funciones relativas a listado y creación de usuarios

Formulario de creación de especialista. El formulario está dividido en tres secciones: 'Datos personales' (Nombre, Primer apellido, Segundo apellido, DNI), 'Datos de acceso' (Correo electrónico, Contraseña) y 'Datos especialista' (Nº Colegiado, Turno, Especialidad, Consulta, Imagen, Descripción). Hay botones 'Registrarse' y 'Cancelar' al final.

Fig 72 Formulario creación especialista

Formulario de registro de paciente. El formulario está dividido en tres secciones: 'Datos personales' (Nombre, Primer apellido, Segundo apellido, DNI, Fecha de nacimiento), 'Datos de acceso' (Correo electrónico, Contraseña) y 'Dirección y contacto' (Plaza/Calle, Dirección, Nº, Piso, Puerta, Provincia, Ciudad, CP, Teléfono, Móvil). Hay botones 'Registrarse' y 'Cancelar' al final.

Fig 73 Formulario creación paciente

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

Estos formularios, a su vez cuentan con lo conocido como FormGroup y FormControl, los cuales son unas directivas de Angular que facilitan el manejo de formularios al poder acceder directamente a los elementos de cada input al igual que permite comprobar si los datos insertados son correctos entre otras funcionalidades.

```
this.registerForm = new FormGroup<any>({
  'nombre': new FormControl(
    null,
    [
      Validators.required,
      CustomValidators.validName
    ]
  ),
  'primer_apellido': new FormControl(
    null,
    [
      Validators.required,
      CustomValidators.validSurname
    ]
  ),
});
```

Fig 74 Estructura FormGroup con FormControls

Cuando enviemos el formulario, el FormControl comprobará que los campos sean válidos y si estos no lo son imprime un mensaje de advertencia por cada campo fallido en el html.

```
onRegisterAttempt(): void {
  this.sendedAttempt = true;

  if (this.registerForm.invalid) {
    return;
  }
}
```

Fig 75 FormControl comprueba si los campos son válidos

```
@if (registerForm.invalid && sendedAttempt) {
  <section class="error-message">
    <h3>Por favor, rellena todos los campos correctamente:</h3>
    <ul>
      @if (registerForm.get('nombre').invalid) {
        <li>
          <strong>Nombre:</strong> Debe empezar por mayúscula y el resto de
          letras en minúsculas, salvo en el caso de nombres compuestos.
        </li>
      }
      @if (registerForm.get('primer_apellido').invalid) {
        <li>
          <strong>Primer apellido:</strong> Debe empezar por mayúscula y
          el resto de letras en minúsculas, salvo en el caso de apellidos
          compuestos.
        </li>
      }
    </ul>
  </section>
}
```

Fig 76 FormControl comprueba si cada campo es invalido e imprime su mensaje correspondiente

Por favor, rellena todos los campos correctamente:

- Nombre: Debe empezar por mayúscula y el resto de letras en minúsculas, salvo en el caso de nombres compuestos.
- Primer apellido: Debe empezar por mayúscula y el resto de letras en minúsculas, salvo en el caso de apellidos compuestos.
- Segundo apellido: Debe empezar por mayúscula y el resto de letras en minúsculas, salvo en el caso de apellidos compuestos.
- DNI: Debe tener un formato válido (7 u 8 números y una letra mayúscula. Por ejemplo: 12345678A).
- Correo electrónico: Debe tener un formato válido (usuario@dominio.com | usuario@dominio).
- Contraseña: Debe tener al menos 8 caracteres, una mayúscula, una minúscula, un número y un carácter especial.
- Especialidad: Debe seleccionar una opción.
- Consulta: Debe seleccionar una opción.
- Turno: Debe seleccionar una opción.
- Número Colegiado: Debe ser un número conformado por 9 dígitos (el 0 a la izquierda cuenta como dígito).
- Imagen: Debe seleccionar una imagen.
- Descripción: Debe de tener una descripción.

Fig 77 Mensajes de error de cada campo al ser inválidos

En el caso contrario que todos los campos sean válidos, crearemos un objeto del mismo tipo que la interfaz que contenga todos los campos necesarios para mandarlos al back, en este caso será un objeto de tipo EspecialistModel.

```
import { RolDataModel } from "../rol-data.model"

export interface EspecialistModel {
  usuario_id: number,
  datos_personales: {
    nombre: string,
    primer_apellido: string,
    segundo_apellido: string,
    email: string,
    dni: string,
    password?: string
  },
  datos_especialista: {
    num_colegiado: string,
    descripcion: string,
    imagen: string,
    turno: string,
    especialidad: {
      especialidad_id: number,
      especialidad_nombre?: string
    },
    consulta: {
      consulta_id: number,
      consulta_nombre?: string
    }
  },
  datos_rol?: RolDataModel
}
```

Fig 78 Atributos de la interfaz EspecialistModel



Llegados a este punto es importante comentar como hemos transformado la imagen que sube el usuario a base 64 para su posterior almacenamiento en la base de datos.

```
<article class="image-field">
  <label for="imagen">Imagen</label>
  <input type="file" formControlName="imagen" id="imagen" [ngClass]="{ 'invalid': A
    registerForm.get('imagen').invalid &&
    sendAttempt }" (change)="onFileSelect($event)">
</article>
```

Fig 79 Input que recoge las imágenes subidas al formulario

Para ello , hemos creado el método onFileSelect() el cual comprueba que hemos subido una imagen y a continuación, llamaremos al método toBase64(file) del servicio FileUpload el cual se encargará de transformar la imagen en una url de metadatos mediante el método reader.readAsDataURL. Con esto habremos conseguido que en la variable imageBase64 se guarde una cadena con los metadatos necesarios para almacenar en la base de datos y luego reconstruirlos en el front.

```
onFileSelect(event: { target: { files: File[]; } }) {
  if (event.target.files && event.target.files[0]) {
    this.fileUploadService.toBase64(event.target.files[0]).then(base64 => {
      this.imageBase64 = base64;
      this.imageToShow = base64;
    });
  } else {
    this.imageBase64 = null;
    this.imageToShow = null;
  }
}
```

Fig 80 Método OnFileSelect()

```
toBase64(file: File): Promise<string> {
  return new Promise((resolve, reject) => {
    const reader: FileReader = new FileReader();
    reader.onload = (event: any) => resolve(event.target.result);
    reader.onerror = error => reject(error);
    reader.readAsDataURL(file);
  });
}
```

Fig 81 Método toBase64(file)

A continuación , crearemos un objeto Model que contenga todos los datos necesarios para completar los campos de la base de datos y se lo pasaremos al Auth service el cual es el servicio encargado de hacer gestionar las acciones sobre los usuarios. En este caso llamaremos al método registerSpecialist() al cual le pasamos el objeto Model creado con anterioridad.

```
private generateEspecialist(): EspecialistModel {
  return {
    usuario_id: this.id ?? null,
    datos_personales: {
      nombre: this.registerForm.get('nombre').value,
      primer_apellido: this.registerForm.get('primer_apellido').value,
      segundo_apellido: this.registerForm.get('segundo_apellido').value,
      dni: this.registerForm.get('dni').value,
      email: this.registerForm.get('email').value,
      password: this.registerForm.get('password').value,
    },
    datos_especialista: {
      consulta: {
        consulta_id: this.registerForm.get('consulta').value,
      },
      especialidad: {
        especialidad_id: this.registerForm.get('especialidad').value,
      },
      num_colgado: this.registerForm.get('numero_colgado').value,
      imagen: this.imageBase64,
      descripcion: this.registerForm.get('descripcion').value,
      turno: this.registerForm.get('turno').value,
    }
  }
}
```

Figura 82 Construcción Objeto EspecialistModel

```
this.authService.registerSpecialist(newEspecialist)
  .subscribe({
    next: (response) => {
      this.onSubmit('registrar');
    },
    error: (error: string[]): void => {
      this.onSubmitError(error);
    }
  });
```

Figura 83 Llamada al método de authService para registrar un especialista

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

Tras esto , enviaremos dicho objeto a través de la ruta creada en el servidor , en este caso es /usuario/registro-especialista. Algo importante a destacar es que , al tratarse de una petición post podemos pasar a la llamada de la ruta , el objeto con los campos necesarios para hacer las operaciones requeridas en el back (luego veremos que esto no es posible en todas las rutas).

```
registerSpecialist(newUser: EspecialistaModel): Observable<any> {  
  return this.http.post(`${this.apiUrl}/usuario/registro-especialista`, newUser)  
    .pipe(catchError(this.handleError));  
}
```

Figura 84 método del authService que llama a la ruta de registro de especialistas en el back

A continuación , tras comprobar el token del usuario y si su rol coincide con el necesario para llevar a cabo la acción , validaremos que los datos recibidos son los correctos pero ahora en la parte del back.

```
router.post(  
  '/usuario/registro-especialista',  
  verifyAccessToken,  
  verifyUserRole([1]),  
  validateEspecialistaRegister,  
  UsuarioController.postRegistro,  
);
```

Figura 85 métodos de verificación de token del usuario y su rol

```
export const validateEspecialistaRegister = {  
  validateUserRegister,  
  body('datos_especialista.num_colegiado')  
    .trim()  
    .notEmpty()  
    .withMessage('El número de colegiado es requerido.')  
    .isNumeric()  
    .withMessage('El número de colegiado debe ser un valor numérico.')  
    .custom((value) => {  
      const regex = /^[0-9]{9}$/;  
  
      if (!regex.test(value)) {  
        throw new Error('El número de colegiado debe tener 9 dígitos.');      }  
  
      if (value < 1) {  
        throw new Error('El número de colegiado no puede ser 0 o negativo.');      }  
    })  
    .return true;  
  },  
  body('datos_especialista.descripcion')  
    .trim()  
    .notEmpty()  
    .withMessage('La descripción es requerida.')  
    .isString()  
    .withMessage('La descripción debe ser una cadena de texto.'),  
  body('datos_especialista.turno')  
    .trim()  
    .notEmpty()  
  }
```

Figura 86 validación de los campos del objeto EspecialistaModel enviado al back

Tras esto , los datos del especialista a guardar , una vez bien validados pasaran a ser enviados al controlador de Usuario , en este caso al método postRegistro() el cual este método también es llamado a la hora de registrar un paciente.

En este método simplemente llamamos al service de Usuario donde si en algún caso se produce un error , lo devolvemos al front y en caso de que todo haya salido bien imprimiremos un mensaje de que la operación se ha conseguido completar.

```
static async postRegistro(req, res) {  
  const errors = [];  
  try {  
    await UsuarioService.createUsuario(req.body);  
  
    return res.status(200).json({ message: 'Usuario creado exitosamente.' });  
  } catch (err) {  
    if (err.message === 'El correo electrónico ya está en uso.') {  
      errors.push(err.message);  
    }  
  
    if (err.message === 'El DNI ya está en uso.') {  
      errors.push(err.message);  
    }  
  
    if (err.message === 'El número de colegiado ya está en uso.') {  
      errors.push(err.message);  
    }  
  
    if (errors.length > 0) {  
      return res.status(409).json({ errors });  
    }  
  
    return res.status(500).json({ errors: [err.message] });  
  }  
}
```

Figura 87 método postRegistro() encargado de llamar al servicio de usuarios para que guarde al usuario en cuestión.

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

Finalmente en el servicio de usuario iniciaremos una transacción que no son más que una secuencia de operaciones que se ejecutan como una única unidad lógica de trabajo de tal forma que al tener que ingresar los datos en dos tablas distintas ( usuario y paciente o usuario y especialista) si no se consigue completar las inserciones correctamente , el sistema hará un rollback ( volverá al último guardado estable) o por el caso contrario de que todo haya salido bien hará un commit ( un guardado automático ).

Tras iniciar la transacción, comprobaremos que el usuario a registrar independientemente de que sea paciente o especialista no existe con anterioridad ya que eso significaría que ya estaría registrado y tras hacer comprobado esto, crearemos un objeto el cual asignará los nombres de los campos de la base de datos a los atributos pasados previamente. Finalmente se ejecutará dicha transacción de tal forma que si todo ha salido correctamente hará el commit y si no un rollback.

```
try {
  if (!isConnProvided) {
    await conn.beginTransaction();
  }

  const emailExists = await UsuarioModel.findByIdByEmail(data.datos_personales.email, conn);

  if (emailExists) {
    throw new Error('El correo electrónico ya está en uso.');
```

Figura 88 creación de la transacción y comprobación que el especialista no exista con anterioridad

## B. Método PUT

Como ya se ha mencionado con anterioridad, el método PUT se suele utilizar para la actualización de datos , en este caso nos volvemos a encontrar con el mismo componente del formulario para el registro de especialistas ya que se ha reutilizado para también usarlo para su edición. Es por ello que nada más clicar en la opción de editar especialista nos sacará ya el formulario con los datos rellenos de dicho especialista (haciendo una petición GET al servidor para coger los datos del especialista).

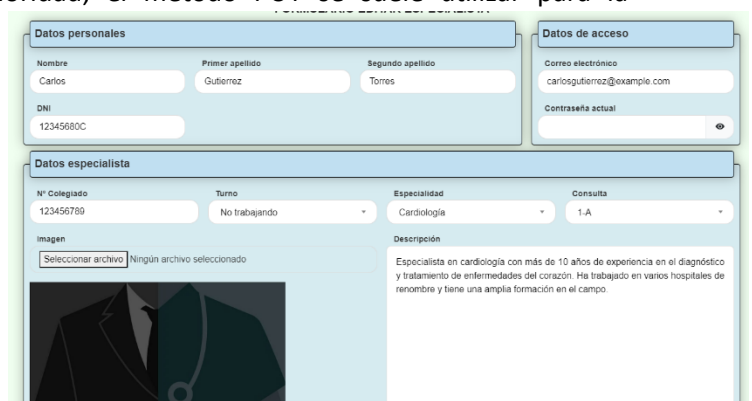
El formulario de edición de especialista está dividido en tres secciones principales: 'Datos personales', 'Datos de acceso' y 'Datos especialista'. La sección 'Datos personales' contiene campos para el nombre, primer apellido, segundo apellido, DNI y correo electrónico. La sección 'Datos de acceso' contiene campos para la contraseña actual y una opción para cambiarla. La sección 'Datos especialista' contiene campos para el número de colegiado, turno, especialidad, consulta, una imagen (con un botón para seleccionar un archivo) y una descripción. El formulario está diseñado con un fondo azul claro y los campos de entrada tienen bordes blancos.

Fig 89 formulario edición especialista

dicho especialista (haciendo una petición GET al servidor para coger los datos del especialista).

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

Al igual que en el caso de la ruta post , si clicamos en editar , el FormControl comprobará que todos los datos son correctos y si es correcto , se creará el objeto que contenga todos los datos de especialista (Model). Luego, en vez de llamar al método registerEspecialist como sucedía con el registro de Especialista , pasaremos a llamar al método updateEspecialist al querer actualizar sus datos.

```
this.authService.updateSpecialist(newEspecialist)
  .subscribe({
    next: (response) => {
      this.onSubmititted('actualizar');
    },
    error: (error: string[]): void => {
      this.onSubmitError(error);
    }
  });
```

Figura 90 llamada al método updateSpecialist()

Si todo funciona correctamente llamará al método onSubmititted que imprimirá un modal con un mensaje indicando que se ha conseguido actualizar correctamente el especialista , por el contrario mostrará un mensaje de error en rojo.

Si entramos en más en detalle con el método updateSpecialist , este tiene un comportamiento similar al de la ruta POST donde en este caso también enviamos el objeto con los datos del especialista. Cabe destacar el uso del `${newUser.usuario_id}` ya que con esto estamos indicando el id del especialista al cual vamos a actualizar los datos.

```
updateSpecialist(newUser: EspecialistModel): Observable<any> {
  return this.http.put(`${this.apiUrl}/usuario/actualizar-especialista/${newUser.usuario_id}`, newUser)
    .pipe(catchError(this.handleError));
}
```

Figura 91 método updateSpecialist() del servicio authService

Tras esto , el service le pasa los datos a la ruta correspondiente en el back.

```
router.put(
  '/usuario/actualizar-especialista/:usuario_id',
  verifyAccessToken,
  verifyUserRole([1]),
  validateUsuarioIdParam,
  validateEspecialistaRegister,
  UsuarioController.putUsuario,
);
```

Figura 92 verificación de token y rol en la ruta PUT del back

Como siempre , tras comprobar que el usuario está logueado y que posee el mismo rol que el necesario para llevar a cabo esta acción, comprueba que le llega correctamente formateados tanto el id del usuario a editar como su conjunto de datos.

```
param('usuario_id')
  .isNumeric()
  .withMessage('El ID del usuario debe ser un valor numérico.')
  .custom((value) => {
    if (value < 1) {
      throw new Error('El ID del usuario debe ser un valor positivo.');
```

Figura 93 Comprobación id especialista

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

A continuación , el objeto con los datos del especialista son enviados al método de actualización de usuarios del controlador de usuarios el cual se encarga simplemente de mandar dichos datos al servicio de usuarios y de mandar una respuesta http al front.

De la misma forma que en el ejemplo de la ruta POST de especialista , crearemos una conexión si es que ya no hay ninguna creada al igual que inicializaremos la transacción para llevar a cabo las operaciones en la base de datos con total seguridad.

```
static async updateUsuario(usuario_id, data, conn = null) {
    const isConnProvided = !!conn;

    if (!isConnProvided) {
        conn = await dbConn.getConnection();
    }

    try {
        if (!isConnProvided) {
            await conn.beginTransaction();
        }
    }
}
```

Figura 94 Establecimiento de conexión y creación transacción

Finalmente comprobaremos que la contraseña , correo del especialista y número de colegiado pasados sean los mismos que los almacenados en la base de datos ( implementación para brindar una mayor seguridad a los usuarios de la aplicación)

```
const password = data.datos_personales.password;
const realPassword = await UsuarioModel.getPasswordById(usuario_id, conn);

const validPassword = await compare(password, realPassword.password);

if (!validPassword) {
    throw new Error('La contraseña actual no es correcta.');
```

```
const emailExists = await UsuarioModel.findByEmail(data.datos_personales.email, conn);

if (emailExists && emailExists.usuario_id !== usuario_id) {
    throw new Error('El correo electrónico ya está en uso.');
```

```
const dniExists = await UsuarioModel.findByDNI(data.datos_personales.dni, conn);

if (dniExists && dniExists.usuario_id !== usuario_id) {
    throw new Error('El DNI ya está en uso.');
```

```
if (data.datos_especialista) {
    const colegiadoExists = await EspecialistaService.readEspecialistaByNumColegiado(data.datos_especialista.num_colegiado, conn);

    if (colegiadoExists && colegiadoExists.usuario_id !== usuario_id) {
        throw new Error('El número de colegiado ya está en uso.');
```

Figura 95 Comprobación de existencia del correo , contraseña y número de colegiado del especialista

Por último, comprobaremos que el usuario existe en su totalidad y que a su vez no sea un usuario administrador para posteriormente actualizar tanto la tabla usuario como la de especialista creando un objeto con sus respectivos campos de la base de datos. Al igual que en el ejemplo anterior , si la transacción se ha completado correctamente se realizará un commit que guarde los datos. Si por el contrario esta no se ha conseguido lograr con éxito hará un rollback hasta el último guardado estable que tenga registro la base de datos.

```
const existingUser = await UsuarioModel.findById(usuario_id, conn);

if (!existingUser) {
    throw new Error('El usuario no existe.');
```

```
if (existingUser.datos_rol.rol_id === 1) {
    throw new Error('El usuario es un admin.');
```

```
const usuario = ObjectFactory.updateUserObject(data);

await UsuarioModel.updateUsuario(usuario, conn);

if (data.datos_paciente) {
    const paciente = ObjectFactory.updatePacienteObject(data);
    await PacienteService.updatePaciente(paciente, conn);
} else {
    const especialista = ObjectFactory.updateEspecialistaObject(data);
    await EspecialistaService.updateEspecialista(especialista, conn);
}

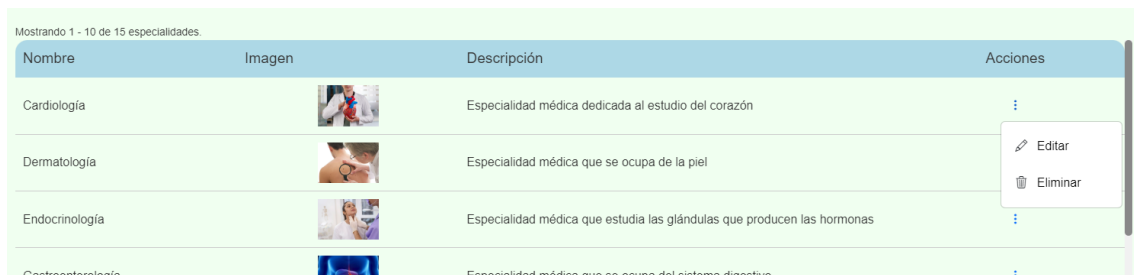
if (!isConnProvided) {
    await conn.commit();
}

} catch (err) {
    if (!isConnProvided) {
        await conn.rollback();
    }
}
```

Figura 96 actualización del usuario por medio del objeto UsuarioModel

## C. Método DELETE

Al igual que en los otros casos , como se ha comentado anteriormente, el método DELETE es utilizado para borrar datos de la base de datos. Partimos de que tenemos un componente que mediante petición GET recogemos el listado de especialidades existentes en la base de datos y los disponemos en una tabla donde mostraremos el nombre de la especialidad ,su correspondiente imagen , una descripción y distintas acciones a poder aplicar a esa especialidad ( a las cuales les hemos vinculado el id de la especialidad para poder actuar dicha acción sobre la especialidad seleccionada) entre las cuales se encuentra la de eliminar la especialidad.







Nombre	Imagen	Descripción	Acciones
Cardiología		Especialidad médica dedicada al estudio del corazón	⋮
Dermatología		Especialidad médica que se ocupa de la piel	⋮ Editar Eliminar
Endocrinología		Especialidad médica que estudia las glándulas que producen las hormonas	⋮
Gastroenterología		Especialidad médica que se ocupa del sistema digestivo	⋮

Figura 97 Vista componente listado de especialidades

En cuanto a la lógica del frontend , tenemos asignado un evento que al pulsar en el botón “eliminar” salga un modal que confirme si el administrador está seguro de querer eliminar dicho registro.

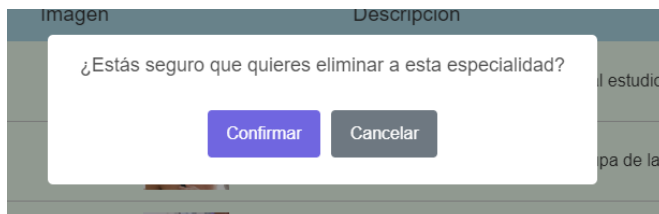


Figura 98 Modal Confirmación eliminación

Una vez confirmada la eliminación ,le pasaremos el id de asociado al service para que siga con el proceso de eliminación de la especialidad.

```
confirmarCancelacion(id: number) {  
  Swal.fire({  
    text: '¿Estás seguro que quieres eliminar a esta especialidad?',  
    confirmButtonText: 'Confirmar',  
    cancelButtonText: 'Cancelar',  
    showCancelButton: true,  
  }).then((result) => {  
    if (result.isConfirmed) {  
      this.adminPanelService  
        .eliminateSpeciality(id)  
        .subscribe({  
          next: (response) => {  
            if (this.actualPage > 1 && this.specialties.length === 1) {  
              this.actualPage--;  
            }  
          }  
        })  
    }  
  })  
}
```

Figura 99 método confirmarCancelacion()

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

En el service , la estructura es similar a lo que hemos estado viendo por ahora , en este caso se utiliza el método delete y como es lógico , le pasamos el id de la especialidad a la ruta del backend para que sepa que especialidad borrar.

```
eliminateSpeciality(id:number):Observable<any>{  
  return this.http.delete<ListedSpecialityModel>(`${this.baseUrl}/especialidad/${id}`).pipe(catchError(this.handleError));  
}
```

Figura 100 método `eliminateSpeciality()` del servicio de especialidades

Tras hacer las comprobaciones de que el usuario esté logueado y tenga el mismo rol que el necesario para llevar a cabo la acción , también comprobaremos que efectivamente se ha enviado el id con el formato correcto ( en este caso que sea numérico) y se llamará al método `deleteEspecialidad` del controlador de Especialidad para seguir con el proceso de eliminación.

```
router.delete(  
  '/especialidad/:especialidad_id',  
  verifyAccessToken,  
  verifyUserRole([1]),  
  validateEspecialidadIdParam,  
  EspecialidadController.deleteEspecialidad,  
);
```

Figura 101 comprobación token y rol de la ruta DELETE del back

```
param('especialidad_id')  
  .isNumeric()  
  .withMessage('El ID de la especialidad debe ser un valor numérico.')  
  .custom((value) => {  
    if (value < 1) {  
      throw new Error('El ID de la especialidad debe ser un valor positivo.');    }  
    return true;  
  })
```

Figura 102 comprobación id especialidad en el back

Tras esto , el controlador llamará al servicio de especialidad , donde al tratarse de una operación simple no es necesario el uso de una transacción, el modelo de especialidad buscará que dicha especialidad existe y procederá a eliminarla de la base de datos.

```
static async deleteEspecialidad(id, conn = dbConn) {  
  try {  
    const idExistente = await EspecialidadModel.findById(id, conn);  
    if (!idExistente) {  
      throw new Error('Especialidad no encontrada.');    }  
    return await EspecialidadModel.deleteById(id, conn);  
  } catch (err) {  
    throw err;  
  }  
}
```

Figura 103 método `deleteEspecialidad()` del servicio de especialidades del back

Si todo ha salido correctamente , el controlador de Especialidad deberá de devolver una respuesta http 200 y por consiguiente se creará un mensaje modal indicando que se ha podido eliminar exitosamente la especialidad ( se vuelve a pedir el listado de especialidades para que estén actualizados).

```
return res.status(200).json({  
  message: 'Especialidad eliminada exitosamente.',  
});
```

Figura 104 Respuesta 200 si se ha eliminado correctamente la especialidad

```
this.getSpecialities();  
Swal.fire({  
  title: 'Enhorabuena',  
  text: 'Has conseguido eliminar la especialidad correctamente',  
  icon: 'success',  
  width: '50%'  
});
```

Figura 105 Petición para actualizar las especialidades existentes

## D. Método GET

Como se ha comentado anteriormente , el método GET es utilizado para recoger datos almacenados de la base de datos y mostrarlos en el frontend. En este caso , para explicar el funcionamiento de la recogida de datos ( que además incluye paginación) iremos explicando de back a front para que se entienda mejor su funcionamiento.

Dicho esto , comenzaremos con el archivo cita.routes que , como en casos anteriores , es el archivo que contiene todas las rutas disponibles por las cuales el servidor hace operaciones con la base de datos. En este caso , para recoger la información de las citas que tiene un paciente asociadas tendremos que llamar a la ruta /cita.

```
router.get(
  '/cita',
  verifyAccessToken,
  verifyUserRole([2]),
  verifyUserId,
  validateQueryParams,
  CitaController.getCitas,
);
```

*Figura 106 Petición GET de citas*

Tras esto , en el back comprobaremos que el usuario está logueado y que cumple con el rol necesario para poder desempeñar la acción de recogida de datos. A su vez comprobaremos que el id del usuario es enviado en el formato correcto al igual que los parámetros de filtrado para la paginación.

```
export const validateQueryParams = [
  query('role')
    .optional()
    .isNumeric()
    .withMessage('El rol debe ser un valor numérico.'),
  query('page')
    .optional()
    .isNumeric()
    .withMessage('El número de página debe ser un valor numérico.'),
  query('limit')
    .optional()
    .isNumeric()
    .withMessage('El límite de elementos por página debe ser un valor numérico.'),
  query('search')
    .optional()
    .isString()
    .withMessage('El término de búsqueda debe ser una cadena de texto.'),
];
```

*Figura 107 Comprobación parámetros de filtrado*

A continuación, llamaremos al método getCitas del controlador de Citas , donde llamaremos en primera instancia a una función global llamada getSearchValues la cual es utilizada por varios métodos del back para obtener los valores de filtrado.



```
static async getCitas(req, res) {
  try {
    const searchValues = getSearchValues(req, 'medicalDateList');
    const citas = await CitaService.readCitas(req.user_id, searchValues);

    return res.status(200).json({
      prev: citas.prev,
      next: citas.next,
      pagina_actual: citas.pagina_actual,
      paginas_totales: citas.paginas_totales,
      cantidad_citas: citas.cantidad_citas,
      result_min: citas.result_min,
      result_max: citas.result_max,
      fecha_inicio: citas.fecha_inicio,
      fecha_fin: citas.fecha_fin,
      items_pagina: citas.items_pagina,
      citas: citas.resultados,
    });
  } catch (err) {
    return res.status(500).json({
      errors: [err.message],
    });
  }
}
```

*Figura 108 Petición para actualizar las especialidades existentes*

El funcionamiento de la función `getSearchValues` es simple. Si hemos recibido la página en la que nos ubicamos la asignaremos al parámetro `page` para devolverlo en un objeto con los demás tipos de filtros a tener en cuenta, si esto no es así siempre nos devolverá los datos por la página uno. Por otro lado, el parámetro `limit` indica el número de elementos por página, por defecto está establecido a diez elementos por página. En cuanto al parámetro `search`, es utilizado en aquellos componentes donde tenemos implementado un buscador, de tal forma que buscaremos los elementos de la base de datos que contengan el valor a buscar.

```
const page      = parseInt(req.query.page) || 1;
const limit     = req.query.limit          || 10;
const search    = req.query.search         || '';
```

*Figura 109 Parámetros `page`, `limit` y `search` para el filtrado y paginación en back*

Volviendo a nuestro ejemplo de recogida de citas con paginación, este tiene un filtro que permite buscar citas entre un rango de fechas. Si no hemos recibido la fecha de inicio, su valor será el primer día del año actual, es decir, el 01/01/2024. Por otra parte, si no hemos recibido una fecha máxima límite, recogeremos todas las citas que ya han sucedido o planificado hasta el día actual del año 2027

```
case 'medicalDateList':
  const fechaInicioCita = req.query.fechaInicioCita
    ? tz(req.query.fechaInicioCita, 'Europe/Madrid').format('YYYY-MM-DD')
    : tz().startOf('year').format('YYYY-MM-DD');
  const fechaFinCita = req.query.fechaFinCita
    ? tz(req.query.fechaFinCita, 'Europe/Madrid').format('YYYY-MM-DD')
    : tz().add(3, 'year').format('YYYY-MM-DD');

  return {
    page: page,
    limit: limit,
    fechaInicioCita: fechaInicioCita,
    fechaFinCita: fechaFinCita
  };
};
```

*Figura 110 Parámetros `fechaInicioCita` y `fechaFinCita`*

(con esto nos aseguramos de mostrar todas las citas previstas para corto y largo plazo de un paciente). Luego, devolvemos un objeto con todos los filtros a tener en cuenta para buscar las citas.

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

Después, llamamos al método `readCitas` del service de citas. Lo primero que hacemos es recuperar los filtros del objeto de filtrado y usamos el objeto `Model` para hacer la búsqueda a la base de datos.

```
static async readCitas(userId, searchValues, conn = dbConn) {
  try {
    const page = searchValues.page;
    const fechaInicioCita = searchValues.fechaInicioCita;
    const fechaFinCita = searchValues.fechaFinCita;
    const limit = searchValues.limit;

    const {
      rows: resultados,
      actualPage: pagina actual,
      total: cantidad_citas,
      totalPages: paginas_totales,
    } = await CitaModel.fetchAll(userId, searchValues, conn);

    if (page > 1 && page > paginas_totales) {
      throw new Error('La página de citas solicitada no existe.');
```

Figura 111 Funcionamiento método `readCitas()`

Como se puede ver , en el objeto model , se filtra por los parámetros comentados con anterioridad. Para saber la página y los elementos que tiene esta , hemos usado las directivas `limit` y `offset` , donde `limit` se encarga de establecer el número de elementos a devolver y el `offset` sería lo referente a la página , el cual calcula en que objeto posicionarse y cuantos de ahí en adelante tiene que devolver para formar la página necesaria.

```
static async fetchAll(userId, searchValues, dbConn) {
  const page = searchValues.page;
  const fechaInicioCita = searchValues.fechaInicioCita;
  const fechaFinCita = searchValues.fechaFinCita;
  const limit = searchValues.limit;
  const offset = (page - 1) * limit;

  const query =
    'SELECT ' +
    ' cita.id, ' +
    ' cita.fecha, ' +
    ' cita.hora, ' +
    ' cita.informe_id, ' +
    ' especialista_user.id AS especialista_id, ' +
    ' especialista_user.nombre AS especialista_nombre, ' +
    ' especialista_user.primer_apellido AS especialista_primer_apellido, ' +
    ' especialista_user.segundo_apellido AS especialista_segundo_apellido, ' +
    ' especialidad.id AS especialidad_id, ' +
    ' especialidad.nombre AS especialidad_nombre, ' +
    ' consulta.id AS consulta_id, ' +
    ' consulta.nombre AS consulta_nombre, ' +
    ' paciente_user.id AS paciente_id, ' +
    ' paciente_user.nombre AS paciente_nombre, ' +
    ' paciente_user.primer_apellido AS paciente_primer_apellido, ' +
    ' paciente_user.segundo_apellido AS paciente_segundo_apellido ' +
    'FROM ' +
    ' cita ' +
    'INNER JOIN ' +
    ' paciente ON cita.paciente_id = paciente.usuario_id ' +
    'INNER JOIN ' +
    ' especialista ON cita.especialista_id = especialista.usuario_id ' +
    'INNER JOIN ' +
    ' usuario AS especialista_user ON especialista.usuario_id = especialista_user.id ' +
    'INNER JOIN ' +
    ' usuario AS paciente_user ON paciente.usuario_id = paciente_user.id ' +
    'INNER JOIN ' +
    ' especialidad ON especialista.especialidad_id = especialidad.id ' +
    'INNER JOIN ' +
    ' consulta ON especialista.consulta_id = consulta.id ' +
    'WHERE ' +
    ' cita.paciente_id = ? ' +
    ' AND cita.fecha BETWEEN ? AND ? ' +
    'ORDER BY ' +
    ' cita.fecha DESC, ' +
    ' cita.hora DESC ' +
    'LIMIT ? OFFSET ?';

  try {
    const [rows] = await dbConn.execute(query, [
      userId,
      fechaInicioCita,
      fechaFinCita,
      `${limit}`,
      `${offset}`,
    ]);
```

Figura 112 consulta de citas

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

Por último preparamos los parámetros a utilizar en la paginación del front los cuales son:

**Next:** Contiene la url con la llamada a la siguiente página y filtros actuales ( en este caso el rango de fechas).Si esta no es posible tomará un valor null.

**Prev:** Lo mismo que el parámetro next pero en este caso contiene la llamada a la página anterior. Si esta es menor o igual que cero devuelve null ya que no existen páginas negativas.

**Página\_actual:** Indica en que página estamos ubicados actualmente.

**Páginas\_totales:** Muestra el número total de páginas.

**Cantidad\_citas:** Indica el número total de citas en la llamada realizada.

```
if (fechaInicioCita) {
  query += `&fechaInicio=${fechaInicioCita}`;
}

if (fechaFinCita) {
  query += `&fechaFin=${fechaFinCita}`;
}

const prev =
  page > 1
    ? `/cita?page=${page - 1}&limit=${limit}${query}`
    : null;
const next =
  page < paginas_totales
    ? `/cita?page=${page + 1}&limit=${limit}${query}`
    : null;
const result_min = (page - 1) * limit + 1;
const result_max =
  resultados[0].citas.length === limit
    ? page * limit
    : (page - 1) * limit + resultados[0].citas.length;
const fecha_inicio = tz(fechaInicioCita, 'Europe/Madrid').format('DD-MM-YYYY');
const fecha_fin = tz(fechaFinCita, 'Europe/Madrid').format('DD-MM-YYYY');
const items_pagina = parseInt(limit);

return {
  prev,
  next,
  pagina_actual,
  paginas_totales,
  cantidad_citas,
  result_min,
  result_max,
  items_pagina,
  fecha_inicio,
  fecha_fin,
  resultados,
};
```

Figura 113 parámetros necesarios para la paginación

**Items\_pagina:** Nos permite establecer distinto número de ítems por página aunque el por defecto sea 10.

**Fecha\_inicio:** Fecha inicial del filtrado por fechas.

**Fecha\_fin:** Fecha máxima / límite del filtrado por fechas.

**Resultados:** El conjunto de citas recogidas en la página.

En cuanto al front , como se ha mencionado con anterioridad contamos con un componente que nos muestra el listado de citas con paginación al igual que poder filtrarlas por un rango de fechas mediante inputs.

Fecha inicio

dd/mm/aaaa

Fecha fin

dd/mm/aaaa

Items

10

Estas son tus citas, Anaís

Mostrando 1 - 10 de 12 citas

Hora	Fecha	Consulta	Especialista	Especialidad	Acción
12:00:00	03.09.2024	5.E	Patricia Medina Navarro	Hematología	
12:00:00	01.09.2024	5.E	Patricia Medina Navarro	Hematología	
09:00:00	23.06.2024	5.E	Patricia Medina Navarro	Hematología	
09:00:00	23.05.2024	5.E	Patricia Medina Navarro	Hematología	
09:00:00	22.05.2024	5.E	Patricia Medina Navarro	Hematología	

« Anterior

1

2

Siguiente »

Figura 114 Listado con paginación y filtrado front

A la hora de recoger los datos , es similar lo que hemos estado haciendo con otras rutas. Como queremos que el listado de citas aparezca nada más cargue en el componente la gran mayoría de la lógica del GET del componente se encontrará en la función de `ngOnInit` ( esta función es conocida como uno de los muchos life cycles de Angular que son funciones especiales que se ejecutan en un determinado punto del ciclo de vida del componente , en este caso , cuando este se está creando). A su vez, hemos utilizado el objeto `Subject` para gestionar eventos de manera controlada , en este caso para recoger las citas. Como se puede ver en la imagen lo que hemos hecho ha sido indicarle al objeto las funciones y eventos que tiene que ejecutar siempre que se le llame y posteriormente se le hace la primera llamada para conseguir la primera página de citas.

```
ngOnInit(): void {  
  this.actualPage = 1;  
  
  this.getCitasSubject  
    .pipe(  
      debounceTime(500)  
    )  
    .subscribe({  
      next: () => {  
        this.getCitas();  
      },  
      error: (error) => {  
        this.errores = error;  
      }  
    });  
  this.initialLoad = true;  
  this.getCitasSubject.next();  
}
```

*Figura 115 Uso de objeto Subject en la función ngOnInit para recoger las citas nada más cargue el componente*

En este caso inicializamos el array que contendrá las citas en vacío e indicamos que la página por defecto sea la 1. Tras esto llamaremos al método `getCitas()` el cual pedirá los datos al archivo `citasService` .

```
getCitas(fechaInicioCita: string, fechaFinCita: string, perPage: number, page: number) {  
  let query: string = `?page=${page}`;  
  
  if (fechaInicioCita) {  
    query += `&fechaInicioCita=${fechaInicioCita}`;  
  }  
  
  if (fechaFinCita) {  
    query += `&fechaFinCita=${fechaFinCita}`;  
  }  
  
  if (perPage) {  
    query += `&limit=${perPage}`;  
  }  
  
  return this.http.get<CitasListModel>(`${this.baseUrl}/cita${query}`);  
}
```

*Figura 116 llamada al método getCitas del servicio de citas*

Este método es ligeramente distinto a los vistos previamente ya que al tener filtros , tenemos que comprobar que si la llamada manda parámetros que no sean nulos ( es decir , cuando el usuario haya seleccionado un filtro) que sean enviados al back para tener en cuenta esos filtros como criterios de búsqueda ya que si son nulos no tiene sentido tenerlos como parámetros de

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

búsqueda. Por otro lado ,algo importante a destacar es que al tratarse de una ruta GET esta no puede enviar objetos como tal al back sino que tiene que hacerlo a través de variables de url.

Tras todo esto, una vez recibido el objeto que contiene tanto los distintos parámetros necesarios para la paginación como el array de citas , estos serán asignados a sus correspondientes variables en el método privado de showResults().

```
getCitas() {
  this.citas = [];
  this.errores = [];

  let request: Observable<CitasListModel> = this.citasService.getCitas(
    this.fechaInicio,
    this.fechaFin,
    parseInt(this.perPage),
    this.actualPage,
  );

  request.subscribe({
    next: (response: CitasListModel) => {
      this.#showResults(response);
      this.dataLoaded = true;
    },
    error: (error: HttpErrorResponse) => {
      this.dataLoaded = true;

      if (error.error.errors) {
        this.errores = error.error.errors;
      } else {
        this.errores = ['Ha ocurrido un error durante el proceso'];
      }
    }
  ));
}
```

Figura 117 método getCitas()

```
#showResults(data) {
  console.log(data);
  this.nextPageUrl = data.next;
  this.previousPageUrl = data.prev;
  this.totalPages = data.paginas_totales;
  this.resultMin = data.result_min;
  this.resultMax = data.result_max;
  this.totalItems = data.cantidad_citas;
  this.itemsPerPage = data.items_pagina;
  this.actualPage = data.pagina_actual;
  this.citas = data.citas[0].citas;
  this.paciente = data.citas[0].datos_paciente;
}
```

Figura 118 método showResults

A continuación ,en cuanto al código del componente ,Angular posee unos parámetros especiales para sus fors , de tal forma que indicando el array de citas y concatenándolo con la directiva paginate , esta lo hace automáticamente si indicamos el número de citas por página , la página actual y el conjunto de citas en su totalidad.

```
<ng-container *ngFor="let cita of (citas | paginate: { id: 'citasListPagination', itemsPerPage: itemsPerPage, currentPage: actualPage, totalItems: totalItems })">
```

Figura 119 funcionamiento paginate en fors de angular

El id indicado en el for que recorre los datos del array de citas está vinculado a un componente que trae angular por defecto que es el pagination-controls. Entre sus atributos cabe destacar el de directionLinks que simplemente hace que los botones sean operativos , al igual que el previousLabel y nextLabel que sirve para ponerle un texto a los botones de búsqueda de página posterior u anterior. A su vez hemos tenido en cuenta que si existe la url de la siguiente página o previa estos botones aparecerán, si por el contrario no están disponibles , se esconden.

```
@if (previousPageUrl || nextPageUrl) {
  <div class="pagination">
    <pagination-controls
      (pageChange)="actualPage = $event;
      id="citasListPagination"
      maxSize="5"      Attribute "maxSize"
      directionLinks="true"      Attribute
      previousLabel="Anterior"      Attribute
      nextLabel="Siguiente"      Attribute
    >>/pagination-controls>
  </div>
}
```

Figura 120 componente pagination-controls

## MARCO PRÁCTICO. Consumo, introducción, actualización y eliminación de datos

A su vez para que cada vez que pinchemos en buscar cita anterior o siguiente , esto se lleve a cabo es necesario tener un evento que vuelva a mandar una petición al servidor para que busque la siguiente página de cita requerida.

```
changePage(page: number) {  
  console.log(page);  
  if (this.initialLoad) {  
    this.dataLoaded = false;  
    this.actualPage = page;  
    this.getCitasSubject.next();  
  }  
}
```

Figura 121 método changePage()

En cuanto a los filtrados se refiere, estos están asociados a un evento que siempre que cambien de valor llaman a una la función changeFilters esta reinicia el paginado a la página uno , y utiliza el objeto Subject para llevar a cabo el proceso de petición de información al back. Con esto nos aseguramos posibles errores a la hora de recoger los datos en la base de datos

```
<article class="init-date-filter">  
  <label for="fechaInicio">Fecha inicio</label>  
  <input type="date" id="fechaInicio" [(ngModel)]="fechaInicio" (change)="updateFilters()">  
</article>  
<article class="end-date-filter">  
  <label for="fechaFin">Fecha fin</label>  
  <input type="date" id="fechaFin" [(ngModel)]="fechaFin" (change)="updateFilters()"> Att  
</article>
```

Figura 122 sección inputs de filtrado

```
updateFilters():void {  
  if (this.initialLoad) {  
    this.dataLoaded = false;  
    this.actualPage = 1;  
    this.getCitasSubject.next();  
  }  
}
```

Figura 123 método updateFilters()

Con esto ya tendríamos nuestras citas paginadas y con la opción de poder filtrarlas por un rango de fechas establecido.

## 4. GENERACIÓN Y ENVÍO DE CORREOS ELECTRÓNICOS

Como se citó en un apartado previo se ha incluido un servicio de correos electrónicos automáticos para situaciones como crear una cuenta, solicitar cita con un especialista, solicitar un reinicio de contraseña... Para que esto sea posible se ha utilizado la biblioteca **Nodemailer** que permite el envío de los correos electrónicos desde un mail de aplicación y la biblioteca **Handlebars** para la confección y compilación de plantillas semánticas que permite generar HTML dinámico utilizando un objeto JSON para ello.

En este apartado detallaremos el funcionamiento del servicio de emails y para ello utilizaremos la situación en la que un usuario crea una cuenta y recibe el correo de bienvenida a la plataforma.

Una vez que se ha completado el registro del usuario y se ha insertado su información en la base

```
const paciente = ObjectFactory.createPacienteObject(data);
paciente.usuario_id = nuevoUsuario.usuario_id;
await PacienteService.createPaciente(paciente, conn);
await EmailService.sendWelcomeEmail(usuario.email, usuario.nombre);
```

Fig 124 Uso del servicio de mails en la creación de usuarios paciente.

de datos, el servicio de correos electrónicos se encargará de crear un correo electrónico a partir de una plantilla y enviarlo al usuario.

El método **sendWelcomeEmail()** será el encargado de controlar qué se debe hacer y en qué orden, en primer lugar hay que crear un transportador es decir, un objeto de configuración donde se define el servicio a utilizar y valores como los parámetros de usuario y contraseña para que el servidor pueda conseguir autenticarse con la cuenta y mandar el email correspondiente.

```
static #createTransporter() {
  return createTransport({ transporter: {
    service: 'gmail',
    auth: {
      user: process.env.EMAIL_ACCOUNT,
      pass: process.env.EMAIL_PASS,
    },
    tls: {
      rejectUnauthorized: false,
    },
  }});
}
```

```
static async sendWelcomeEmail(to, name) {
  const transporter = EmailService.#createTransporter();
  const compiledTemplate = EmailService.#compileTemplate(
    { templateName: 'welcome.handlebars', data: { name } });

  const mailDetails = EmailService.#createMailDetails(
    process.env.EMAIL_ACCOUNT,
    to,
    { subject: 'Bienvenido a Clínica Médica Coslada',
      compiledTemplate },
  );

  try {
    return await transporter.sendMail(mailDetails);
  } catch (err) {
    throw err;
  }
}
```

Fig 125 Método encargado de mandar el mail de bienvenida.

Fig 126 Generación del transportador.

## MARCO PRÁCTICO. Generación y envío de correos electrónicos

El siguiente paso será compilar la plantilla handlebars para incorporar los datos dinámicos al texto HTML para ello utilizaremos el método

```
static #compileTemplate(templateName, data) {  
  const templatePath = join(templatesPath, templateName);  
  const source = readFileSync(templatePath, options: 'utf8');  
  const template = compile(source);  
  
  return template(data);  
}
```

Fig 127 Compilación de la plantilla handlebars.

**compileTemplate()** que recibe por parámetro el nombre de la plantilla que se debe utilizar y los datos, el objeto JSON con esos datos dinámicos que se deben incorporar. Tras buscar y encontrar la plantilla correspondiente en su directorio y esta se compila junto con el JSON que en este caso concreto es muy sencillo ya que sólo recibe el nombre del usuario.

```
<!DOCTYPE html>  
<html lang="es">  
  <head>  
    <title>Bienvenido a Clínica Médica Coslada</title>  
  </head>  
  <body>  
    <h1>¡Bienvenido, {{name}}!</h1>  
    <p>Estamos encantados de que te hayas unido a Clínica Médica Coslada. Aquí  
      podrás gestionar tus citas médicas de manera eficiente y sencilla.</p>  
    <p>Si tienes alguna pregunta, no dudes en ponerte en contacto con nosotros.</p>  
    <p>Gracias,</p>  
    <p>El equipo de Clínica Médica Coslada</p>  
  </body>  
</html>
```

Fig 128 Plantilla handlebars para el mail de bienvenida.

En tercer lugar, debemos crear los detalles del mail: remitente, destinatario, asunto, contenido y adjuntos (que en este caso será un array vacío, pero en otros casos

```
static #createMailDetails(from, to, subject, html, attachments = []) {  
  return {  
    from: from,  
    to: to,  
    subject: subject,  
    html: html,  
    attachments: attachments,  
  };  
}
```

Fig 129 Generación del objeto con los detalles del mail.

como la confirmación de una cita médica será un PDF) para ello utilizamos el método **createMailDetails()** que devuelve un objeto con estos parámetros.

Una vez hecho esto se envía el mail al usuario usando el método **sendMail()** de Nodemailer que pertenece al objeto transportador.



Fig 130 Mail de bienvenida.



## 4. GENERACIÓN Y DESCARGA DE DOCUMENTOS PDF

### 1) Solicitud de descarga en el cliente

En la aplicación generamos documentación en PDF de diferentes tipos: comprobantes de citas, informes médicos y prescripciones de fármacos. Para explicar el funcionamiento de este sistema de generación y descarga de PDFs utilizaremos el caso de las prescripciones.

Listado medicamentos

Estas son tus medicaciones, Alberto

Descargar en PDF

Nombre	Hora	Dosis	Inicio	Fin	Observaciones
Amoxicilina	07:00	1	03-05-2024	03-06-2024	Tomar antes del desayuno
	17:00	2	04-05-2024		Tomar durante la cena
	22:00	1	01-05-2024	01-06-2024	
	00:00	2	02-05-2024		
Atorvastatina	01:00	2	06-05-2024		
Ibuprofeno	08:00	1	28-04-2024	29-05-2024	Tomar con el desayuno
	23:00	2	28-04-2024		
Lorazepam	23:00	1	05-05-2024	05-06-2024	

Fig 131 Vista de la medicación del paciente.

Cuando el usuario paciente pulsa en “descargar en PDF” se inicia la función **downloadPrescripcion()** que se comunicará con el servicio correspondiente de Angular que será el encargado de llamar al servidor para que se genere el PDF y llevar a cabo la descarga en el dispositivo en el que se esté ejecutando el cliente web.

```
downloadPrescripcion(): void {
  this.medikacionesService
    .getDownloadMedicacion()
    .subscribe( observerOrNext: {
      next: (response: any): void => {
        const blob: Blob = new Blob( blobParts: [response])
        saveAs(
          blob,
          filename: `prescripcion_${this.userData.nombre}_
            ${this.userData.primer_apellido}_
            ${this.userData.segundo_apellido}.pdf`);
      },
      error: (error: string[]): void => {
        thiserrores = error;
      }
    });
}
```

Fig 132 Método de Angular que maneja la descarga.

### 2) Manejo por parte del servidor

Esta petición llega al servidor y como se vio en apartados anteriores es gestionada por app.js que la destina al fichero de rutas correspondiente y desde ahí pasa al método del controlador correspondiente que será el encargado de comunicarse con el servicio que una vez que recibe las

```
static async printPrescripcionPdf(pacienteId, conn = dbConn) {
  try {
    const prescripcionesPaciente =
      await PacienteTomaMedicamentoModel.findPrescripciones(pacienteId, conn);

    if (prescripcionesPaciente.prescripciones.length === 0) {
      throw new Error( message: 'No hay recetas para este paciente. ');
    }

    return await PdfService.generateReceta(prescripcionesPaciente);
  } catch (err) {
    throw err;
  }
}
```

Fig 133 Servicio de Node.JS para el manejo de las prescripciones del paciente.

prescripciones del paciente desde el modelo de la base de datos, le pasa estos datos al servicio de PDF que será el encargado de generar el documento.

### 3) Generación del PDF

El método **generateReceta()** del servicio de PDF será el encargado de generar el PDF el cual se generará a partir de dos bibliotecas, la biblioteca de **handlebars** que vimos en el punto anterior y la biblioteca **puppeteer** que proporciona una API de alto nivel para controlar navegadores Chromium a través del protocolo DevTools, de forma que el PDF se generará en una nueva ventana transparente para el usuario.

```
static async generateReceta(medicamentos) {
  const template = readFileSync(join(templatesSource, 'receta.handlebars'), { options: 'utf8' });
  const compiledTemplate = compile(template);
  const date = new Date();
  const monthNames = [
    'enero',
    'febrero',
    'marzo',
    'abril',
    'mayo',
    'junio',
    'julio',
    'agosto',
    'septiembre',
    'octubre',
    'noviembre',
    'diciembre',
  ];

  medicamentos.fecha = `${date.getDate()} de ${
    monthNames[date.getMonth()]
  } de ${date.getFullYear()}`;

  const bodyHtml = compiledTemplate(medicamentos);
  const filename =
    `receta_${medicamentos.datos_paciente.nombre}_` +
    `${medicamentos.datos_paciente.primer_apellido}_` +
    `${medicamentos.datos_paciente.segundo_apellido}.pdf`;

  return await PdfService.#generatePDFWithTemplate(bodyHtml, filename);
}
```

Fig 134 Método del servicio de PDF encargado de generar el PDF de prescripciones.

Esta generación del PDF se llevará a cabo dentro del directorio `/tmp/pdfs` de forma que una vez generado y enviado al usuario (o en el caso de que suceda un error tras su creación) se pueda eliminar del servidor para que no quede rastro. Para ello se utiliza el método **generatePDFWithTemplate()**

```
static async #generatePDFWithTemplate(bodyHtml, filename) {
  const pdfPath = join(tmpPdfPath, filename);
  const dir = dirname(pdfPath);

  if (!existsSync(dir)) {
    mkdirSync(dir, { options: { recursive: true } });
  }

  await PdfService.#generatePDF(bodyHtml, pdfPath);

  return pdfPath;
}
```

Fig 135 Generación del árbol de directorios.

que será el encargado de generar el árbol de directorios si no existen.

Una vez hecho esto se generará el PDF utilizando puppeteer a través del método **generatePDF()**, este método lanza un navegador con el método **launch()** y abre una nueva página inicializando un contenido basado en las plantillas que se utilizan (cabecera, cuerpo y pie), a continuación, se personalizan las opciones del PDF (formato, márgenes, plantilla de header...) y se genera en la página del navegador.

```
static async #generatePDF(bodyHtml, pdfPath) {
  const header = readFileSync(join(templatesSource, 'header.handlebars'), { options: 'utf8' });
  const compiledHeader = compile(header);

  const footer = readFileSync(join(templatesSource, 'footer.handlebars'), { options: 'utf8' });
  const compiledFooter = compile(footer);

  const headerHtml = compileHeader({ context: {} });
  const footerHtml = compileFooter({ context: {} });

  const browser = await launch();
  const page = await browser.newPage();
  await page.setContent(bodyHtml);

  const options = {
    format: 'A4',
    displayHeaderFooter: true,
    printBackground: true,
    headerTemplate: headerHtml,
    footerTemplate: footerHtml,
    margin: {
      top: '2cm',
      right: '2cm',
      bottom: '2cm',
      left: '2cm',
    },
  };

  await page.pdf({ options: { path: pdfPath, ...options } });

  await browser.close();
}
```

Fig 136 Método encargado de generar el PDF utilizando puppeteer.

```
<section class="meds-data">
  <h2>Medicamentos</h2>
  {{#each prescripciones}}
    <section class="medicamento">
      <h3>{{this.medicamento.nombre}}</h3>
      <p><strong>Descripción:</strong> {{this.medicamento.descripcion}}</p>
      <table>
        <thead>
          <tr>
            <th>Hora</th>
            <th>Dosis</th>
            <th>Fecha de inicio</th>
            <th>Fecha de fin</th>
            <th>Observaciones</th>
          </tr>
        </thead>
        <tbody>
          {{#each this.medicamento.tomas}}
            <tr>
              <td>{{hora}}</td>
              <td>{{dosis}}</td>
              <td>{{fecha_inicio}}</td>
              <td>{{fecha_fin}}</td>
              <td>{{observaciones}}</td>
            </tr>
          {{/each}}
        </tbody>
      </table>
    </section>
  {{/each}}
</section>
```

Fig 137 Fragmento de la plantilla de handlebars.

Una vez finalizado se cierra el navegador y se devuelve la ruta del PDF al controlador que será el encargado de generar una respuesta de tipo descarga.

```
res.status(200).download(file, async (err) => {
  await PdfService.destroyPDF(file);
  if (err) {
    console.error('Error al descargar el archivo:', err);
  }
});
```

Fig 138 Respuesta de descarga desde el servidor.

4) Destrucción del PDF

Completada la respuesta se produce la destrucción del PDF, para ello se vuelve a utilizar el servicio de PDF y en concreto el método **destroyPDF()** que recibe la ruta del PDF por parámetro y se encargara de buscarlo y destruirlo.

```
static async destroyPDF(file) {
  if (existsSync(file)) {
    unlink(file, callback: (unlinkError) => {
      if (unlinkError) {
        console.log(`Error al eliminar el archivo subido: ${unlinkError}`);
      }
    });
  } else {
    console.log(`El archivo no existe en: ${file}`);
  }
}
```

Fig 139 Método encargado de la destrucción del fichero.

5) Descarga en el cliente

Cuando llega la respuesta OK del servidor al cliente junto con el PDF a descargar el servicio informa de ello al componente y a través de un **blob** permite su descarga en el equipo del usuario.

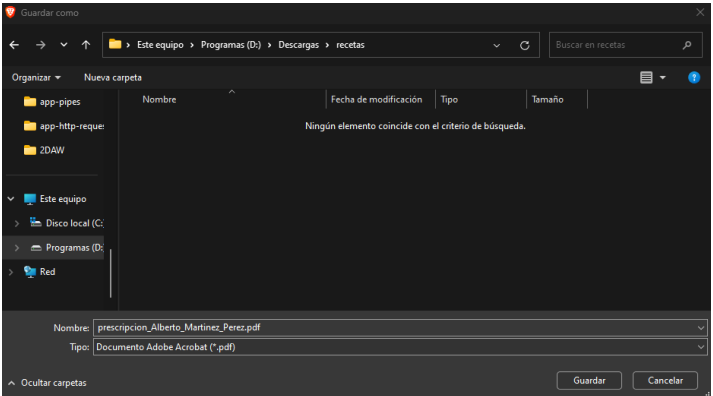


Fig 140 Ventana de descarga del PDF en el navegador.

Clinica Médica Coslada, SL

RECETA DE MEDICAMENTOS

Paciente: Alberto Martinez Perez  
Número de Historia Clínica: 2024042018551244

Medicamentos

Amoxicilina  
Descripción: Antibiótico de amplio espectro

Hora	Dosis	Fecha de inicio	Fecha de fin	Observaciones
07:00	1	03-05-2024	03-06-2024	Tomar antes del desayuno
17:00	2	04-05-2024		Tomar durante la cena
22:00	1	01-05-2024	01-06-2024	
00:00	2	02-05-2024		

Atorvastatina  
Descripción: Estatina para reducir el colesterol

Hora	Dosis	Fecha de inicio	Fecha de fin	Observaciones
------	-------	-----------------	--------------	---------------

Fig 141 Fragmento del PDF resultante.

## 5. GENERACIÓN DE CÓDIGOS QR

Se ha incluido una funcionalidad que permite la generación de un código QR que almacena la información de la cita de un paciente. Este QR acompaña al PDF con los detalles de la cita que haya solicitado el paciente.

```
export const generateQRCode = async (data) => {
  try {
    const citaString = JSON.stringify(data);
    return await toDataURL(citaString);
  } catch (err) {
    throw new Error('Error al generar el código QR.');
```

Fig 142 Función encargada de la generación del código QR.

El funcionamiento de este método es el siguiente, recibe los datos por parámetro (un objeto JSON que incluye datos del paciente, datos del especialista, datos de la cita...) los cuales son convertidos primero en un String y posteriormente usando el método **toDataURL()** de la librería **qrcode** son convertidos a un código QR en base 64, posteriormente este código es pasado por parámetro al servicio de PDF que utilizando la plantilla correspondiente lo añade a la plantilla generada para las citas.

```
const newCitaId = await CitaModel.createCita(cita, conn);
const newCita = await CitaModel.fetchById(newCitaId.id, conn);
const qr = await generateQRCode(newCita);
const paciente = await UsuarioService.readEmailByUserId(cita.paciente_id, conn);
const emailPaciente = paciente.email;

pdf = await PdfService.generateCitaPDF(newCita, qr);

await EmailService.sendPdfCita(newCita, emailPaciente, pdf);
```

Fig 143 Fragmento de código donde se utiliza la funcionalidad de QR.

Una vez que el paciente recibe el correo electrónico con el PDF correspondiente puede visualizar el QR.



Código QR necesario para acceder a la consulta.

Fig 144 Código QR generado y visible en el PDF de cita.

## 6. AUTOMATIZACIÓN DE PROCESOS

Con el fin de automatizar la limpieza de determinadas tablas hemos decidido integrar PL-SQL en el proyecto, en concreto para la generación de 2 eventos, uno de ellos invocando a un procedimiento.

### A. Evento para la eliminación de registros en la tabla token

Para evitar que la tabla que guarda los tokens de reinicio crezca de forma infinita guardando datos que no serán útiles debido a su naturaleza de expirar una hora después de ser creados hemos decidido crear un evento que se encargue de truncar esta tabla todos los días a las 02:00:00.

```
DELIMITER //
CREATE EVENT limpiar_tabla_tokens_event
ON SCHEDULE EVERY 1 DAY
STARTS CONCAT(CURRENT_DATE, ' 02:00:00')
DO
BEGIN
TRUNCATE TABLE token;
END //
DELIMITER ;
```

Fig 145 Evento para la eliminación de registros en la tabla token.

### B. Evento para eliminar las tomas vencidas

Así mismo para evitar tener datos innecesarios en las tablas de toma y paciente\_toma\_medicamento de la base de datos hemos creado un evento que se encargue de llamar de forma periódica (todos los días a las 02:00:00) a un procedimiento que se encargue de limpiar los registros que hayan caducado por ser tomas vencidas.

```
DELIMITER //
CREATE EVENT eliminar_tomas_vencidas_event
ON SCHEDULE EVERY 1 DAY
STARTS CONCAT(CURRENT_DATE, ' 02:00:00')
DO
CALL eliminar_tomas_vencidas_procedure()//
DELIMITER ;
```

Fig 146 Evento para la eliminación de tomas vencidas

El procedimiento consiste en dos partes principales, una primera parte que se encargará de definir las variables y generar el cursor el cual se construye a partir de una sentencia SELECT que recoge todas las fechas cuya fecha\_fin sea menor a la fecha actual, de esa forma nos aseguramos de eliminar todas las tomas que hayan vencido en el día anterior.

Además, se decide cómo se manejarán los errores que puedan aparecer durante la ejecución. En nuestro caso lo haremos guardando cualquier fallo que se produzca en una tabla de log y realizando un rollback de los cambios que se hayan podido producir para evitar que puedan guardarse datos incompletos.

```
-- Inicial transacción seteando autocommit a 0 para poder hacer rollback o commit
SET autocommit = 0;

-- Abrir cursor
OPEN tomas_vencidas;

-- Loop para recorrer el cursor
loop_lectura_tomas: LOOP
    -- Fetch del cursor a la variable _toma_id
    FETCH tomas_vencidas INTO _toma_id;

    -- Si no hay mas registros, salir del loop
    IF done THEN
        LEAVE loop_lectura_tomas;
    END IF;

    -- Eliminar el registro de la tabla paciente_toma_medimento
    DELETE FROM paciente_toma_medimento WHERE toma_id = _toma_id;

    -- Eliminar el registro de la tabla toma
    DELETE FROM toma WHERE id = _toma_id;
END LOOP;

-- Cerrar cursor
CLOSE tomas_vencidas;

-- Commit en caso de éxito
COMMIT;

END;
```

Fig 147 Procedimiento para la eliminación de tomas vencidas. Generación del cursor.

En segundo lugar, tenemos toda la lógica de eliminación la cual se realiza a través de un LOOP de PL-SQL que cicla a través del cursor eliminando los registros en la

paciente\_toma\_medimento y en toma con el id correspondiente al valor que tenga el cursor en la iteración.

Si todo ha funcionado de forma correcta se finaliza el LOOP, se cierra el cursor y se realiza un commit finalizando de esta manera el procedimiento.

```
BEGIN
    -- Declaración de variables
    DECLARE done INT DEFAULT FALSE;
    DECLARE _toma_id INT;

    -- Declaración de variables para el manejo de errores
    DECLARE sql_state CHAR(5);
    DECLARE err_no INT;
    DECLARE err_txt VARCHAR(255);

    -- Declaración del manejador del cursor
    DECLARE tomas_vencidas CURSOR FOR SELECT id FROM toma WHERE fecha_fin < CURDATE();
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- Manejo de errores
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        -- Rollback en caso de error
        ROLLBACK;

        -- Capturar el error
        GET DIAGNOSTICS CONDITION 1
            sql_state = RETURNED_SQLSTATE,
            err_no = MYSQL_ERRNO,
            err_txt = MESSAGE_TEXT;

        -- Insertar el error en la tabla de log
        INSERT INTO error_log (errno, sql_state, error_text)
            VALUES (err_no, sql_state, err_txt);
    END;

END;
```

Fig 148 Procedimiento para la eliminación de tomas vencidas. Bucle de eliminación.

## 7. GENERACIÓN DE DOCUMENTACIÓN AUTOMÁTICA

Para llevar a cabo una labor de documentación del código hemos decidido utilizar dos herramientas: **Swagger** y **JSDoc**. A continuación, se detalla el funcionamiento de ambas herramientas.

### A. Swagger / Swagger UI

Como se explicó en el apartado de tecnologías Swagger es una herramienta cuya finalidad principal es detallar servicios web de tipo RESTful. Por su parte Swagger UI se encarga de convertir esta documentación en una interfaz web que pueda ser usado por cualquier usuario.

Para facilitar el uso de la herramienta se ha usado la librería **swagger-jsdoc**.

#### 1) Archivo de configuración

Se debe generar un fichero `swagger.js` que en nuestro caso hemos decidido almacenar en el directorio `/docs` del servidor y dentro de él importar y definir las opciones básicas de configuración.

```
openapi: '3.0.0',
info: {
  title: 'MediAPP API',
  version: '1.0.0',
  description: 'API para la aplicación MediAPP',
},
servers: [
  {
    url: `${process.env.SERV_API_URL}`,
    description: 'Development server',
  },
],
```

Fig 149 Configuración de Swagger.

Estas opciones básicas se componen de:

- La versión de OpenAPI que se está utilizando.
- Información básica de título, versión y descripción
- El servidor (o conjunto de servidores) que alojarán la API cada uno de ellos definido por una URL única y una descripción.

#### 2) Definición de la ruta en el archivo `app.js`

En el fichero `app.js` hay que definir una ruta que será la encargada de permitir generar la UI de Swagger.

```
// Configuración de Swagger UI para servir la documentación de la API
app.use('/api-docs', serve, setup(swaggerDocument));
```

Fig 150 Ruta del servidor que permitirá el acceso a Swagger UI.



### 3) Componentes de Swagger

Swagger funciona a través de componentes que son objetos que se utilizan para definir los esquemas que se utilizan en la API, a su vez los esquemas son modelos que definen la estructura de datos que enviará la ruta una vez se finalice el proceso de procesamiento de la solicitud del cliente.

Estos esquemas se definen dentro del fichero `swagger.js` una vez que se han definido las configuraciones básicas.

```
Provincia: {
  type: 'array',
  items: {
    type: 'object',
    properties: {
      id: {
        type: 'string',
        description: 'El id de la provincia',
      },
      nombre: {
        type: 'string',
        description: 'El nombre de la provincia',
      },
    },
  },
},
```

Fig 151 Ejemplo de esquema en swagger.

### 4) Comentarios @swagger

Para que las rutas de Swagger UI puedan funcionar es necesario crear un comentario JSDoc con el identificador **@swagger** en los ficheros de rutas (como alternativa se pueden definir objetos **path** en el fichero de configuración).

Estos comentarios se compondrán de información sobre el tipo de ruta, un resumen de lo que se espera de esta ruta, si requiere de autenticación y de los tipos de respuestas (códigos de estado) que se pueden producir con su ejecución. Cada uno de estos códigos tendrá una descripción y el tipo de esquema que devolverá.

```
/**
 * @swagger
 * /provincia:
 *   get:
 *     summary: Obtiene las provincias
 *     tags: [Provincia]
 *     responses:
 *       200:
 *         description: Las provincias fueron obtenidas exitosamente
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Provincia'
 *       404:
 *         description: Las provincias no fueron encontradas
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/NotFoundError'
 *       500:
 *         description: Error al obtener las provincias
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/ServerError'
 */
```

Fig 152 Ejemplo de comentario @swagger.

## 5) Interfaz web

Una vez definidos todos los pasos previos si accedemos a la ruta que se definió en app.js accederemos a la interfaz gráfica de swagger donde pondremos ver el conjunto de rutas que tenga definidas nuestra API.

Haciendo clic en una de ellas podremos ver la información que fue definida en el comentario @swagger.

La potencia de Swagger UI es que permite utilizar las rutas de la API desde aquí sin necesidad de utilizar Postman ni ninguna otra herramienta del estilo. Para ello habría que hacer clic sobre la opción **try it out**.

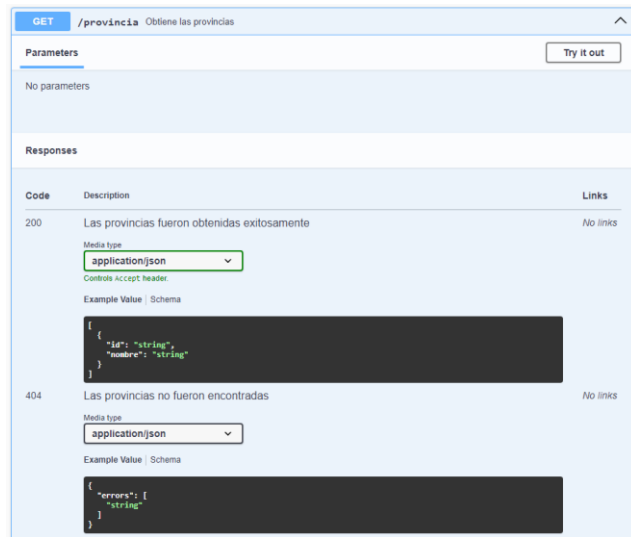


Fig 153 Swagger UI de la ruta GET /api/provincia.

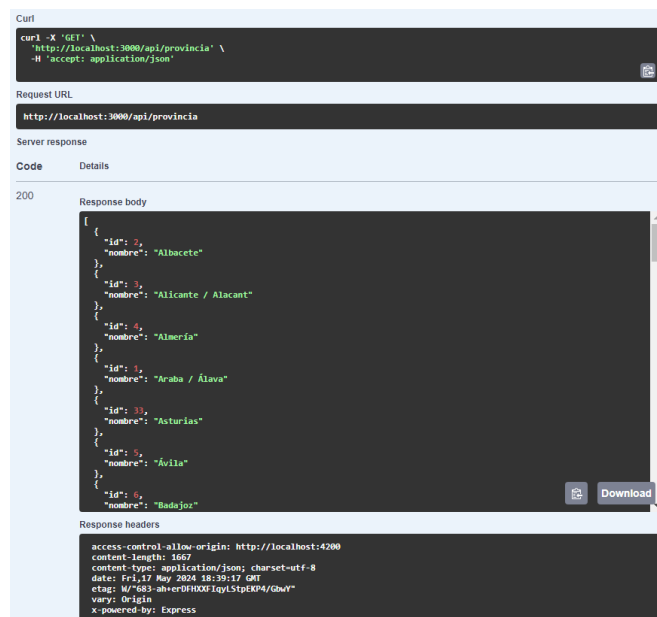


Fig 154 Respuesta de la API.

Además, en la sección **schemas** podremos ver de forma gráfica los datos que produce cada uno de los esquemas que hayan sido definidos en el fichero de configuración.

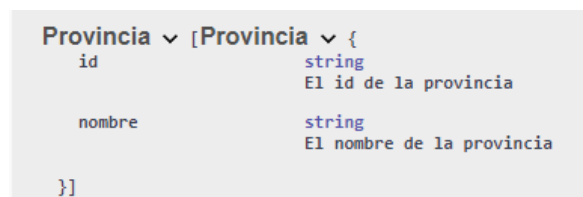


Fig 155 Esquema de Provincia en Swagger UI.

## B. JSDoc

Además, hemos decidido hacer uso de JSDoc para documentar las diversas funciones que se encuentran dentro de la API y del servidor.

Para ello hemos hecho uso de la librería **jsdoc**.

### 1) Configuración de JSDoc

Para realizar esta documentación automática es necesaria una configuración previa:

- Un fichero `jsdoc.json` donde se detallan el directorio de destino de los ficheros, los directorios que están excluidos, etc.
- Un script de ejecución en `package.json` ya que a diferencia de Swagger, JSDoc requiere de su ejecución desde consola para generar la documentación automática.

```
{
  "source": {
    "include": ["."],
    "excludePattern": ".*(node_modules|docs|tmp|public).*"
  },
  "opts": {
    "recurse": true,
    "destination": "./docs/jsdocs"
  }
}
```

Fig 156 Archivo de configuración de JSDoc.

- En el caso de declarar espacios de nombres será necesario crear un fichero `namespaces.js` donde se detallan estos espacios de nombres.

```
/**
 * @namespace Helpers-JWT
 */
```

Fig 157 Fichero `namespaces.js`.

Para declarar un espacio de nombres en el fichero simplemente tendremos que crear un comentario JSDoc y añadir la etiqueta `@namespace` seguida del nombre del espacio de nombres.

- Por último, al igual que ocurría con Swagger, en el fichero `app.js` habrá que definir la ruta que se utilizará para acceder a esta documentación.

```
// Configuración para servir los archivos estáticos desde el directorio 'jsdocs'
app.use('/docs', expressStatic(join(__dirname, 'docs', 'jsdocs')));
```

Fig 158 Ruta a JSDoc en `app.js`.

### 2) Creación de comentarios JSDoc

Para utilizar JSDoc tenemos que crear comentarios de este tipo encima de las funciones. Estos comentarios se caracterizan por tener una serie de etiquetas

```
/**
 * @method findByName
 * @description Método para obtener un medicamento por su nombre.
 * @static
 * @async
 * @memberof MedicamentoModel
 * @param {string} nombre - El nombre del medicamento.
 * @param {Object} dbConn - La conexión a la base de datos.
 * @returns {Promise<Object>} El medicamento.
```

Fig 159 Ejemplo de comentario JSDoc.

como **@method** (define el nombre del método), **@description** (sirve para dar una descripción del método), **@class** (define el objeto JS como una clase), **@memberof** (identifica a que clase o espacio de nombres pertenece), etc.

### 3) Generación de la documentación

Para la generación de la documentación se requiere la ejecución de un script desde consola, el cual se encargará de generar la documentación en archivos HTML.

```
"scripts": {
  "start": "node app.js",
  "dev": "nodemon app.js",
  "doc": "jsdoc -c jsdoc.json",
}
```

Fig 160 Script de ejecución de JSDoc.

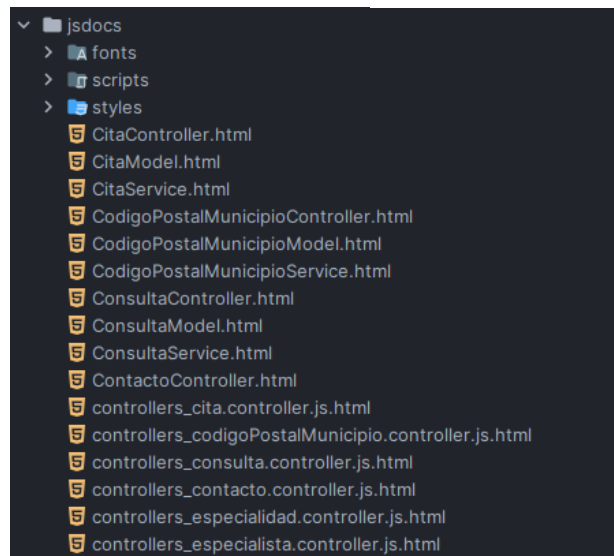


Fig 161 Directorio de JSDoc.

### 4) Visualización en web

Para visualizar el comentario de JSDoc en el archivo HTML debemos acceder a la ruta que se definió en el fichero app.js y buscar el método en concreto.

```
(async, static) findByNombre(nombre, dbConn) → {Promise.  
<Object>}
```

Método para obtener un medicamento por su nombre.

#### Parameters:

Name	Type	Description
nombre	string	El nombre del medicamento.
dbConn	Object	La conexión a la base de datos.

Source: [models/medicamento.model.js, line 153](#)

#### Throws:

Si ocurre un error durante la operación, se lanzará un error.

#### Type

Error

#### Returns:

El medicamento.

#### Type

Promise.<Object>

Fig 162 Visualización de documentación automática en web.