

A dark blue vertical bar is positioned on the left side of the page. A blue arrow-shaped banner points to the right from this bar, containing the text 'TRABAJO DE FIN DE CICLO'. In the bottom-left corner, there are several thin, curved, light blue lines that sweep upwards and to the right.

TRABAJO DE FIN DE CICLO

DESARROLLO DE UNA APLICACIÓN WEB DE GESTIÓN MÉDICA

AUTORES: Rafael Romero Roibu y Alberto Martínez Pérez

TUTOR: Elkin Guadilla González

FECHA DE ENTREGA: 12 de junio de 2024

ÍNDICE DE CONTENIDOS

RESUMEN	9
ABSTRACT.....	10
INTRODUCCIÓN	11
OBJETIVOS.....	12
ALCANCE DEL PROYECTO.....	13
1. REQUISITOS FUNCIONALES Y NO FUNCIONALES	13
2. PROTOTIPO / MOCKUP	15
A. Navegación a través de la barra de navegación	15
B. Mi Espacio - Rol Administrador	16
1) Gestión de usuarios	16
2) Gestión de especialidades	17
3) Gestión de especialistas	18
C. Mi Espacio - Rol Especialista	19
1) Consultar agenda diaria.....	19
2) Gestión historia clínica - Medicación.....	19
3) Gestión historia clínica - Mediciones de glucosa y tensión arterial	20
4) Gestión historia clínica - Informes médicos	21
D. Mi Espacio - Rol Paciente	22
1) Solicitar cita	22
2) Historial de citas	23
3) Historial clínico	24
4) Editar perfil	25
3. TECNOLOGÍAS USADAS.....	26
A. Angular.....	26
B. Figma	26
C. Postman	27
D. Git	27
E. GitHub.....	27
F. Express.js	28
G. WebStorm.....	28
H. MySQL.....	29
I. MySQL Workbench	29
J. <i>Procedural Language / Structured Query Language (PL/SQL)</i>	29

K. Bootstrap	30
L. Handlebars.js	30
M. <i>Sassy Cascading StyleSheets</i> (SCSS)	30
N. <i>JavaScript Object Notation Web Tokens</i> (JWT)	31
O. Swagger	31
P. Swagger UI	31
Q. JSDoc	32
R. Mocha, Chai y Supertest	32
S. Markdown	33
4. DIAGRAMAS DE ENTIDAD-RELACIÓN	34
A. Diagrama de Chen	34
B. Diagrama de estructura de datos	35
5. DIAGRAMA DE CASOS DE USO	36
A. Herencia de actores	36
B. Casos de uso del usuario	36
C. Casos de uso del administrador	37
D. Casos de uso del paciente	37
E. Casos de uso del especialista	37
MARCO PRÁCTICO	38
1. ESTRUCTURA DEL PROYECTO	38
A. Directorio <i>database</i>	38
B. Directorio <i>docs</i>	38
C. Directorio <i>frontend</i>	39
D. Directorio <i>server</i>	40
2. AUTENTICACIÓN, AUTORIZACIÓN Y CONTROL DE ACCESO	42
A. INICIO DE SESIÓN	42
1) Pantalla de login	42
2) Validación de datos en <i>frontend</i>	43
3) Llamada al servicio de autenticación	44
4) Activación del interceptor para el login	44
5) Inicio de sesión en el servidor	45
6) Redirección al fichero de rutas	45
7) Validación de datos en <i>backend</i>	45
8) Llamada al controlador de usuarios	46
9) Llamada al servicio de usuarios	47
10) Llamada al modelo de usuarios	48

11) Firma de los tokens de acceso y refresco	48
12) Envío de los tokens al cliente	49
B. PROTECCIÓN DE RUTAS EN EL CLIENTE	50
C. PROTECCIÓN DE RUTAS EN EL SERVIDOR	51
1) Verificación del token de acceso	52
2) Verificación del rol de usuario	53
3) Verificación del id del usuario	53
D. MANEJO DE LA CABECERA <i>AUTHORIZATION</i> POR EL CLIENTE	53
E. TRATAMIENTO DEL TOKEN DE REFRESCO	54
1) Intercepción del error 401 desde el servidor.	54
2) Llamada al servicio de autenticación	56
3) Comunicación con el servidor	56
4) Manejo del token por el controlador	57
5) Funcionamiento del servicio	57
F. REINICIO DE CONTRASEÑA	58
1) Servicio del cliente	59
2) Manejo por el servidor	59
3) Continuar el proceso tras el correo electrónico.	61
4) Finalización del proceso en el servidor	62
G. POLITICA DE <i>CROSS-ORIGIN RESOURCE SHARING</i> (CORS)	63
3. CONSUMO, INTRODUCCIÓN, ACTUALIZACIÓN Y ELIMINACIÓN DE DATOS	64
4. GENERACIÓN Y ENVÍO DE CORREOS ELECTRÓNICOS	64
5. GENERACIÓN Y DESCARGA DE DOCUMENTOS PDF	65
6. GENERACIÓN DE CÓDIGOS QR	65
7. AUTOMATIZACIÓN DE PROCESOS	66
A. EVENTO PARA LA ELIMINACIÓN DE REGISTROS EN LA TABLA TOKEN	66
B. EVENTO PARA ELIMINAR LAS TOMAS VENCIDAS	66
8. DOCUMENTACIÓN AUTOMÁTICA	68
A. SWAGGER / SWAGGER UI	68
1) Archivo de configuración	68
2) Definición de la ruta en el archivo app.js	68
3) Componentes de Swagger	69
4) Comentarios @swagger	69
5) Interfaz web	69
B. JSDoc	71

1) Configuración de JSDoc	71
2) Creación de comentarios JSDoc	72
3) Generación de la documentación	72
4) Visualización en web	73
CONCLUSIONES.....	74
BIBLIOGRAFÍA.....	75
WEBGRAFÍA.....	76

ÍNDICE DE FIGURAS

Fig 1 Prototipo de la navegación a través de la barra de navegación de la web.....	15
Fig 2 Prototipo para la gestión de usuarios por parte del administrador.....	16
Fig 3 Prototipo para la gestión de especialidades por parte del administrador.....	17
Fig 4 Prototipo para la gestión de especialistas por parte del administrador.....	18
Fig 5 Prototipo para la consulta de la agenda diaria por parte del especialista.....	19
Fig 6 Prototipo para la gestión de la medicación por parte del especialista.....	19
Fig 7 Prototipo para consulta de las mediciones de glucosa y tensión arterial del paciente.....	20
Fig 8 Prototipo para la consulta y generación de nuevos informes médicos.....	21
Fig 9 Prototipo para la solicitud de citas con especialistas por parte del paciente.....	22
Fig 10 Prototipo para la gestión de citas por parte del paciente.....	23
Fig 11 Prototipo para la inserción de datos de mediciones de tensión arterial y glucosa por parte del paciente.....	24
Fig 12 Prototipo para la edición del perfil por parte del paciente.....	25
Fig 13 Logo de Angular.....	26
Fig 14 Logo de Figma.....	26
Fig 15 Logo de Postman.....	27
Fig 16 Logo de Git.....	27
Fig 17 Logo de GitHub.....	27
Fig 18 Logo de Express.js.....	28
Fig 19 Logo de WebStorm.....	28
Fig 20 Logo de MySQL.....	29
Fig 21 Logo de MySQL Workbench.....	29
Fig 22 Logo de PL/SQL.....	29
Fig 23 Logo de Bootstrap.....	30
Fig 24 Logo de Handlebars.js.....	30
Fig 25 Logo de SASS-SCSS.....	30
Fig 26 Logo de JSON Web Token.....	31
Fig 27 Logo de Swagger.....	31
Fig 28 Logo de Swagger UI.....	31
Fig 29 Logo de JsDoc.....	32
Fig 30 Logos de Mocha, Chai y Supertest.....	32
Fig 31 Logo de Markdown.....	33

Fig 32 Diagrama de entidad - relación (diagrama de Chen).....	34
Fig 33 Diagrama de entidad-relación (diagrama de estructura de datos).	35
Fig 34 Herencia de actores del diagrama de casos de uso.....	36
Fig 35 Diagrama de casos de uso del usuario.	36
Fig 36 Diagrama de casos de uso del administrador.....	37
Fig 37 Casos de uso del paciente.	37
Fig 38 Casos de uso del especialista.....	37
Fig 39 Estructura general del proyecto.	38
Fig 40 Estructura del directorio 'database'.	38
Fig 41 Estructura directorio 'docs'.	38
Fig 42 Estructura del directorio 'frontend'.....	39
Fig 43 Subdirectorios del directorio 'core'.	39
Fig 44 Subdirectorios 'enviroments', 'pages' y 'shared'.	40
Fig 45 Estructura del directorio 'server'.	40
Fig 46 Estructura de los subdirectorios 'helpers', 'routes' y 'util'.	41
Fig 47 Formulario de login de la aplicación.....	42
Fig 48 Error en la introducción del email.	43
Fig 49 Generación del formulario reactivo para el login.....	43
Fig 50 Ejemplo de validador: Validador para emails.....	43
Fig 51 Método encargado de gestionar el login en el cliente.	44
Fig 52 Método encargado de comunicarse con el backend y llevar a cabo el login.	44
Fig 53 Interceptor para el login del usuario.	44
Fig 54 Ruta del servidor para el login.....	45
Fig 55 Función de validación.	45
Fig 56 Ejemplo de una petición fallida utilizando Postman.	46
Fig 57 Método de login del controlador de usuarios.	47
Fig 58 Estructura del método de login del servicio.....	47
Fig 59 Función de utilidad para la creación de un pool de conexiones a la base de datos. ..	47
Fig 60 Método de búsqueda de un usuario en la base de datos a través del email.....	48
Fig 61 Inicio de sesión fallido.	48
Fig 62 Función encargada de la firma del token de acceso.	48
Fig 63 Función encargada de la firma del token de refresco.	48
Fig 64 Ejemplo de respuesta desde el servidor ante un login correcto.....	49
Fig 65 Almacenamiento local del navegador.	49

Fig 66 Pantalla de opciones del usuario en función del rol.....	50
Fig 67 Ejemplo de ruta protegida en Angular.	50
Fig 68 Guardia de rol de paciente.	50
Fig 69 Método del servicio de autenticación para obtener el rol del usuario.	51
Fig 70 Ejemplo de protección de rutas en Node.JS.....	51
Fig 71 Funcionalidad para la verificación del token de acceso.	52
Fig 72 Mecanismo que impide que un paciente (role 2) pueda acceder a datos ajenos a los de su cuenta.....	52
Fig 73 Middleware para la verificación del rol del usuario.	53
Fig 74 Middleware para la verificación del identificador del usuario.....	53
Fig 75 Generación de la cabecera Authorization en el cliente.....	54
Fig 76 Captura de las herramientas de desarrollador donde se puede ver la cabecera Authorization.	54
Fig 77 Método principal del interceptor encargado del refresco de token.....	55
Fig 78 Método encargado de gestionar el error 401.	55
Fig 79 Método encargado de reciclar la petición original que provocó el 401.....	55
Fig 80 Método del servicio de autenticación encargado de solicitar un token de refresco..	56
Fig 81 Enrutador para el refresco del token.	56
Fig 82 Método del controlador encargado del refresco del token.....	57
Fig 83 Método del modelo encargado de devolver el token de refresco.....	57
Fig 84 Método del servicio encargado de realizar el proceso de refresco del token.	58
Fig 85 Formulario de recuperación de contraseña.	58
Fig 86 Mensaje de confirmación de envío del email.....	59
Fig 87 Servicio del cliente encargado de la funcionalidad de reinicio de contraseña.	59
Fig 88 Controlador encargado del reinicio de contraseña.....	60
Fig 89 Método del servicio de Token encargado de crear el token de reinicio.	60
Fig 90 Servicio encargado del reinicio de contraseña.....	60
Fig 91 Método encargado de enviar un correo electrónico para continuar el proceso de reinicio.....	61
Fig 92 Email de recuperación de contraseña.	61
Fig 93 Ruta para continuar el proceso.	61
Fig 94 Captura del parámetro del token.	61
Fig 95 Formulario para la recuperación de contraseña. Introducción de nueva contraseña.	62
Fig 96 Servicio de Angular encargado de continuar el reinicio de contraseña.....	62

Fig 97 Ruta del servidor que continua el reinicio de contraseña.....	62
Fig 98 Método del controlador para continuar el proceso de reinicio de contraseña.	63
Fig 99 Método del servicio para continuar el proceso de reinicio de contraseña.....	63
Fig 100 Método de utilidades para la encriptación de contraseñas.....	63
Fig 101 Configuración de CORS del servidor.....	64
Fig 102 Variables de entorno para las CORS.	64
Fig 103 Error de CORS.	64
Fig 104 Función encargada de la generación del código QR.....	65
Fig 105 Fragmento de código donde se utiliza la funcionalidad de QR.	65
Fig 106 Código QR generado y visible en el PDF de cita.	65
Fig 107 Evento para la eliminación de registros en la tabla token.	66
Fig 108 Evento para la eliminación de tomas vencidas.	66
Fig 109 Procedimiento para la eliminación de tomas vencidas.....	67
Fig 110 Configuración de Swagger.	68
Fig 111 Ruta del servidor que permitirá el acceso a Swagger UI.	68
Fig 112 Ejemplo de esquema en swagger.....	69
Fig 113 Ejemplo de comentario @swagger.	69
Fig 114 Swagger UI de la ruta GET /api/provincia.	70
Fig 115 Respuesta de la API.	70
Fig 116 Esquema de Provincia en Swagger UI.....	71
Fig 117 Archivo de configuración de JSDoc.....	71
Fig 118 Fichero namespaces.js.....	71
Fig 119 Ruta a JSDoc en app.js.	71
Fig 120 Ejemplo de comentario JSDoc.	72
Fig 121 Script de ejecución de JSDoc.	72
Fig 122 Directorio de JSDoc.....	72
Fig 123 Visualización de documentación automática en web.	73

RESUMEN

ABSTRACT

INTRODUCCIÓN

En la actualidad, la atención sanitaria, así como la gestión de su información, el permitir a los pacientes no sólo llevar a cabo un seguimiento de su historial clínico, sino que sean participantes activos del mismo, es de alta importancia en un mundo digitalizado como el que tenemos hoy en día (por ejemplo, aportando información como mediciones regulares de tensión arterial o de glucosa).

Por desgracia, no existe ninguna aplicación de referencia en cuanto a nivel de gestión clínica debido a que la mayoría de las que existen cuentan con interfaces poco intuitivas y accesibles, además de ser ineficientes, ya que sólo permiten la entrada de datos por parte del personal médico y no de los pacientes.

A la vista de esto y considerando el desarrollo web como una de las ramas más importantes dentro del sector de la tecnología, hemos decidido desarrollar una aplicación web de gestión clínica que sea usable y accesible tanto para pacientes como profesionales, y que permita además llevar una gestión del centro clínico en tema de modificación de especialidades o de personal. Esperamos que con ella podamos demostrar los conocimientos que hemos ido adquiriendo a lo largo del ciclo formativo.

Por supuesto, somos conscientes de los desafíos que acompañan al desarrollo de una aplicación de este estilo, como es mantener la seguridad y confidencialidad de los datos médicos de los usuarios y profesionales que empleen la aplicación.

OBJETIVOS

El objetivo final de cara a la presentación proyecto será crear una aplicación web que permita a un paciente solicitar cita con su especialista, así como poder visualizar los informes resultantes de las citas. No obstante, el objetivo final de la aplicación avanza más allá de la presentación al estar previsto un desarrollo posterior que aporte a la aplicación nuevas funcionalidades tales como chat, registro de alergias conocidas, posibilidad de realizar consultas por videoconferencia, etc.

A continuación, se enumerarán los objetivos previstos de cara al desarrollo de este proyecto empezando por los objetivos más básicos relativos a la planificación y diseño de la aplicación web de gestión clínica y escalando hasta llegar a los puntos de programación y desarrollo de esta.

1. Desarrollar una aplicación web funcional e intuitiva.
2. Aprender la sintaxis y funcionamiento de Angular y Express.js.
3. Asegurar el correcto funcionamiento en los 3 estándares de pantalla actuales: monitor, Tablet y móvil.
4. Definir un proceso de registro de usuarios para que se puedan dar de alta en la clínica.
5. Permitir a un usuario administrador el poder gestionar a los usuarios y la información visible en las secciones de “Especialidades” y de “Especialistas”.
6. Definir un proceso de inicio de sesión que permita realizar diferentes funciones en relación con el rol del usuario.
7. Realizar un sistema que permita a un paciente poder subir sus mediciones de tensión arterial y glucosa permitiendo de esta manera al especialista tener un seguimiento sobre ello.
8. Realizar un sistema que permite a un paciente poder visualizar la medicación que tiene asignada, así como al especialista la posibilidad de editar, añadir o eliminar medicación a este listado.
9. Permitir a un especialista poder añadir nuevos medicamentos a la base de datos.

ALCANCE DEL PROYECTO

1. REQUISITOS FUNCIONALES Y NO FUNCIONALES

Funcionales:

1. Diseñar un formulario de registro, guardando los datos de forma segura.
2. Implementar un sistema de autenticación de usuarios basándose en el usuario y contraseña indicados en el formulario de registro.
3. Mostrar en la sección “Especialidades” todas las especialidades disponibles en la clínica.
4. Mostrar en la sección “Especialistas” todos los especialistas de la clínica organizados por especialidad.
5. Poder pedir citas con los especialistas sólo si se ha iniciado sesión previamente.
6. Permitir a un paciente el poder ver sus medicaciones y su pauta de toma.
7. Permitir a un especialista asignar medicamentos a sus pacientes.
8. Dar la posibilidad a un paciente de subir sus mediciones de tensión arterial y glucosa.
9. Permitir a un especialista llevar el seguimiento de las mediciones realizadas por los pacientes.
10. Conceder a un paciente el poder ver los informes médicos escritos por los especialistas.
11. Autorizar a un especialista el poder escribir el informe de su paciente.
12. Conceder privilegios de administración a un usuario para que pueda gestionar las cuentas de usuario, así como la información visible en las secciones principales de la web.

No funcionales:

1. El desarrollo se realizará utilizando los *frameworks* de Angular para toda la lógica relativa al front-end y de Express.js para la lógica relativa el *backend*.
2. La interfaz debe ser amigable y fácil de utilizar.
3. La interfaz debe ser responsiva pudiéndose adaptar a monitores, tablets y móviles.
4. El código deberá estar correctamente modularizado para poder mejorar su mantenimiento y escalabilidad.
5. El código estará correctamente documentado para facilitar su comprensión.
6. El sistema debe estar disponible en cualquier momento del día.

7. Los pacientes se podrán registrar de forma autónoma pero los especialistas necesitarán de un registro por parte del administrador.
8. Los datos asociados a registros, informes y mediciones serán almacenados de forma segura en la base de datos.

2. PROTOTIPO / MOCKUP

A. Navegación a través de la barra de navegación

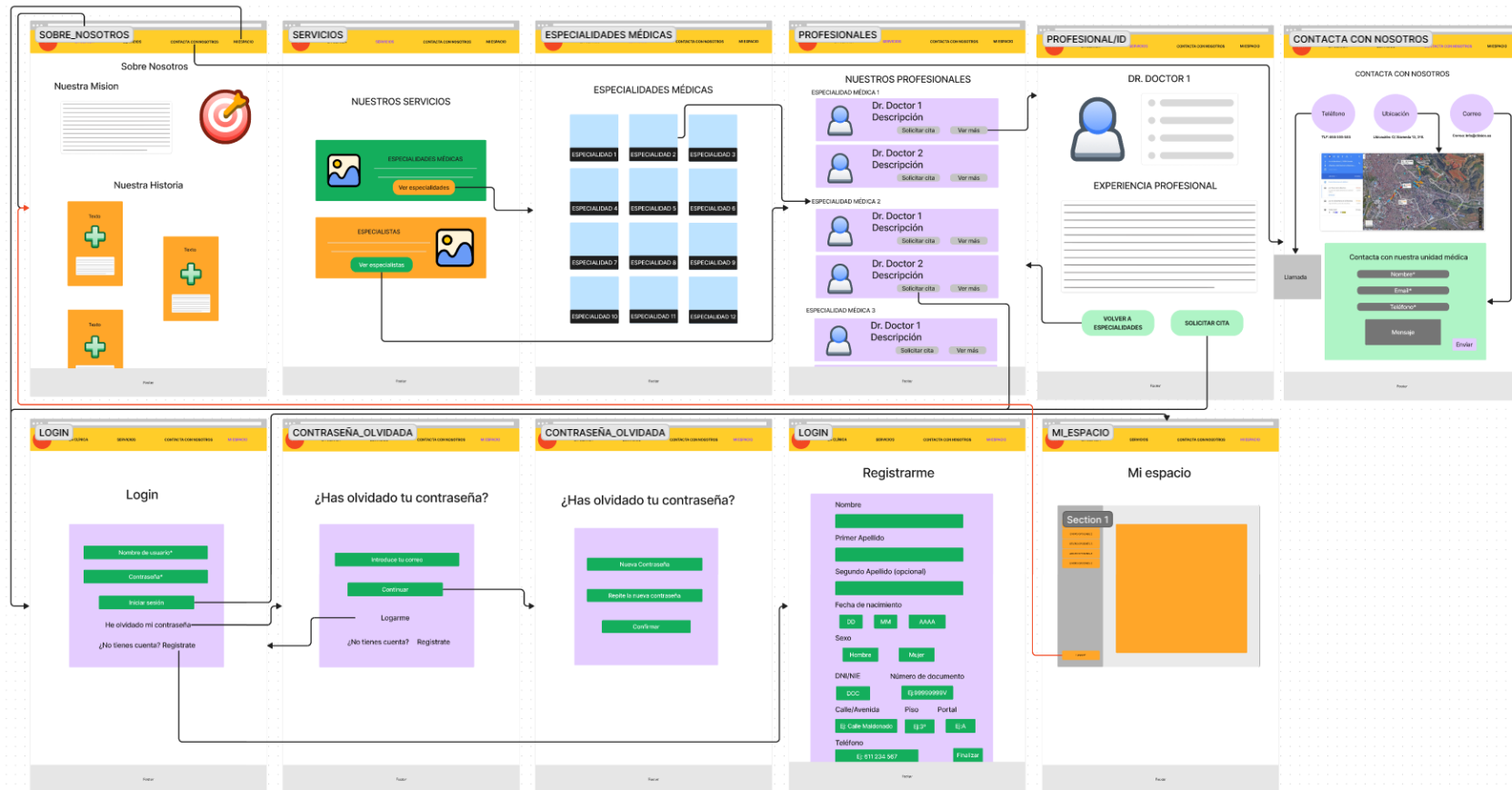


Fig 1 Prototipo de la navegación a través de la barra de navegación de la web.

B. Mi Espacio - Rol Administrador

1) Gestión de usuarios

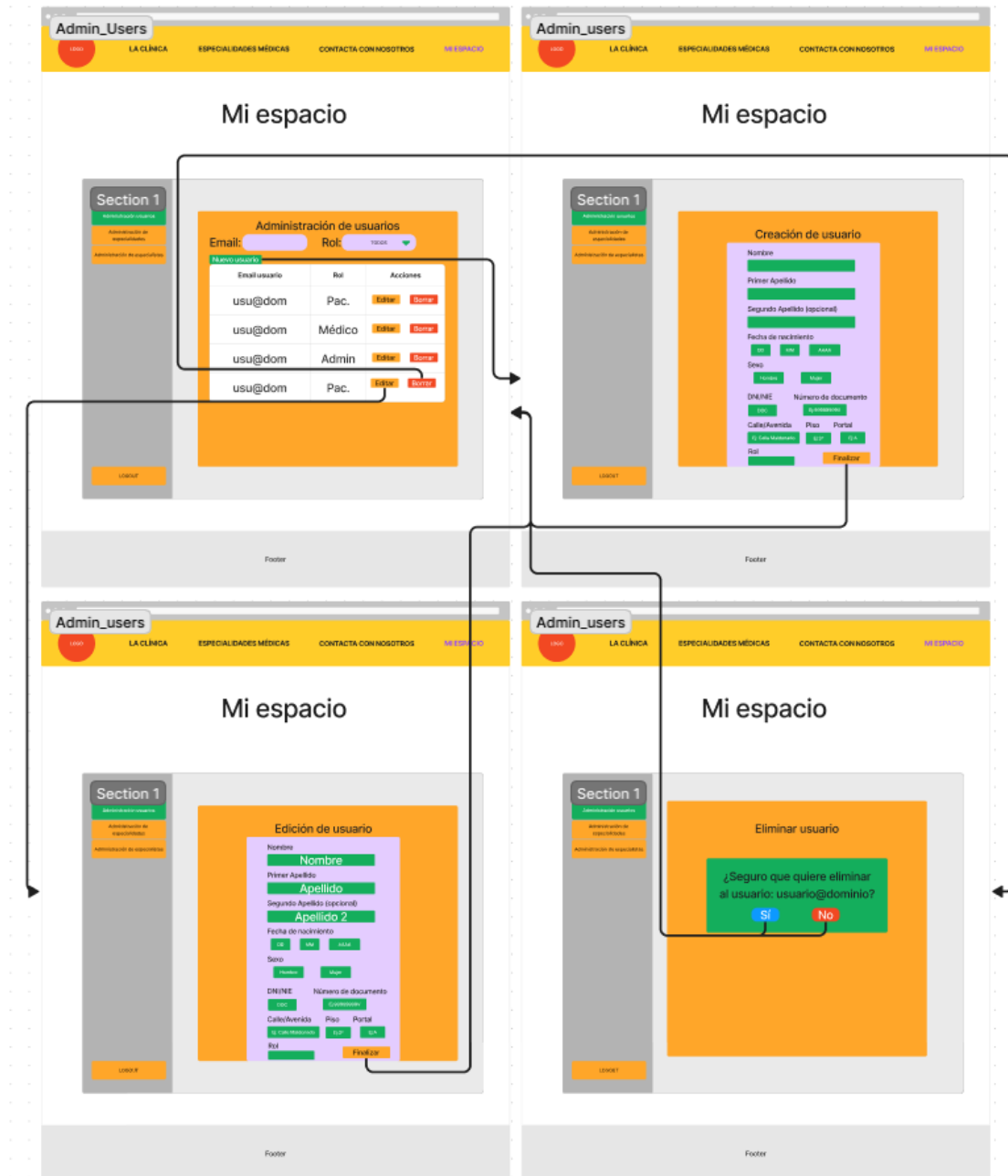


Fig 2 Prototipo para la gestión de usuarios por parte del administrador.

2) Gestión de especialidades

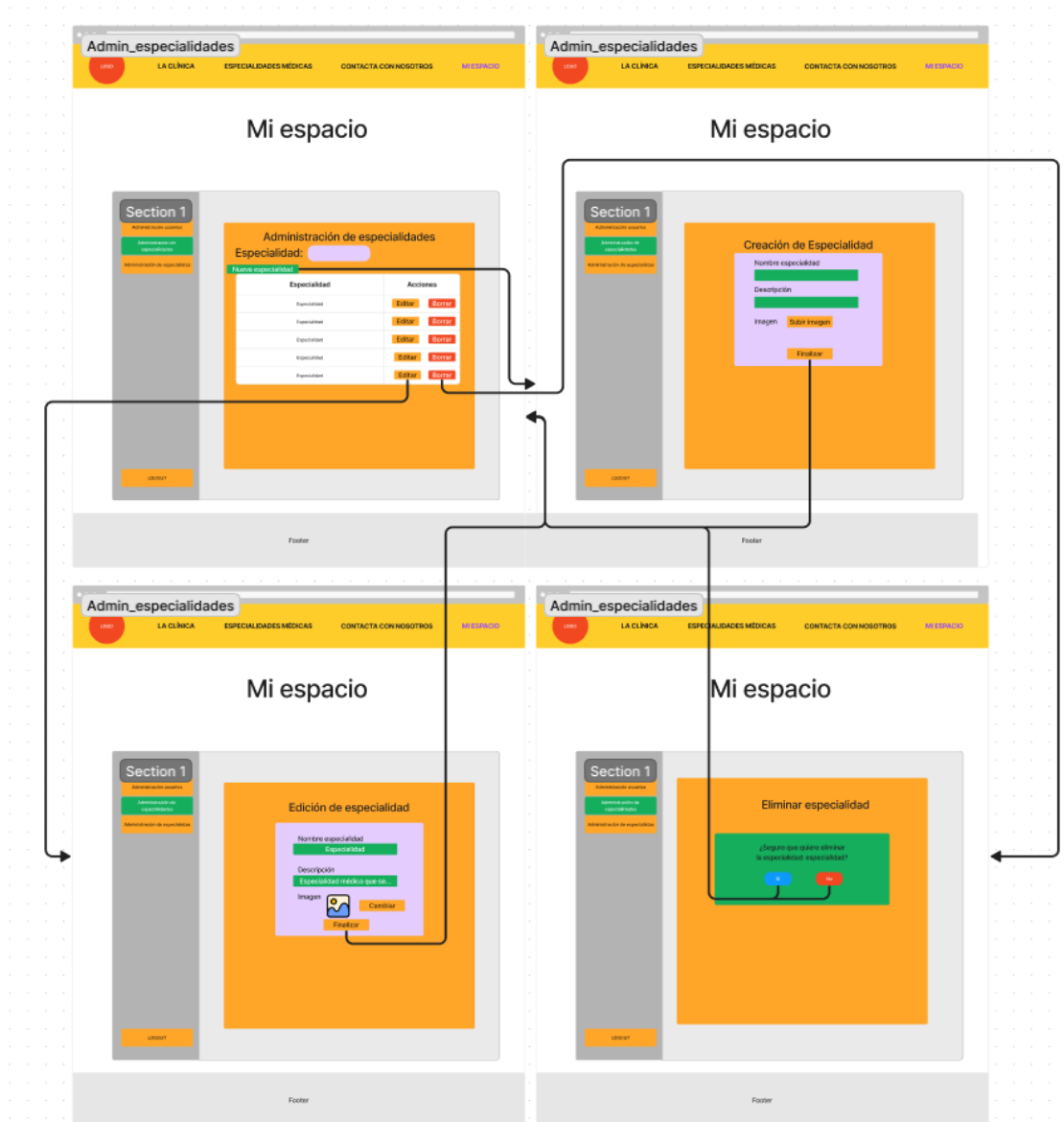


Fig 3 Prototipo para la gestión de especialidades por parte del administrador.

3) Gestión de especialistas

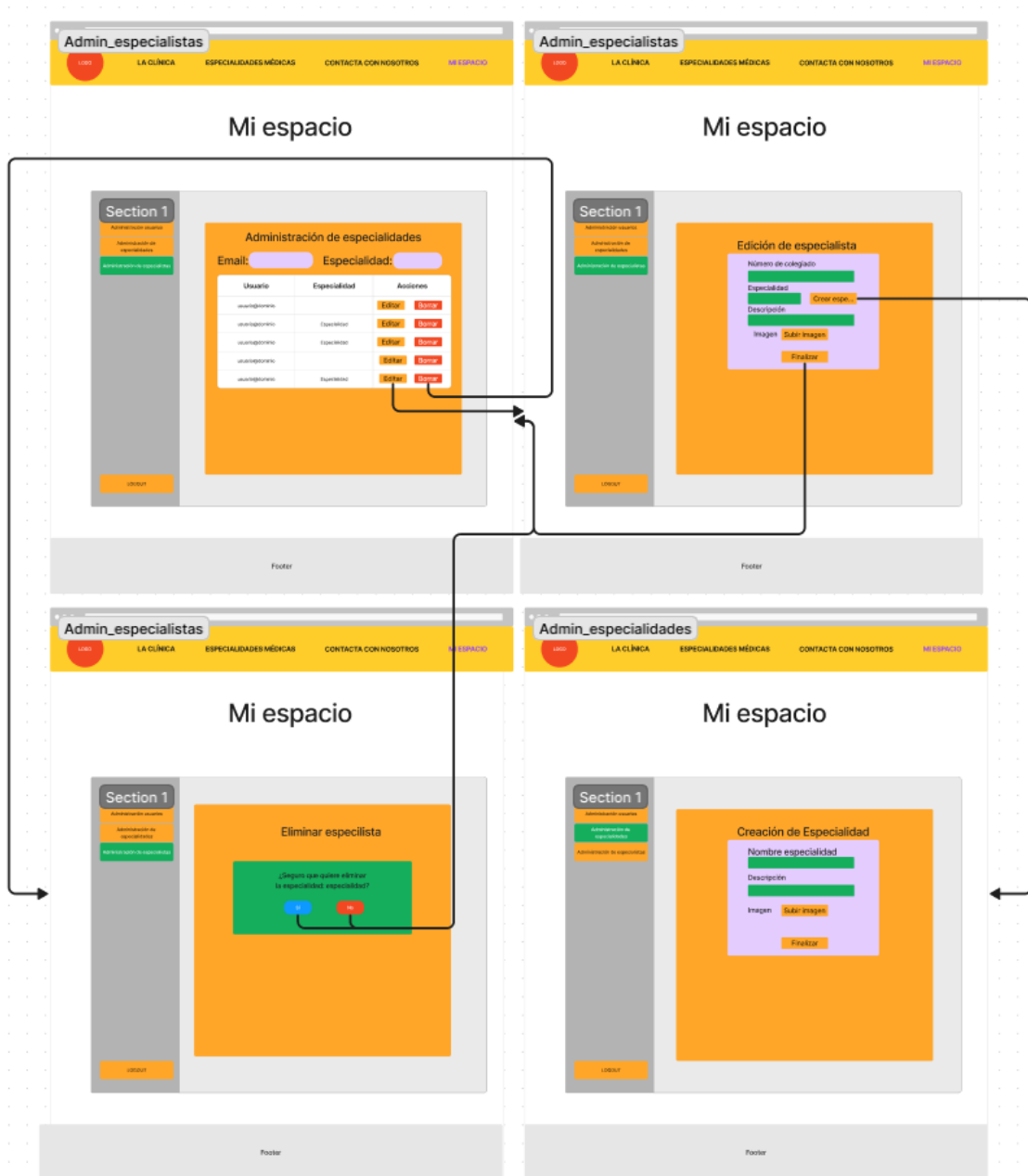


Fig 4 Prototipo para la gestión de especialistas por parte del administrador.

C. Mi Espacio - Rol Especialista

1) Consultar agenda diaria



Fig 5 Prototipo para la consulta de la agenda diaria por parte del especialista.

2) Gestión historia clínica - Medicación

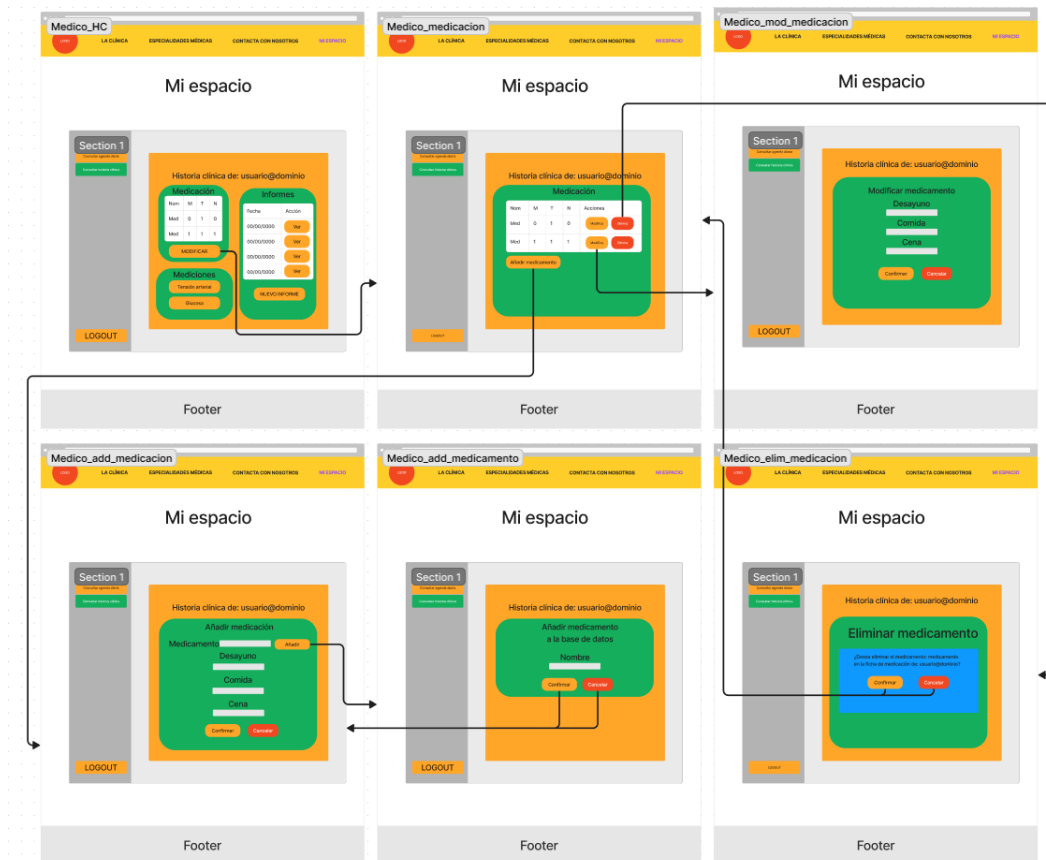


Fig 6 Prototipo para la gestión de la medicación por parte del especialista.

3) Gestión historia clínica - Mediciones de glucosa y tensión arterial

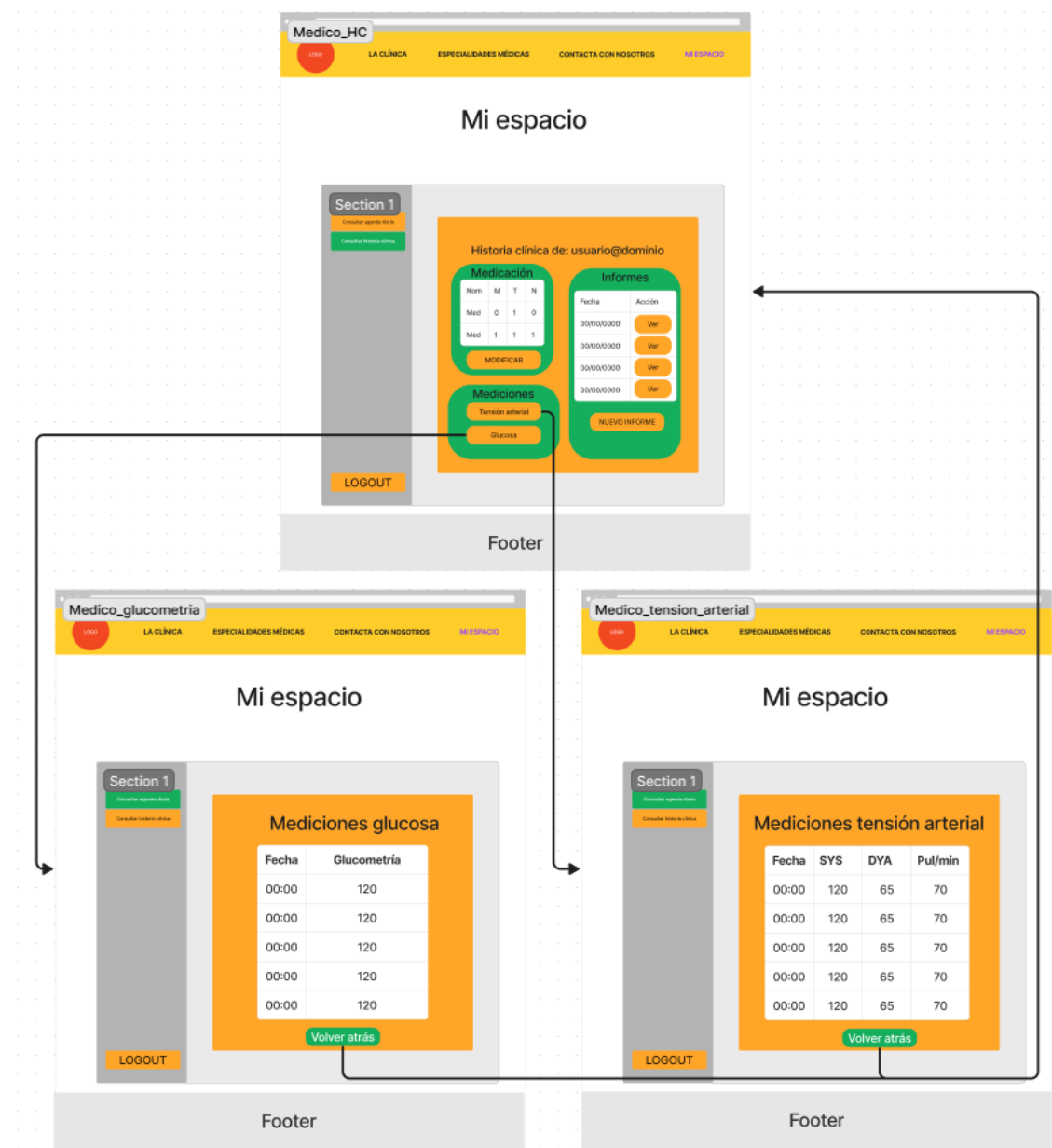


Fig 7 Prototipo para consulta de las mediciones de glucosa y tensión arterial del paciente.

4) Gestión historia clínica - Informes médicos

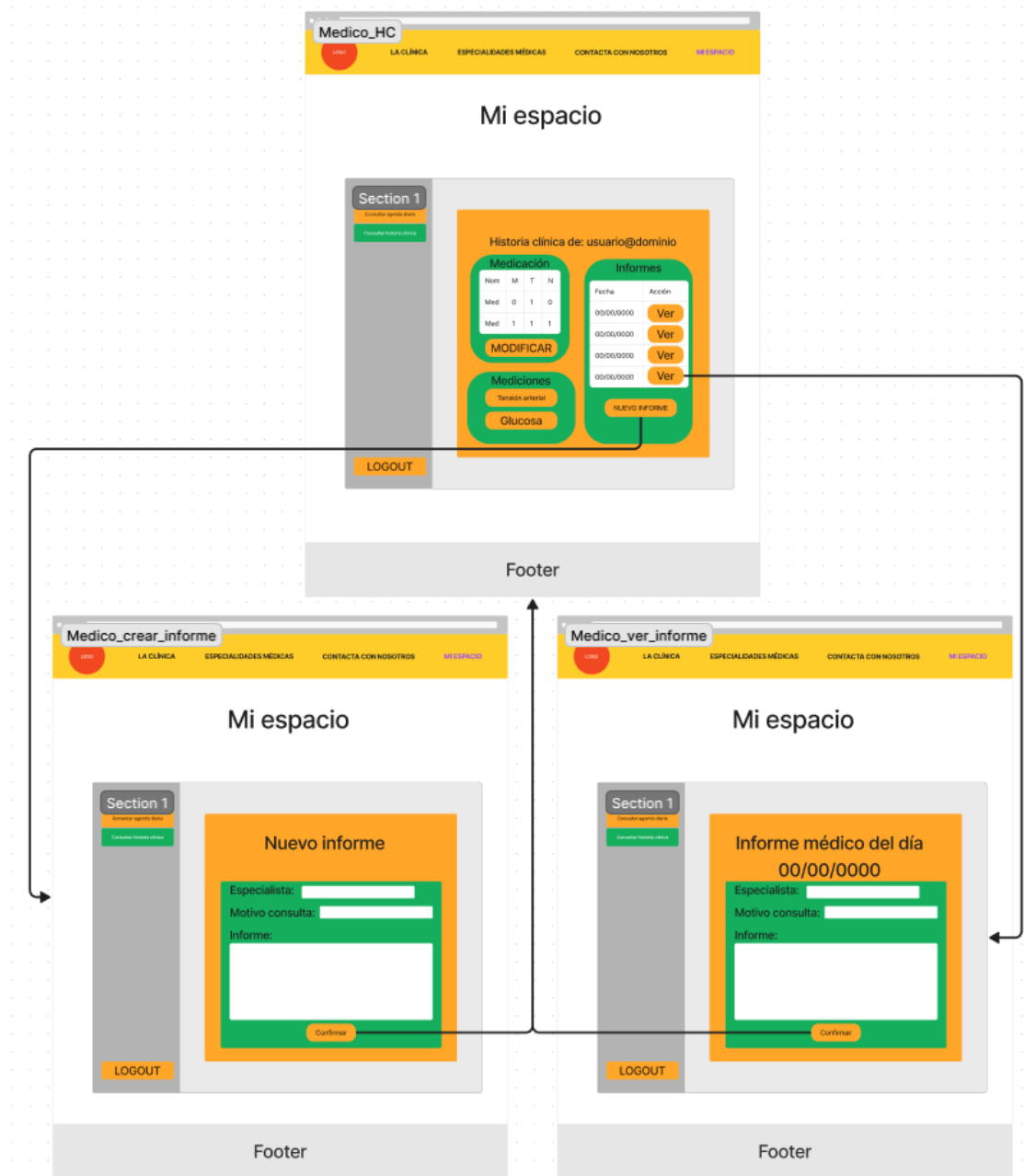


Fig 8 Prototipo para la consulta y generación de nuevos informes médicos.

D. Mi Espacio - Rol Paciente

1) Solicitar cita

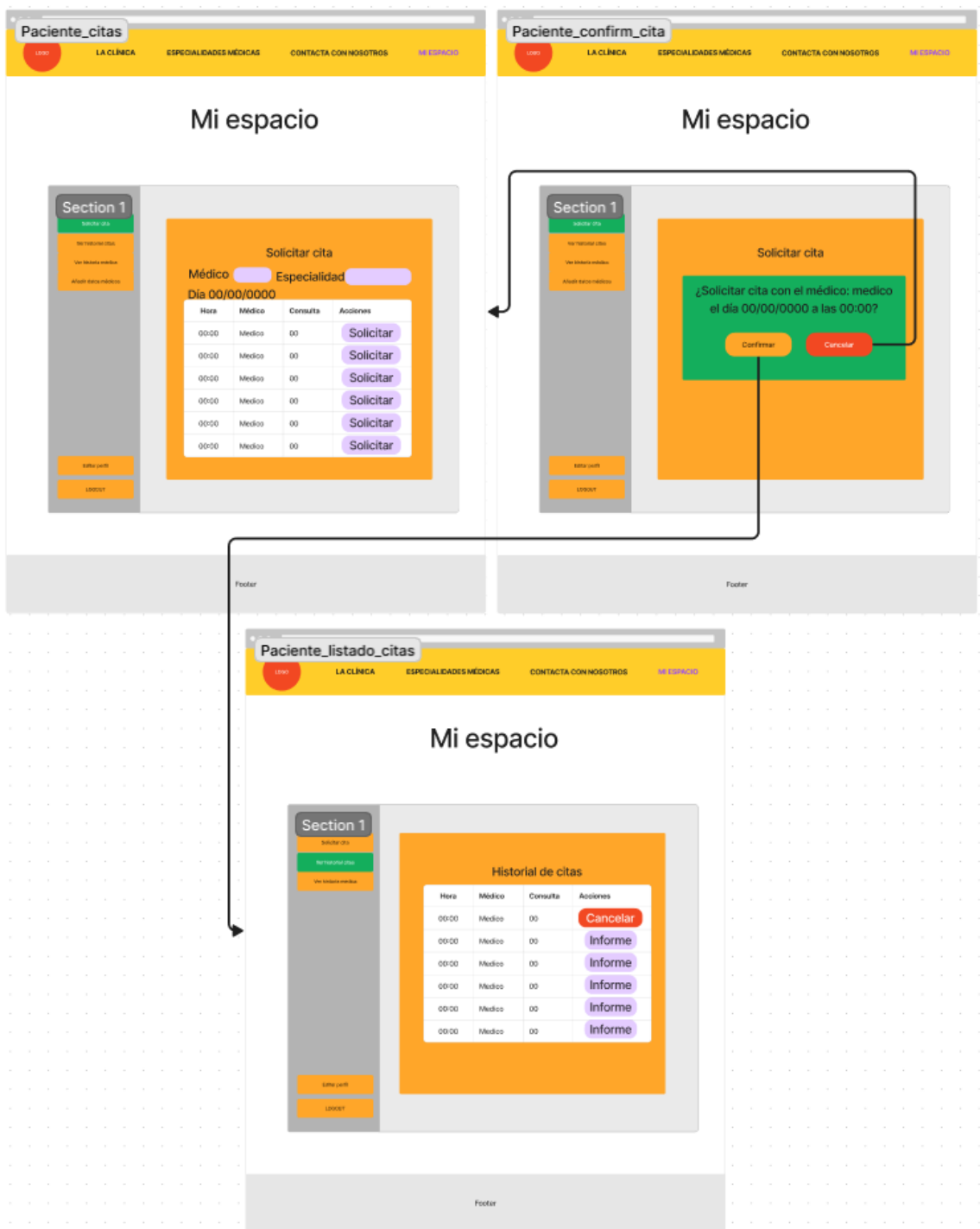


Fig 9 Prototipo para la solicitud de citas con especialistas por parte del paciente.

2) Historial de citas

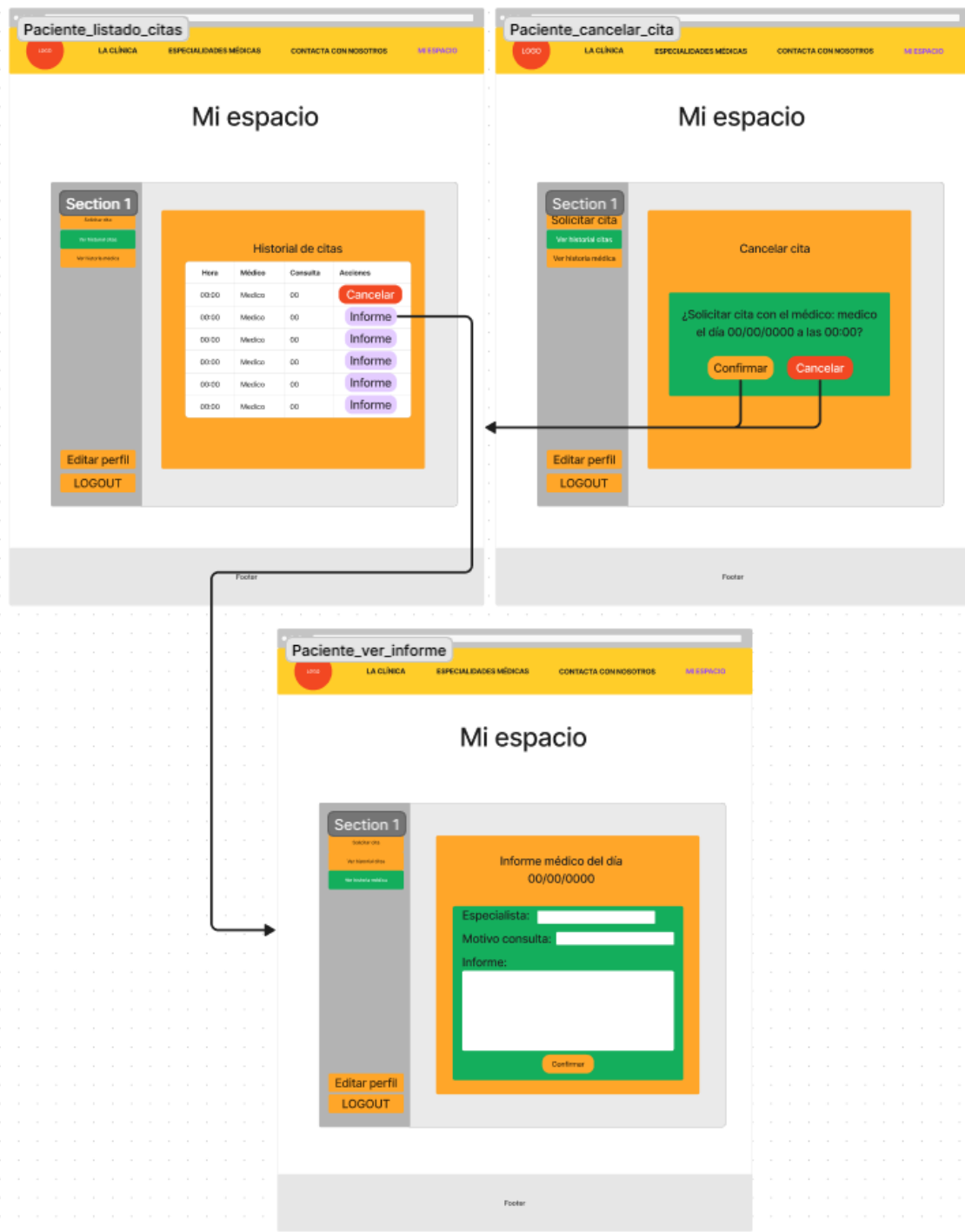


Fig 10 Prototipo para la gestión de citas por parte del paciente.

3) Historial clínico

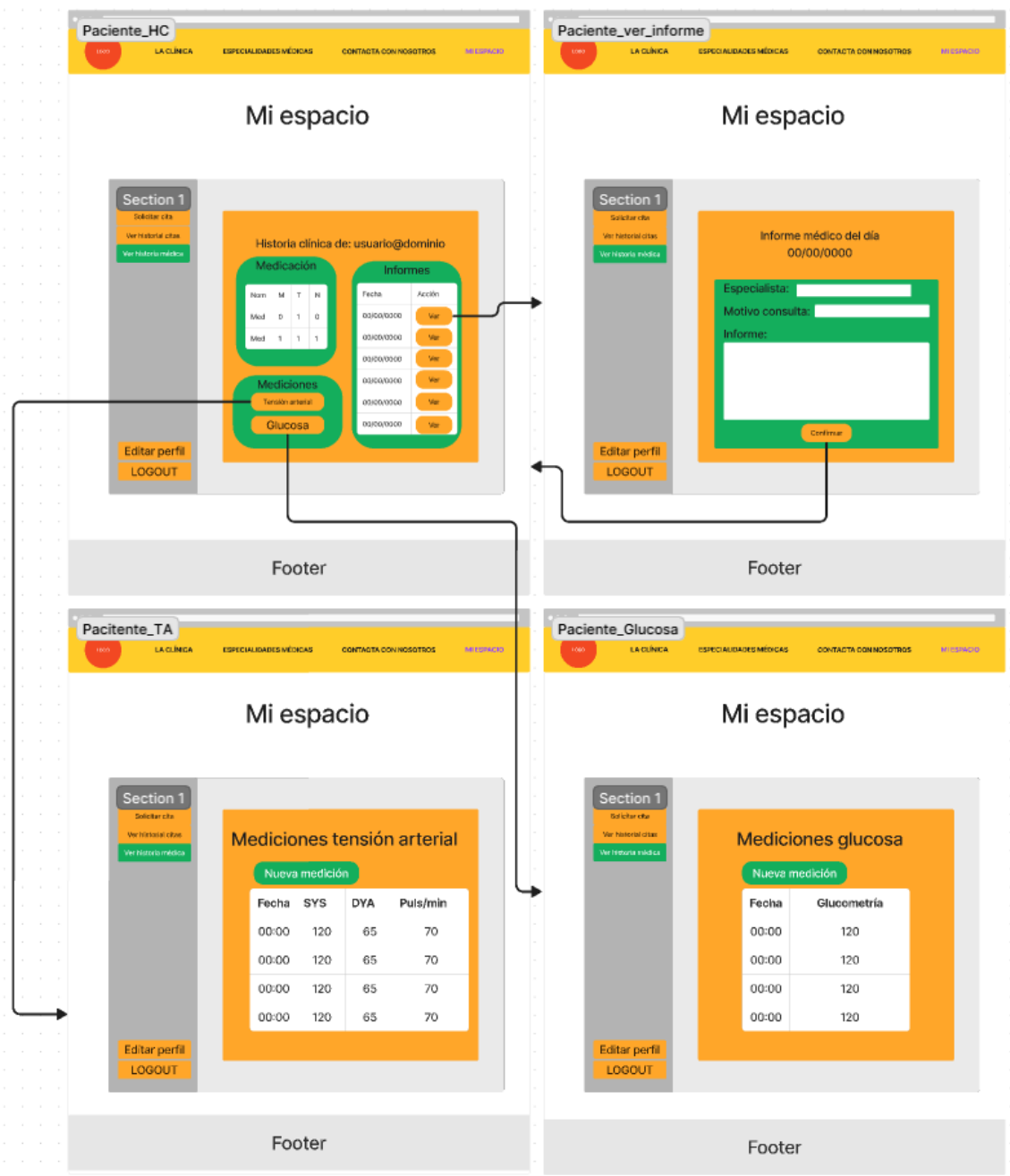


Fig 11 Prototipo para la inserción de datos de mediciones de tensión arterial y glucosa por parte del paciente.

4) Editar perfil

El prototipo muestra una interfaz web para la edición del perfil de un paciente. La estructura es la siguiente:

- Encabezado (Header):** Incluye el título "Paciente_perfil" y un menú de navegación con los enlaces: "LA CLÍNICA", "ESPECIALIDADES MÉDICAS", "CONTACTA CON NOSOTROS" y "MI ESPACIO".
- Cuerpo Principal:**
 - Título:** "Mi espacio".
 - Sección 1 (Lateral izquierdo):** Contiene los enlaces "Solicitar cita", "Ver historial citas" y "Ver historia médica".
 - Formulario de Edición (Central):** Un recuadro con fondo naranja que contiene:
 - Título: "Perfil de: usuario@dominio".
 - Campos de texto: "Nombre:", "Apellido:", "Dirección:" y "Contraseña:".
 - Botón: "Actualizar" (verde).
 - Botones de Acción (Lateral izquierdo inferior):** "Editar perfil" (verde) y "LOGOUT" (naranja).
- Pie de Página (Footer):** Un recuadro gris con el texto "Footer".

Fig 12 Prototipo para la edición del perfil por parte del paciente.

3. TECNOLOGÍAS USADAS

A. Angular

Angular es un *framework* de desarrollo de aplicaciones web creado por Google, diseñado para facilitar la creación de aplicaciones dinámicas y de una sola página (SPA). Angular se basa en el paradigma de arquitectura de componentes, lo que significa que las aplicaciones se construyen mediante la composición de componentes reutilizables. Estos componentes encapsulan la funcionalidad y la interfaz de usuario de la aplicación, lo que facilita la organización del código y el mantenimiento a medida que la aplicación crece en complejidad.



Fig 13 Logo de Angular.

Hemos utilizado esta tecnología ya que hoy en día es uno de los *frameworks* más utilizados a nivel de desarrollo en cuanto a la parte de *frontend*. Además, debido a su arquitectura de modelo-vista-presentador (MVP) igual que la gran cantidad de funciones que trae incluidas como el enlazado de datos bidireccional, inyección de dependencias, enrutados y servicios entre otros, permite conseguir de manera sencilla, una aplicación escalable, mantenible y eficiente.

B. Figma

Figma es una herramienta de diseño de interfaces de usuario (UI) basada en la nube que permite a los diseñadores crear, colaborar y compartir diseños de aplicaciones web y móviles de manera eficiente. Es una aplicación todo en uno que abarca desde la creación de *wireframes* y prototipos hasta el diseño visual y la generación de especificaciones de diseño.

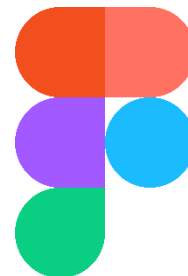


Fig 14 Logo de Figma.

Hemos decidido usar esta tecnología para el maquetado de nuestro sitio web debido a la posibilidad de colaboración en tiempo real remota que ofrece. A su vez, al estar en la nube y ser compatible con diferentes sistemas operativos permite una gran accesibilidad, lo cual junto a su diseño amigable facilita su uso e implementación.

C. Postman

Es una plataforma que permite diseñar, probar, documentar y monitorear interfaces de programación de aplicaciones (APIs) de manera eficiente. A través de la interfaz gráfica de usuario permite la creación y envío de diferentes tipos de solicitudes HTTP (GET, POST, PUT, DELETE...) así como características adicionales como las cabeceras que se van a enviar, el contenido del cuerpo, etc. a APIs. Además, permite otras características más avanzadas como la automatización de pruebas o la generación de documentación automática.



Fig 15 Logo de Postman.

Hemos decidido utilizar esta tecnología porque permite comprobar el funcionamiento de nuestra API REST de una forma rápida y sencilla y sin requerir de tener un *frontend* funcional para realizar solicitudes y ver el resultado de estas.

D. Git

Es un sistema de control de versiones distribuido que permite llevar un registro de los cambios realizados en el código fuente de un proyecto a lo largo del tiempo. Gracias a Git los desarrolladores pueden realizar un seguimiento de las modificaciones realizadas en un proyecto, así como revertir cambios anteriores si es necesario facilitando y haciendo más efectivo el desarrollo colaborativo con otros miembros del equipo de desarrollo.



Fig 16 Logo de Git.

Además, su modelo descentralizado permite que cada desarrollador tenga una copia completa en su propio sistema del historial de cambios que ha ido sufriendo el proyecto, permitiendo de esa manera un flujo de trabajo flexible e independiente que posibilita la sincronización de cambios con el repositorio central cuando sea necesario.

E. GitHub

Es una plataforma de desarrollo colaborativo basada en la nube que utiliza Git como control de versiones. Permite a los desarrolladores almacenar, gestionar y colaborar en proyectos de software de manera eficiente al trabajar directamente sobre un repositorio remoto alojado en la nube.

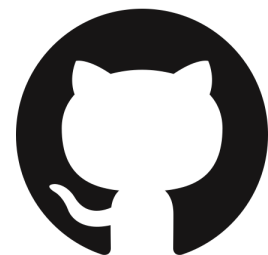


Fig 17 Logo de GitHub.

La plataforma ofrece además una variedad de herramientas y características como por ejemplo el seguimiento de problemas, la gestión de proyectos, la revisión de código o la integración y el despliegue continuo (CI/CD).

F. Express.js

Es un *framework* de desarrollo de aplicaciones web para Node.js que a su vez se trata de un entorno de ejecución de JavaScript en el lado del servidor. Destaca principalmente por ser ligero y flexible.



Fig 18 Logo de Express.js.

Express.js simplifica la creación de servidores web basados en Node.js al proporcionar una capa de abstracción sobre el servidor HTTP nativo de Node.js, facilitando de esa manera la definición de rutas, el manejo de solicitudes y respuestas, el manejo del *middleware* (función que es ejecutada entre la recepción de una solicitud HTTP y el envío de la respuesta por parte del servidor) y la configuración de la aplicación.

Además, Express.js es altamente extensible, lo que permite integrar multitud de bibliotecas para agregar funcionalidades adicionales cuando sea necesario.

Hemos decidido utilizar esta tecnología porque queríamos aprovechar el trabajo de fin de ciclo para aprender el funcionamiento de una tecnología de *backend* diferente a la vista durante el curso. Eso junto con sus características de alta velocidad y rendimiento, su facilidad de escalabilidad y flexibilidad nos hizo decantarnos por utilizar Node.js junto con Express.js en el *backend*.

G. WebStorm

Es un potente entorno de desarrollo integrado (IDE) desarrollado por la compañía JetBrains y que está diseñado específicamente para el desarrollo de aplicaciones web que utilizan tecnologías como HTML, CSS, JavaScript y Typescript, así como *frameworks* relacionados como Angular, React, NestJS o Express.js.



Fig 19 Logo de WebStorm.

Proporciona un conjunto de herramientas que ayudan a escribir, editar, depurar y refactorizar código de una manera eficiente, rápida y simple.

Entre sus características principales destacan su autocompletado inteligente, su análisis de código estático, la depuración integrada, la fácil integración con sistemas de control de versiones y el soporte para *frameworks* y bibliotecas de código.

H. MySQL

MySQL es un sistema de gestión de bases de datos relacional (organiza los datos en tablas relacionadas entre sí) de código abierto ampliamente utilizado en todo el



mundo. Ofrece una sólida combinación de rendimiento, confiabilidad y facilidad de uso, lo que lo convierte en una opción popular en el desarrollo de aplicaciones.

Fig 20 Logo de MySQL.

Hemos utilizado esta base de datos debido a la escalabilidad que permite al poder manejar un gran volumen de datos al igual que de cantidad de usuarios concurrentes se refiere. Además, ofrece una amplia gama de características de seguridad como la encriptación de datos, autenticación de usuarios y permisos, lo que permite que sea una base de datos muy versátil y fácil de integrar en diferentes entornos.

I. MySQL Workbench

MySQL Workbench es una herramienta gráfica de diseño y administración de bases de datos que se utiliza junto con MySQL para simplificar tareas de desarrollo y administración.



Lo hemos utilizado debido a que ofrece una interfaz intuitiva que permite a los desarrolladores y administradores de bases de datos realizar diversas tareas como diseño de esquemas, consulta y manipulación de datos, y optimización de consultas de manera gráfica, entre otras.

Fig 21 Logo de MySQL Workbench.

J. Procedural Language / Structured Query Language (PL/SQL)

PL/SQL es un lenguaje de programación procedimental que sirve como extensión del estándar SQL y permite incluir capacidades de programación procedural. Con P/-SQL los desarrolladores de bases de datos pueden escribir bloques de código que pueden realizar diversas acciones tales como la manipulación de datos o el control de flujo de ejecución.



Fig 22 Logo de PL/SQL.

Hemos decidido usar esta tecnología ya que con ella podemos conseguir automatizar procesos en la base de datos de nuestra aplicación, por ejemplo, generar eventos que a una hora determinada del día lleven a cabo el truncado de una tabla o que llamen a un determinado procedimiento que lleve a cabo borrados secuenciales de datos que ya no sean necesarios como por ejemplo prescripciones de medicamentos que ya no están activas al haber superado la fecha de finalización del tratamiento.

K. Bootstrap

Bootstrap es un popular *framework* de código abierto para desarrollo *frontend*, utilizado para crear interfaces web y aplicaciones con mayor rapidez y eficiencia.



Fig 23 Logo de Bootstrap.

Hemos decidido utilizarlo ya que ofrece una gran variedad de componentes y estilos predefinidos que pueden ser utilizados directamente en el sitio web como son botones, formularios, modales y barras de navegación entre otros. A su vez, es una gran herramienta al facilitar la creación de diseños responsivos que se adaptan automáticamente al ancho de la pantalla al igual que es fácil de utilizar ya que se usan clases CSS intuitivas para aplicar los estilos.

L. Handlebars.js

Handlebars.js es un motor de plantillas JavaScript que simplifica la generación de HTML al permitir la creación de plantillas de forma más organizada y eficiente.



Fig 24 Logo de Handlebars.js

Hemos utilizado esta tecnología ya que permite reutilizar fragmentos de HTML en múltiples partes de la aplicación sin necesidad de duplicar el mismo código HTML. A su vez facilita la inserción dinámica de datos en las plantillas, permitiendo vincular datos a tus plantillas y luego renderizarlas con los datos específicos, lo que es especialmente útil en aplicaciones web dinámicas donde los datos cambian frecuentemente.

M. Sassy Cascading StyleSheets (SCSS)

SCSS es una extensión de CSS y una evolución de *Syntactically Awesome Stylesheets* (SASS) que ofrece una sintaxis más avanzada y poderosa para la escritura de hojas de estilo en la web. Introduce características adicionales que no están presentes en CSS tradicional, como variables, anidamiento, mixins, herencia y operaciones matemáticas, lo que permite a los desarrolladores escribir estilos de manera más modular.



Fig 25 Logo de SASS-SCSS.

Hemos decidido añadir esta tecnología como forma de estilado de nuestro proyecto para tener una mayor flexibilidad y adaptabilidad de nuestras hojas de estilo al generar gracias a ella código de estilado menos repetitivo y más modularizado.

N. JavaScript Object Notation Web Tokens (JWT)

Los JSON Web Tokens o JWT son un pequeño paquete de información seguro y compacto que permite transmitir datos entre dos partes (*frontend* y *backend*) de forma segura. Está formado por tres partes: el encabezado que describe el tipo de token y el algoritmo de firma que se ha utilizado, la carga útil que contiene la información que se quiere transmitir y la firma que se utiliza para verificar que el mensaje no ha sido alterado por ninguna de las partes.

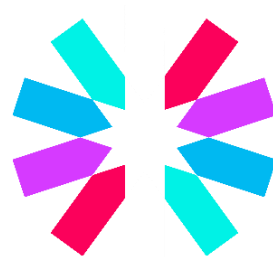


Fig 26 Logo de JSON Web Token.

Hemos decidido utilizarlo ya que se está convirtiendo actualmente en el estándar de aplicaciones web y APIs para la autenticación y transferencia de información entre cliente y servidor. En nuestro caso nos sirve principalmente como mecanismo de defensa contra la intrusión al utilizarlo como barrera de autenticación a través de *middlewares* de Node.js/Express.js.

O. Swagger

Swagger es una herramienta de código abierto que se utiliza para diseñar, construir, documentar y consumir servicios web REST. Permite describir la funcionalidad de una API de una manera clara y estructurada, utilizando un formato JSON o YAML llamado *OpenAPI Specification*.



Fig 27 Logo de Swagger.

En estos ficheros se definen *endpoints* de su API, los parámetros que aceptan, los tipos de datos que devuelven, los códigos de respuesta, entre otros detalles.

Hemos decidido utilizarlo porque es una forma sencilla y ágil de generar documentación automatizada.

P. Swagger UI

Swagger UI es una interfaz de usuario generada automáticamente a partir de los documentos JSON o YAML de Swagger. Utilizando estos documentos genera una interfaz web dinámica



Fig 28 Logo de Swagger UI.

que permite explorar y probar los *endpoints* de la API directamente desde el navegador.

Esta interfaz muestra un listado de los *endpoints* de la API junto con detalles sobre los parámetros que aceptan y los códigos de respuesta que devuelven. Así mismo permite a los usuarios enviar solicitudes HTTP a la API rellendo formularios para comprobar el comportamiento en tiempo real de esta.

Hemos decidido utilizar esta tecnología porque nos parece una manera interesante, dinámica y atractiva de mostrar la documentación de la API no sólo mostrando los datos de esta sino también permitiendo el ejecutar llamadas a la API para comprobar respuesta.

Q. JSDoc

JSDoc es una convención y una herramienta para documentar código JavaScript. Permite a los desarrolladores describir de manera clara y estructurada el comportamiento y la estructura del



Fig 29 Logo de JsDoc.

código a través de comentarios especiales en el código fuente que luego son procesados por la herramienta para generar documentación legible de forma automática a través de ficheros HTML.

Esta herramienta permite agregar anotaciones a los comentarios (`@function`, `@param`, `@return`, `@public`, `@description`, `@async...`) que proporcionan información extra sobre los tipos de datos que se esperan recibir o retornar, el tipo de función del que se trata, su modificador de acceso, etc.

La decisión de incluir esta tecnología es porque queríamos documentar no sólo las rutas de la API sino también las clases y métodos utilizados para mejorar la mantenibilidad y comprensión del código.

R. Mocha, Chai y Supertest

Mocha, Chai y Supertest son herramientas para realizar pruebas y testeos en aplicaciones Node.JS y en especial en APIs.



Fig 30 Logos de Mocha, Chai y Supertest.

- a) **Mocha**: Es un framework de pruebas unitarias y de integración para JS diseñado para ser simple, flexible y fácil de usar. Mocha proporciona una estructura para escribir y ejecutar pruebas de forma organizada permitiendo definir suites de pruebas, casos de prueba, etc.

- b) **Chai**: Es una biblioteca de aserciones para Node.JS que permite la utilización de *expect*, *should*, *assert*... y, de esa forma, posibilitar elegir el estilo con el que se prefieren escribir las pruebas de una manera legible y comprensible.
- c) **Supertest**: Es una biblioteca de pruebas HTTP para Node.JS que se utiliza principalmente para realizar pruebas de integración APIs RESTful. Permite a los desarrolladores realizar solicitudes HTTP a su API y realizar aserciones sobre las respuestas recibidas, facilitando de esa forma la automatización de pruebas extremo a extremo y la verificación del comportamiento correcto de la API.

En conjunto son herramientas con un alto potencial y proporcionan un conjunto de funcionalidades para escribir, organizar y ejecutar pruebas en aplicaciones JS y APIs.

S. Markdown

Markdown es un lenguaje de marcado ligero y una herramienta para la creación de texto con formato utilizando una sintaxis fácil de leer y escribir. Es ampliamente utilizado para escribir documentación, archivos README, blogs, y otros tipos de contenido textuales que requieren formato.



Fig 31 Logo de Markdown.

Markdown se destaca por su simplicidad y la facilidad con la que se puede convertir en HTML u otros formatos para su visualización en la web. Los archivos Markdown pueden ser procesados por diversas herramientas y editores para generar documentación visualmente atractiva.

La decisión de incluir Markdown en el proyecto se debe a su utilidad para la creación de documentación clara y estructurada, en nuestro caso lo hemos utilizado para crear un archivo README bien formateado que explique el propósito, la constitución y la configuración del proyecto como página principal de nuestro repositorio remoto.

4. DIAGRAMAS DE ENTIDAD-RELACIÓN

A. Diagrama de Chen

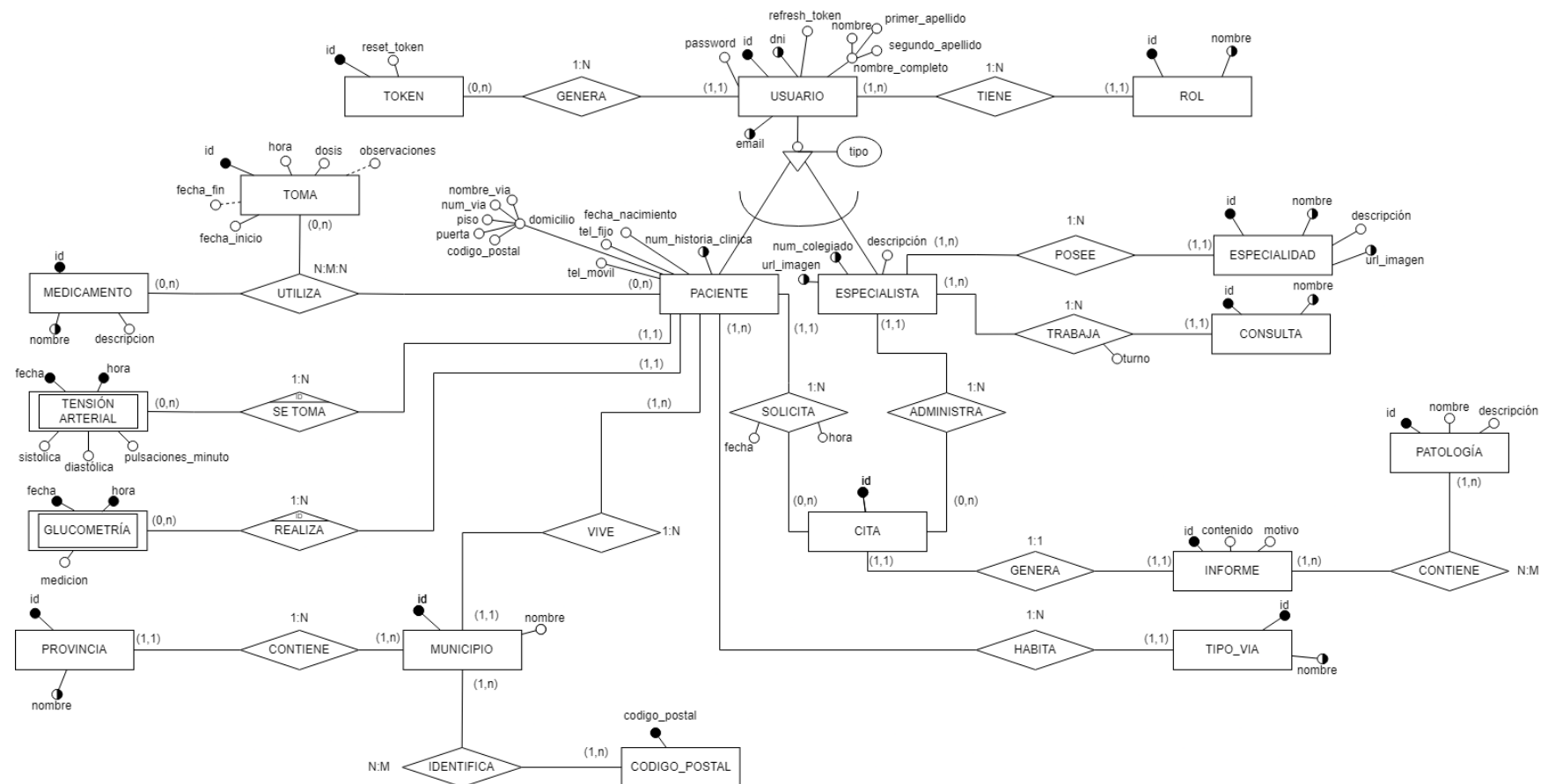


Fig 32 Diagrama de entidad - relación (diagrama de Chen).

B. Diagrama de estructura de datos

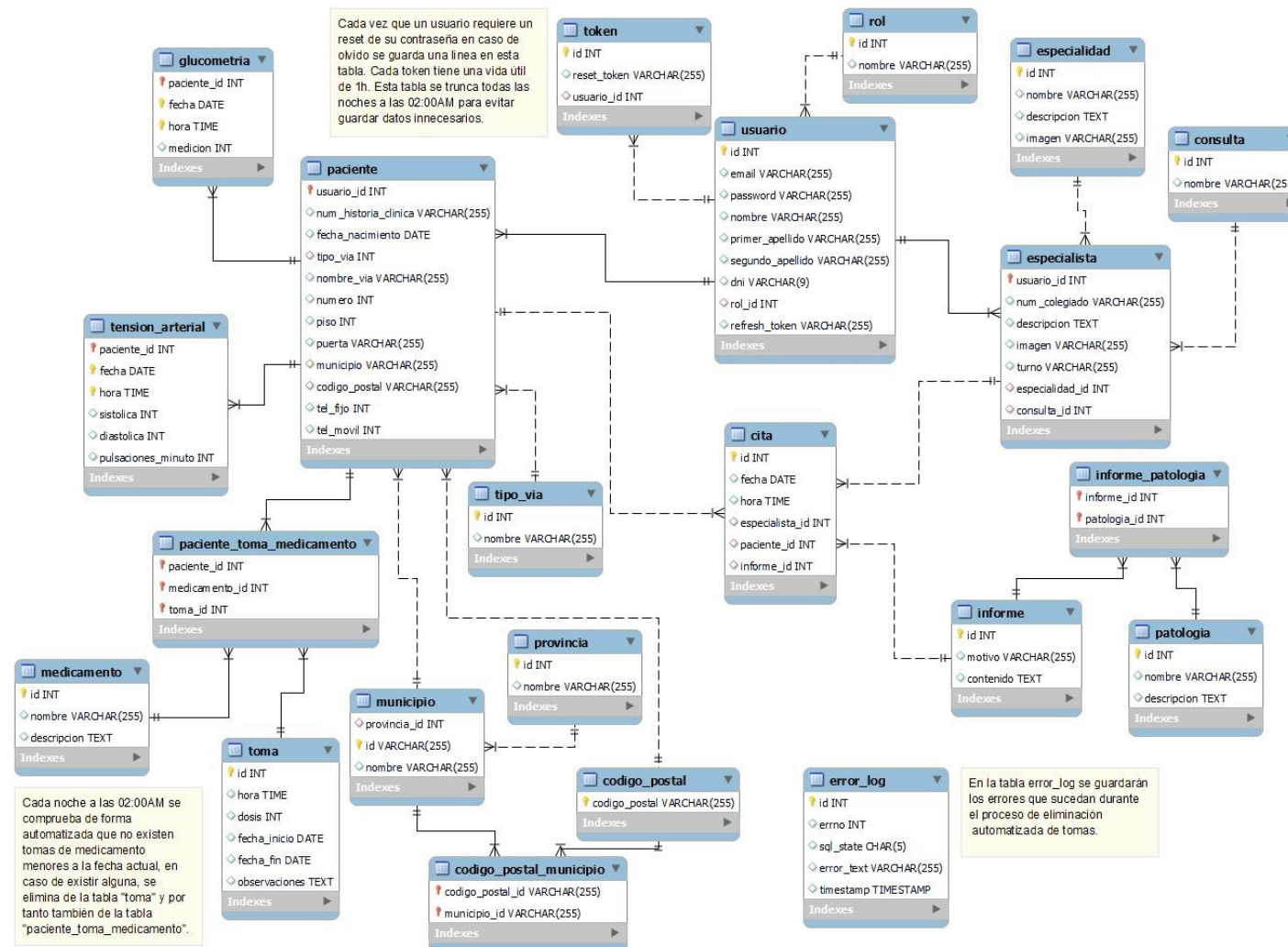


Fig 33 Diagrama de entidad-relación (diagrama de estructura de datos).

5. DIAGRAMA DE CASOS DE USO

A. Herencia de actores

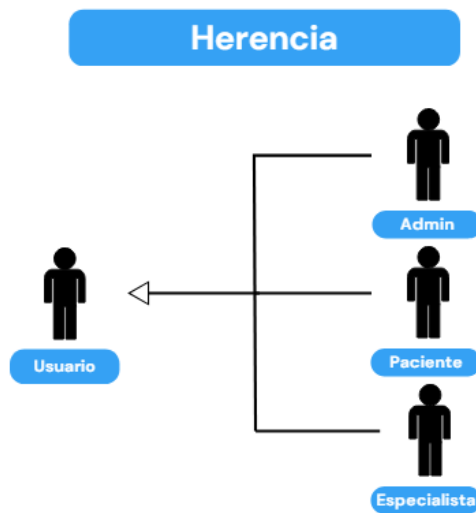


Fig 34 Herencia de actores del diagrama de casos de uso.

B. Casos de uso del usuario

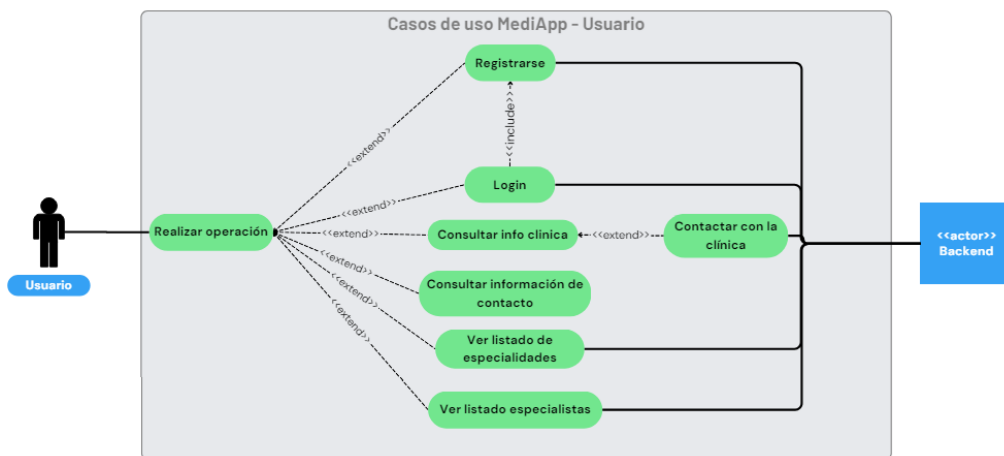


Fig 35 Diagrama de casos de uso del usuario.

C. Casos de uso del administrador

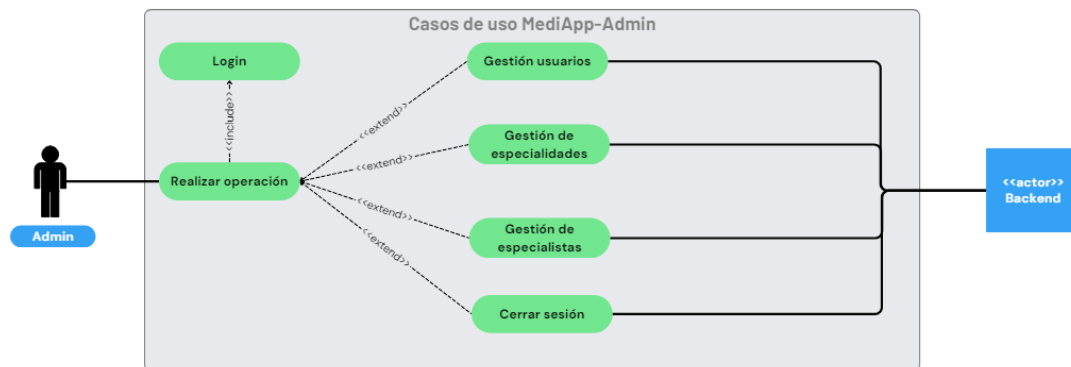


Fig 36 Diagrama de casos de uso del administrador.

D. Casos de uso del paciente

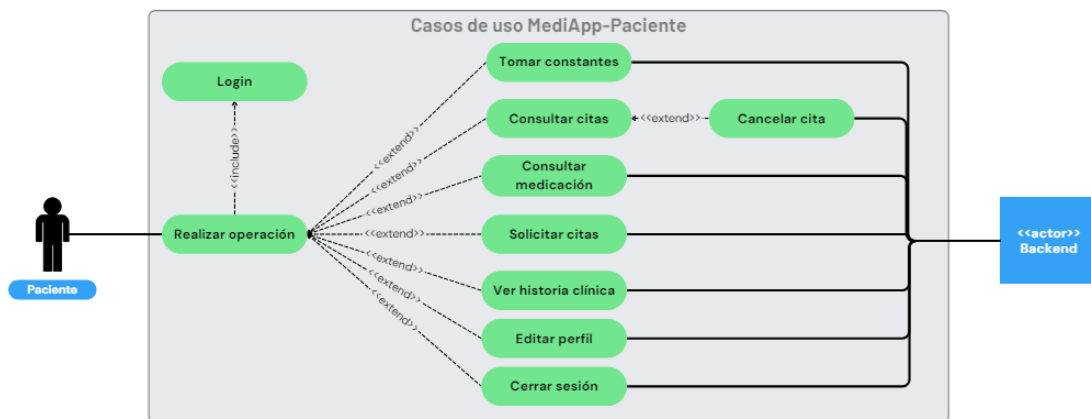


Fig 37 Casos de uso del paciente.

E. Casos de uso del especialista

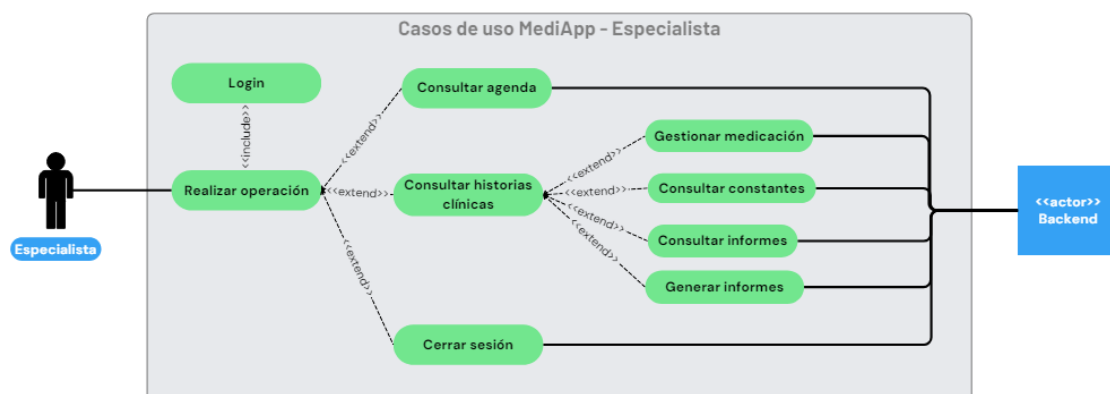


Fig 38 Casos de uso del especialista

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.

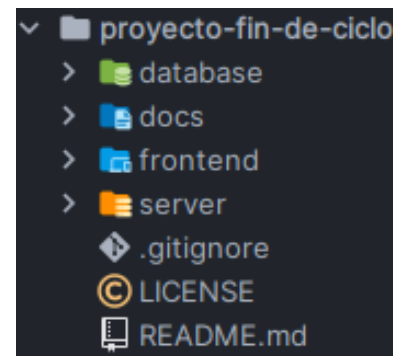


Fig 39 Estructura general del proyecto.

A. Directorio *database*

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

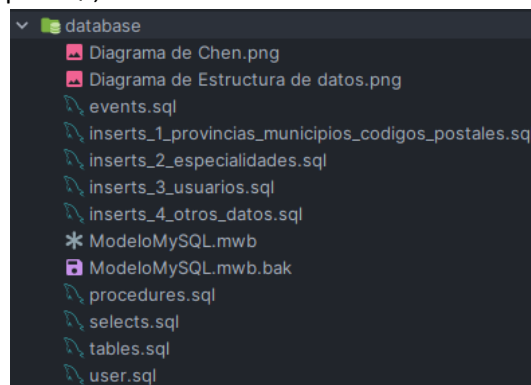


Fig 40 Estructura del directorio 'database'.

B. Directorio *docs*

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

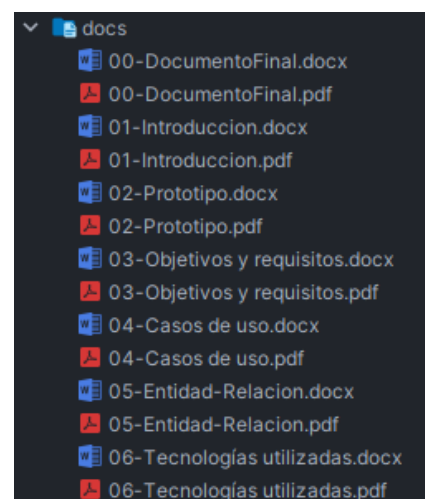


Fig 41 Estructura directorio 'docs'.

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 mueve 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.

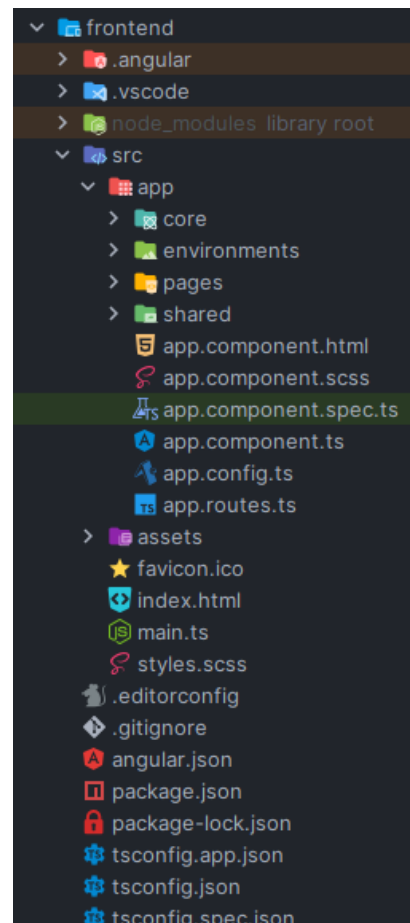


Fig 42 Estructura del directorio 'frontend'.

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

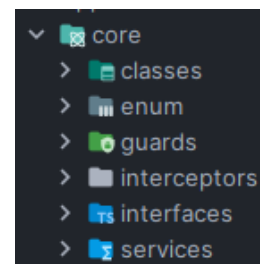


Fig 43 Subdirectorios del directorio 'core'.

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.

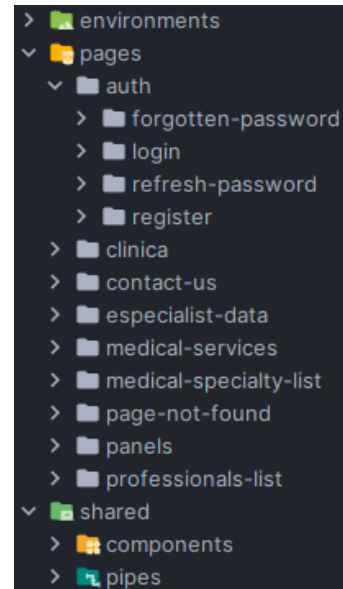


Fig 44 Subdirectorios 'enviroments', 'pages' y 'shared'.

- 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:

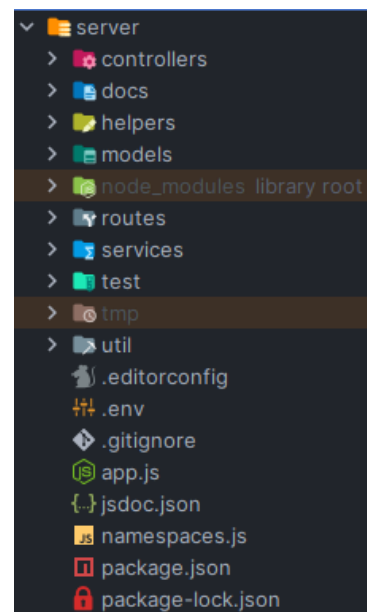


Fig 45 Estructura del directorio 'server'.

- a) **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.
- 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.
- 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

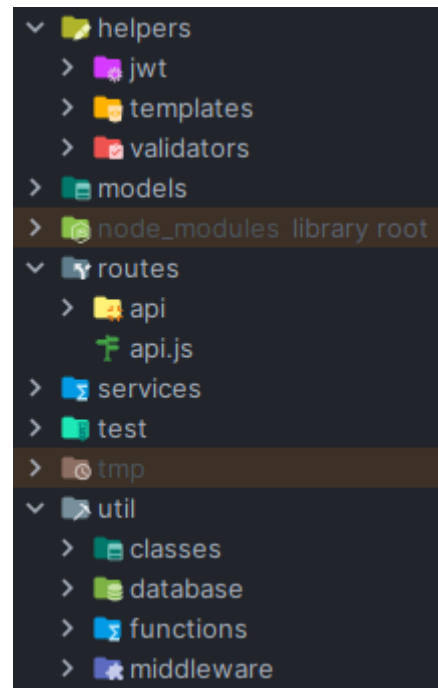


Fig 46 Estructura de los subdirectorios 'helpers', 'routes' y 'util'.

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 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).

Fig 47 Formulario de login de la aplicación.

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.

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.

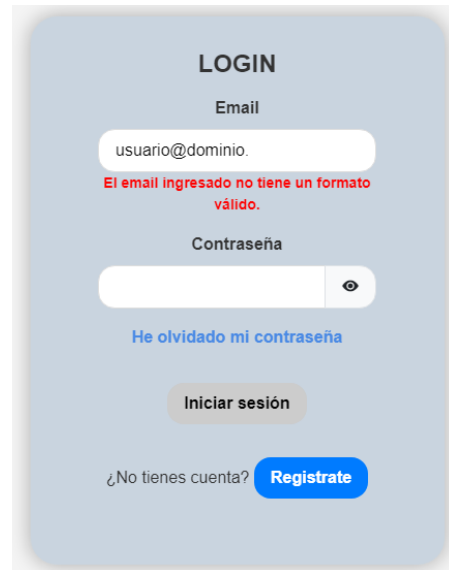
A screenshot of a login form titled "LOGIN". It has two input fields: "Email" and "Contraseña". The "Email" field contains the text "usuario@dominio." and has a red error message below it: "El email ingresado no tiene un formato válido." The "Contraseña" field is empty and has a toggle icon on the right. Below the fields are links for "He olvidado mi contraseña" and "¿No tienes cuenta? Registerate". There are also buttons for "Iniciar sesión" and "Registerate".

Fig 48 Error en la introducción del email.

```
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 49 Generación del formulario reactivo 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 50 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: HttpResponse): void => {
        this.error = error.error.errors[0]
          .toString()
          .replace(/Error: /g, '');
        this.isLoading = false;
      }
    });
}
```

Fig 51 Método encargado de gestionar el login en el cliente.

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

Fig 52 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:

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.

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

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req)
      .pipe(
        tap( observerOrNext: 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( selector: (error) => {
          return throwError( errorFactory: () => error);
        })
      );
  }
}
```

Fig 53 Interceptor para el login del usuario.

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 54 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 librería de express (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 datos no cumpla los requisitos se almacenan en un

```
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 55 Función de validación.

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*.

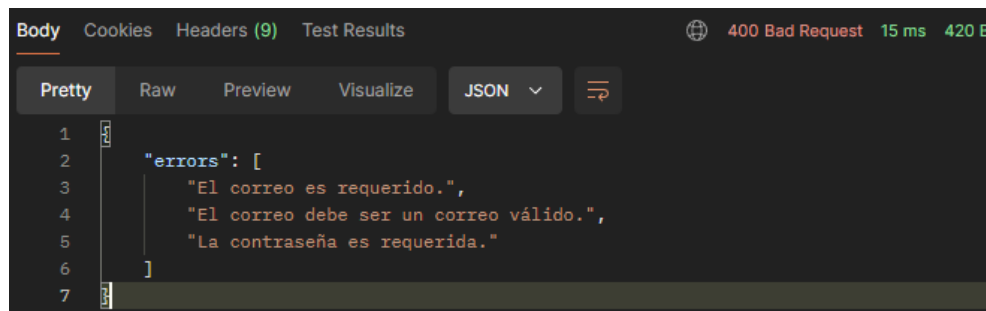


Fig 56 Ejemplo de una petición fallida utilizando Postman.

En el caso de que todo sea correcto se invoca a la función `next()` del validador que marca la finalización del middleware de validación.

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 57 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 59 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.findById(email, conn);

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

Fig 58 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.

```
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 60 Método de búsqueda de un usuario en la base de datos a través del email.

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.

Fig 61 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 62 Función encargada de la firma del token de acceso.

Fig 63 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 ⌵
```

```
{  
  "message": "Inicio de sesi\u00f3n exitoso.",  
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
    eyJpc2VybmVjZkljaXoxMSwibWludClyYyBxLmljolC3E2YWxzShhWUI0tJBGBglcnRvI  
    iwiawF0ijoxNzeEIzncyNTUSLCleHAioIEJMThUsMmMTT9.  
    kfqkQXqSpprjlO9_oT42PLMF3cacgGDH9_Bx9ogceBst".  
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
    eyJpc2VybmVjZkljaXoxMSwibWludClyYyBxLmljolC3E2YWxzShhWUI0tJBGBglcnRvI  
    iwiawF0ijoxNzeEIzncyNTUSLCleHAioIEJMThUsMmMTT9.  
    9HSesGIzl6vhdBHBhiqqJC-C2zc1Llx9wbIK663JD4"
```

Fig 64 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.

http://localhost:4200	
Origin	http://localhost:4200
Key	Value
access_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkjoxLj1c2VyX3JvbGUiOiEsnVzZXJfYmFtZSI6Ikp...
refresh_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkjoxLj1c2VyX3JvbGUiOiEsnVzZXJfYmFtZSI6Ikp...

Fig 65 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 66 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 67 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 autenticación el rol del usuario, algo que se consigue gracias a la decodificación del token de acceso. Si el rol concuerda con el necesario

```
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 68 Guardia de rol de paciente.

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 69 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 70 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 71 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 72 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 rol 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 73 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 74 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 75 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.

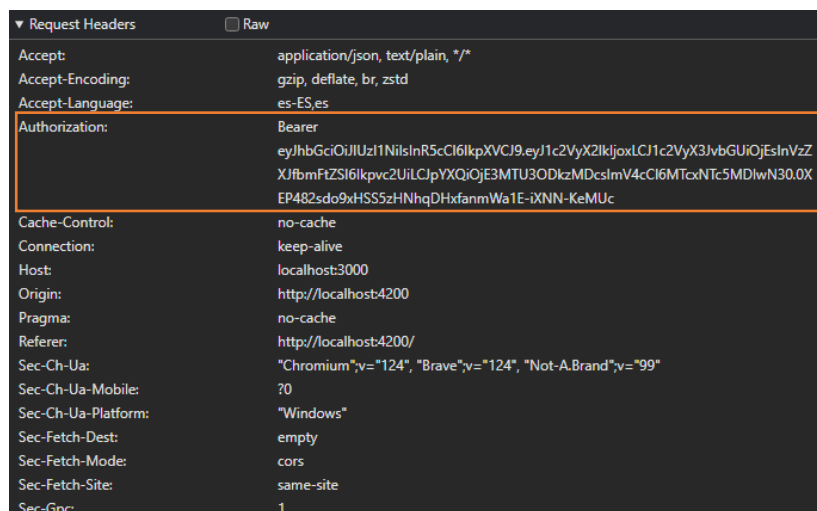


Fig 76 Captura de las herramientas de desarrollador donde se puede ver la cabecera Authorization.

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.

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

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

Fig 77 Método principal del interceptor encargado del refresco de token.

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.

```
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 78 Método encargado de gestionar el error 401.

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: {  
        headers: {  
            Authorization: `Bearer ${token}`  
        }  
    });  
}
```

Fig 79 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, en caso de que no lo esté lanzará un error que, como hemos visto más arriba, provocará el cierre de sesión.

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.

```
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 80 Método del servicio de autenticación encargado de solicitar un token de refresco.

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

Fig 81 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 82 Método del controlador encargado del refresco del token.

5) Funcionamiento del servicio

El servicio se va 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 83 Método del modelo encargado de devolver el token de refresco.

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

refresco en el registro del usuario, por último, los nuevos token 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 84 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.



El formulario tiene un fondo azul claro con un recuadro central de color azul más oscuro. Dentro del recuadro, el título "FORMULARIO RECUPERACIÓN CONTRASEÑA" está en letras blancas mayúsculas. Debajo del título, el texto "Correo electrónico" también es blanco. Hay un campo de entrada de texto blanco con el placeholder "Introduce tu correo electrónico". En la parte inferior del recuadro, hay un botón rectangular azul con el texto "Confirmar" en blanco.

Fig 85 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.

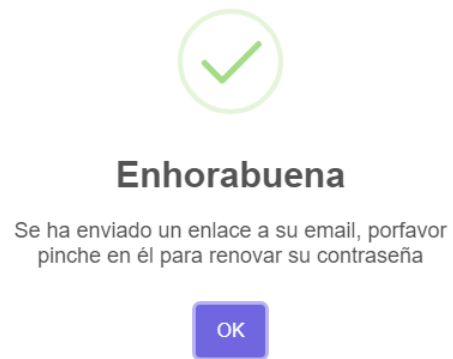


Fig 86 Mensaje de confirmación de envío del email.

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

Fig 87 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 88 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 89 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 90 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 91 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.

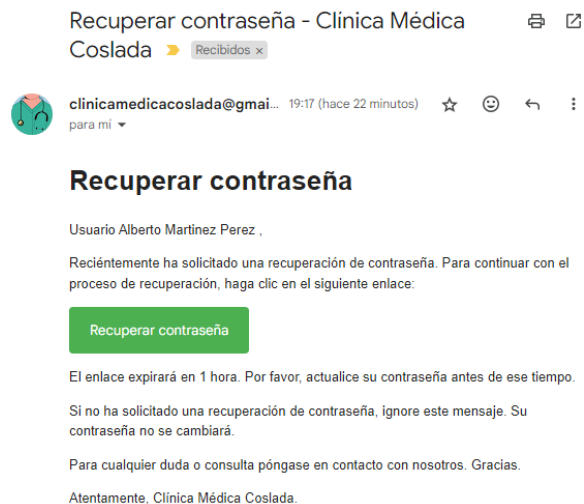


Fig 92 Email de recuperación de contraseña.

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

Fig 93 Ruta 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 94 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 encuentra la sección "Nueva contraseña" con un campo de entrada que contiene el texto "Introduce la nueva contrase" y un ícono de ojo para alternar la visibilidad. A continuación, la sección "Repite la contraseña" con un campo de entrada que contiene "Repite la contraseña" y otro ícono de ojo. En la parte inferior, hay un botón azul con el texto "Confirmar" en blanco.

Fig 95 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 96 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 97 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 98 Método del controlador para continuar el proceso de reinicio de contraseña.

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

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

Fig 99 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 100 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. POLITICA 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 101 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 102 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 'login:1
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 103 Error de CORS.

3. CONSUMO, INTRODUCCIÓN, ACTUALIZACIÓN Y ELIMINACIÓN DE DATOS

4. GENERACIÓN Y ENVÍO DE CORREOS ELECTRÓNICOS

5. GENERACIÓN Y DESCARGA DE DOCUMENTOS PDF

6. 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 104 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 105 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 106 Código QR generado y visible en el PDF de cita.

7. 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 107 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)

```
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 108 Evento para la eliminación de tomas vencidas.

a un procedimiento que se encargue de limpiar los registros que hayan caducado por ser tomas vencidas.

El procedimiento `eliminar_tomas_vencidas_procedure` 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.

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 `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.

```
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;

-- Iniciar 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_medicamento
DELETE FROM paciente_toma_medicamento 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 109 Procedimiento para la eliminación de tomas vencidas.

8. 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.

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.

```
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 110 Configuración de Swagger.

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 111 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 112 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'
 */
router.get(
  path: '/provincia',
  ProvinciaController.getProvincias
);
```

Fig 113 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.

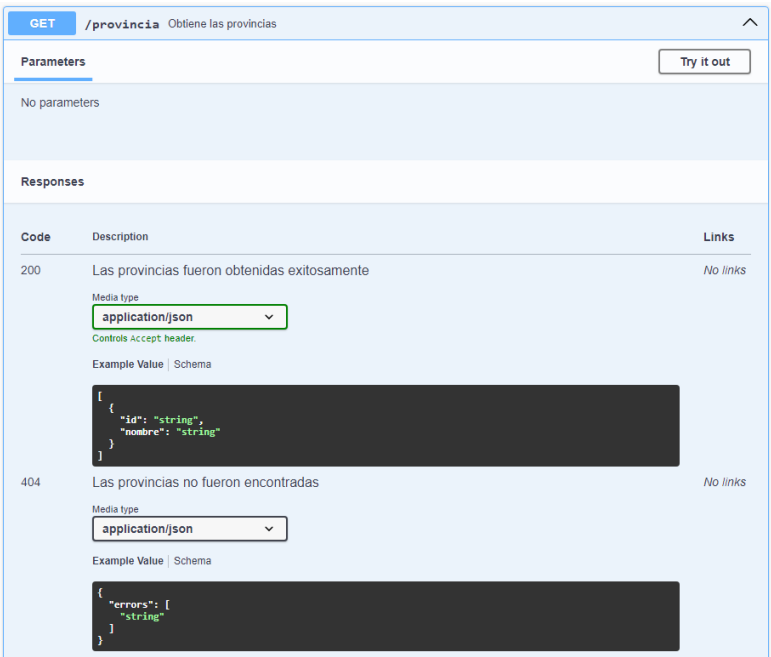


Fig 114 Swagger UI de la ruta GET /api/provincia.

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”.

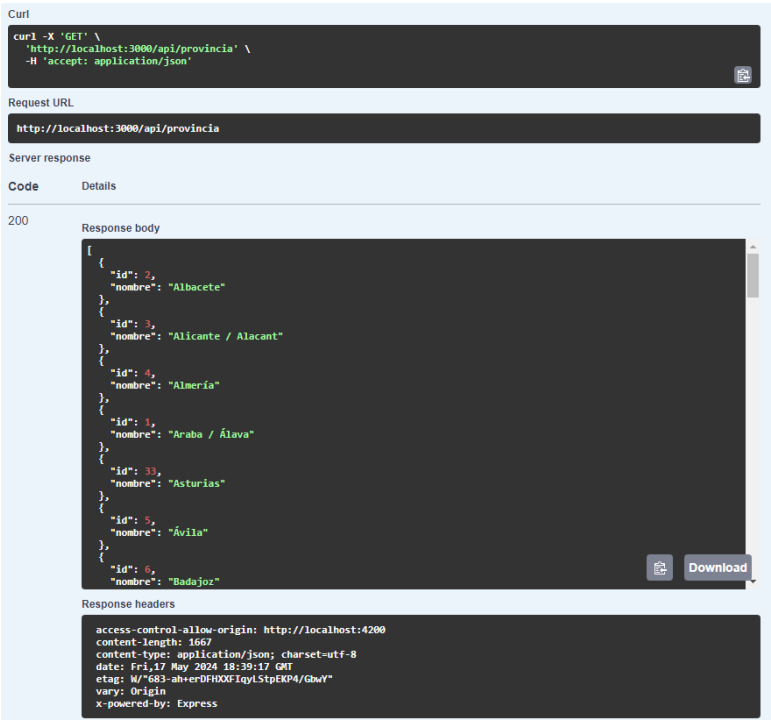


Fig 115 Respuesta de la API.

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.

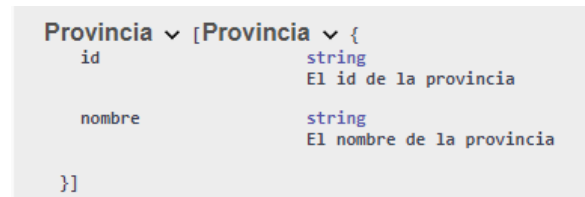


Fig 116 Esquema de Provincia en Swagger UI.

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 117 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.

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.

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

/**
 * @namespace Helpers-Validators-Body
 */
```

Fig 118 Fichero namespaces.js.

- 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 119 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 como `@method` (define el nombre del método),

```
/**
 * @method findByNombre
 * @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.
 * @throws {Error} Si ocurre un error durante la operación, se lanzará un error.
 */
```

Fig 120 Ejemplo de comentario JSDoc.

`@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 121 Script de ejecución de JSDoc.

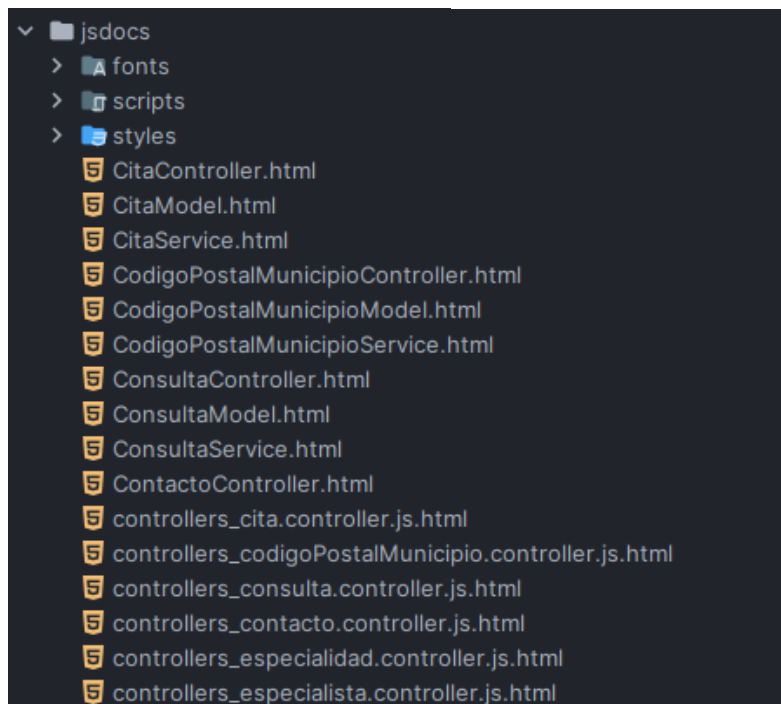


Fig 122 Directorio de JSDoc.

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 123 Visualización de documentación automática en web.

CONCLUSIONES

BIBLIOGRAFÍA

WEBGRAFÍA