# MARCO PRÁCTICO

## ESTRUCTURA DEL PROYECTO

Texto

Descripción generada automáticamentePara 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 39 Estructura general del proyecto.

1. Directorio *database*

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

Texto

Descripción generada automáticamente

Fig 40 Estructura del directorio 'database'.

1. Texto

   Descripción generada automáticamenteDirectorio *docs*

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

Fig 41 Estructura directorio 'docs'.

1. Directorio *frontend*

Captura de pantalla de un celular

Descripción generada automáticamenteEs 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:

1. **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 42 Estructura del directorio 'frontend'.

Este directorio queda subdividido de la siguiente forma**:**

1. Pantalla de computadora con letras

   Descripción generada automáticamente con confianza media**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 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 Imagen que contiene Texto

   Descripción generada automáticamenteutilizados para compartir datos entre componentes.

Fig 43 Subdirectorios del directorio 'core'.

1. **enviroments:** Contiene los archivos de configuración de enterono para diferentes configuraciones de despliegue.
2. **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.
3. **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 44 Subdirectorios 'enviroments', 'pages' y 'shared'.

1. **Interfaz de usuario gráfica, Aplicación, Chat o mensaje de texto

   Descripción generada automáticamenteassests**: En este directorio se almacenan los recursos estáticos de la aplicación tales como imágenes, iconos, etc.
2. 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:

Fig 45 Estructura del directorio 'server'.

1. **controllers:** En esta carpeta se almacenan los controlares que en una aplicación Node.JS son los responsables de manejar las solicitudes entrantes, procesar los datos (generalmente cediéndoselos al servicio principal asociado a él) y devolver la respuesta correspondiente al cliente.
2. Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

   Descripción generada automáticamente**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.
3. **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.

Fig 46 Estructura de los subdirectorios 'helpers', 'routes' y 'util'.

1. **logs:** Directorio que almacena los ficheros de log generados gracias a las librerías **morgan** y **rotating-file-stream.**
2. **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.
3. **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.
4. **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.
5. **test:** Contiene los archivos para las pruebas del servidor.
6. **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.
7. **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 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.

## 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.

1. Inicio de sesión
2. Pantalla de login

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamentePara 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).

1. Validación de datos en *frontend*

Fig 9 Formulario de login.

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.

Texto, Aplicación

Descripción generada automáticamente

Fig 10 Error en la introducción del email.

Texto

Descripción generada automáticamenteEsto 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.

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

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamenteFig 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.

1. 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.

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

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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

1. Activación del interceptor para el login

Esta petición HTTP dispara un interceptor, concretamente del LoginInteceptor.

Texto

Descripción generada automáticamente

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.

1. 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.

1. Redirección al fichero de rutas

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza mediaEl 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).

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.

1. Texto

   Descripción generada automáticamenteValidación 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.

Fig 17 Función de validación del servidor.

En el caso de que uno o más de los 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*.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

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.

1. 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.

Texto

Descripción generada automáticamente

Fig 19 Método de login del controlador de usuarios.

1. Llamada al servicio de usuarios

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteEste 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.

Fig 20 Estructura del método de login del servicio.

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

1. Llamada al modelo de usuarios

Texto

Descripción generada automáticamenteUna 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.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamenteCon 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.

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

1. 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.

Fig 23 Inicio de sesión fallido.

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

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.

1. 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.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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 inteceptor se encargará de almacenar a nivel local del navegador Texto

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

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

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

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

1. 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.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamentePara 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.

Fig 30 Ejemplo de ruta protegida en Angular.

Texto

Descripción generada automáticamenteCuando 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 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.

Fig 31 Guardia de rol de paciente.

Texto

Descripción generada automáticamente

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

1. 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:

* Interfaz de usuario gráfica, Texto, Aplicación

  Descripción generada automáticamenteEn 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 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.

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

1. Texto

   Descripción generada automáticamenteVerificación 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.

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:

* **Texto

  Descripción generada automáticamentereq.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.

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.

1. 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.

Texto

Descripción generada automáticamente

Fig 36 Middleware para la verificación del rol del usuario.

1. Verificación del id del usuario

Texto

Descripción generada automáticamentePor 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.

Fig 37 Middleware para la verificación del identificador del usuario.

1. 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).

Texto

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

Fig 38 Generación de la cabecera Authorization en el cliente.

Texto

Descripción generada automáticamente

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

1. 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.

Captura de pantalla de un celular

Descripción generada automáticamente

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()**.

Texto

Descripción generada automáticamente

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()**.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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.

1. 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.

Texto

Descripción generada automáticamente

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.

1. Interfaz de usuario gráfica, Sitio web

   Descripción generada automáticamenteComunicación 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.

Fig 44 Enrutador para el refresco del token.

1. Manejo del token por el controlador

Texto

Descripción generada automáticamenteAl 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.

Fig 45 Método del controlador encargado del refresco del token.

1. Funcionamiento del servicio

Texto

Descripción generada automáticamenteEl 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.

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.

Texto

Descripción generada automáticamente

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.

1. Reinicio de contraseña

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamenteComo 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.

Fig 48 Formulario de recuperación de contraseña.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteEn 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.

Fig 49 Mensaje de confirmación de envío del email.

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.

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.

Texto

Descripción generada automáticamente

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

1. Manejo por el servidor

Texto, Sitio web

Descripción generada automáticamenteAl 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 recibido.

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.

Texto

Descripción generada automáticamente

Fig 51 Controlador encargado del reinicio de contraseña.

Texto

Descripción generada automáticamenteLo 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.

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

Texto

Descripción generada automáticamente

Fig 53 Servicio encargado del reinicio de contraseña.

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

Texto

Descripción generada automáticamente

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

1. Continuar el proceso tras el correo electrónico.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamenteEn 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.

Texto

Descripción generada automáticamente

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:

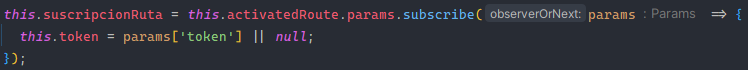


Fig 57 Captura del parámetro del token.

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

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

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

1. Finalización del proceso en el servidor

Texto

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

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.

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Interfaz de usuario gráfica, Texto

Descripción generada automáticamenteLo 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 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.

Fig 63 Método de utilidades para la encriptación de contraseñas.

1. 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 Texto

Descripción generada automáticamentenavegadores 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. En resumen, CORS es una forma segura de permitir que un dominio Texto

Descripción generada automáticamenteacceda a recursos de otro dominio.

Fig 64 Configuración de CORS del servidor.

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

Fig 65 Variables de entorno para las CORS.

Texto

Descripción generada automáticamente

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

## 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.

Texto

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

Fig 69 Ejemplo Ruta DELETE.

Método PUT

Texto

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

Fig 70 Ejemplo Ruta PUT

1. 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.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamenteEn 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.

Fig 71 Funciones relativas a listado y creación de usuarios

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente 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.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Fig 72 Formulario creación especialista

Fig 73 Formulario creación paciente

Texto

Descripción generada automáticamenteEstos formularios, a su vez cuentan con lo conocido como FormsGroup y FormsControl , los cuales son unas directivas de Angular que facilita 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.

Fig 74 Estructura FormGroup con FormControls

Texto

Descripción generada automáticamenteCuando 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.

Texto

Descripción generada automáticamente

Fig 75 FormControl comprueba si los campos son válidos

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

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

Fig 77 Mensajes de error de cada campo al ser inválidos

Texto

Descripción generada automáticamenteEn 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.

Fig 78 Atributos de la interfaz EspecialistModel

Pantalla con letras

Descripción generada automáticamenteLlegados 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.

Fig 79 Input que recoge las imágenes subidas al formulario

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamentePara 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.

Fig 80 Método OnFileSelect()

Fig 81 Método toBase64(file)

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteA 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.

Figura 83 Llamada al método de authService para registrar un especialista

Figura 82 Construcción Objeto EspecialistModel

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).

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamenteA 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.

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamenteTras 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.

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

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 ).

Texto

Descripción generada automáticamenteTras 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.

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

1. Método PUT

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteComo 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).

Fig 89 formulario edición especialista

Texto

Descripción generada automáticamenteAl 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.

Figura 90 llamada al método updateSpecialist()

Si todo funciona correctamente llamará al método onSubmitted 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.

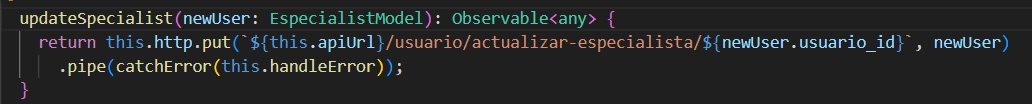


Figura 91 método updateSpecialist() del servicio authService

Texto

Descripción generada automáticamenteTras esto , el service le pasa los datos a la ruta correspondiente en el back.

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

Texto

Descripción generada automáticamenteComo 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.

Figura 93 Comprobación id especialista

Texto

Descripción generada automáticamenteA 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.

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)

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamentePor 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.

Figura 96 actualización del usuario por medio del objeto UsuarioModel

1. 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.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

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.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

Figura 99 método confirmarCancelacion()

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.

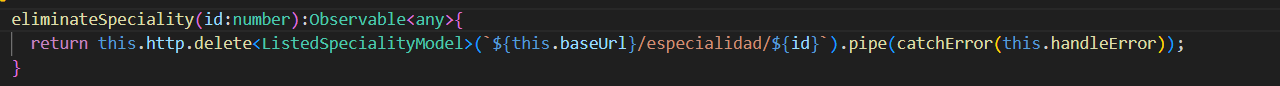


Figura 100 método eliminateSpeciality() del servicio de especialidades

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTras 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.

Figura 102 comprobación id especialidad en el back

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

Texto

Descripción generada automáticamenteTras 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.

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

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteSi 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).

Figura 104 Respuesta 200 si se ha eliminado correctamente la especialidad

Figura 105 Petición para actualizar las especialidades existentes

1. 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.

Texto

Descripción generada automáticamenteDicho 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.

Figura 106 Petición GET de citas

Texto

Descripción generada automáticamenteTras 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.

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.

Texto

Descripción generada automáticamente

Figura 108 Petición para actualizar las especialidades existentes

Texto

Descripción generada automáticamenteEl 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.

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

Texto

Descripción generada automáticamenteVolviendo 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 ( 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.

Figura 110 Parámetros fechaInicioCita y fechaFinCita

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 las búsqueda a la base de datos.

Captura de pantalla de computadora

Descripción generada automáticamente

Figura 111 Funcionamiento método readCitas()

Texto

Descripción generada automáticamenteComo 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.

Figura 112 consulta de citas

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

Texto

Descripción generada automáticamenteNext: 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.

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.

Interfaz de usuario gráfica, Tabla

Descripción generada automáticamenteEn 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.

Figura 114 Listado con paginación y filtrado front

Texto

Descripción generada automáticamenteA 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.

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 .

Captura de pantalla de computadora

Descripción generada automáticamente

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 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.

Texto

Descripción generada automáticamenteTras 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().

Texto

Descripción generada automáticamente

Figura 118 método showResults

Figura 117 método getCitas()

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.

Figura 119 funcionamiento paginate en fors de angular

Texto

Descripción generada automáticamenteEl 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.

Figura 120 componente pagination-controls

Texto

Descripción generada automáticamenteA 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.

Figura 121 método changePage()

Texto

Descripción generada automáticamenteCaptura de pantalla de un celular

Descripción generada automáticamenteEn 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

Figura 123 método updateFilters()

Figura 122 sección inputs de filtrado

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.

Texto

Descripción generada automáticamenteUna vez que se ha completado el registro del usuario y se ha insertado su información en la base de datos, el servicio de correos electrónicos se encargará de crear un correo electrónico a partir de una plantilla y enviarlo al usuario.

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

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteEl 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.

Fig 125 Método encargado de mandar el mail de bienvenida.

Fig 126 Generación del transportador.

Texto

Descripción generada automáticamenteEl siguiente paso será compilar la plantilla handlebars para incorporar los datos dinámicos al texto HTML para ello utilizaremos el método **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.

Fig 127 Compilación de la plantilla handlebars.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Fig 128 Plantilla handlebars para el mail de bienvenida.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamenteTexto

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

Fig 129 Generación del objeto con los detalles del mail.

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.

## 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.

Tabla

Descripción generada automáticamente

Fig 131 Vista de la medicación del paciente.

Texto

Descripción generada automáticamenteCuando 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.

Fig 132 Método de Angular que maneja la descarga.

1. Manejo por parte del servidor

Texto

Descripción generada automáticamenteEsta 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 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.

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

1. 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.

Texto

Descripción generada automáticamente

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

Fig 135 Generación del árbol de directorios.

Texto

Descripción generada automáticamenteUna 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.

Fig 136 Método encargado de generar el PDF utilizando puppeteer.

Texto

Descripción generada automáticamente

Fig 137 Fragmento de la plantilla de handlebars.

Texto

Descripción generada automáticamenteUna 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.

Fig 138 Respuesta de descarga desde el servidor.

1. Destrucción del PDF

Captura de pantalla de un celular

Descripción generada automáticamenteCompletada 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.

Fig 139 Método encargado de la destrucción del fichero.

1. Descarga en el cliente

Captura de pantalla de un celular

Descripción generada automáticamenteCuando 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.

Imagen que contiene Texto

Descripción generada automáticamente

Fig 141 Fragmento del PDF resultante.

## GENERACIÓN DE CÓDIGOS QR

Texto

Descripción generada automáticamenteSe 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.

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.

Texto

Descripción generada automáticamente

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

Descripción generada automáticamente

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

## 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.

1. 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.

Texto

Descripción generada automáticamente

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

1. 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.

Texto

Descripción generada automáticamente

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.

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamenteEn 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 tabla paciente\_toma\_medicamento 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.

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

## 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.

1. 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.

Texto

Descripción generada automáticamente

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.

1. 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.

Texto

Descripción generada automáticamente

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

1. Componentes de Swagger

Texto

Descripción generada automáticamenteSwagger 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 procesado de la solicitud del cliente.

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

Fig 151 Ejemplo de esquema en swagger.

1. Comentarios @swagger

Para que las rutas de Swagger UI puedan funcionar es necesario crear un comentario JSDoc con el identificar **@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.

Texto

Descripción generada automáticamente

Fig 152 Ejemplo de comentario @swagger.

1. Interfaz web

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteUna 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.

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

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***.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Fig 154 Respuesta de la API.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteAdemás, 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.

1. 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:

* Texto

  Descripción generada automáticamenteUn fichero jsdoc.json donde se detallan el directorio de destino de los ficheros, los directorios que están excluidos, etc.

Fig 156 Archivo de configuración de JSDoc.

* 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.
* Interfaz de usuario gráfica, Texto, Aplicación

  Descripción generada automáticamenteEn el caso de declarar espacios de nombres será necesario crear un fichero namespaces.js donde se detallen estos espacios de nombres.

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.



Fig 158 Ruta a JSDoc en app.js.

1. Creación de comentarios JSDoc

Texto

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

Fig 159 Ejemplo de comentario JSDoc.

1. Generación de la documentación

Imagen que contiene reloj, grande, verde, calle

Descripción generada automáticamentePara 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.

Fig 160 Script de ejecución de JSDoc.

Texto

Descripción generada automáticamente

Fig 161 Directorio de JSDoc.

1. Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

   Descripción generada automáticamenteVisualización 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.

Fig 162 Visualización de documentación automática en web.