



Curso 2023-2024

Ciclo Formativo de Grado Superior Desarrollo de Aplicaciones Web

MediAPP

Creación de una plataforma web de gestión médica usando Angular y Express.JS

AUTORES Rafael Romero Roibu
 Alberto Martínez Pérez

TUTOR Elkin Guadilla González

FECHA DE ENTREGA 12 de junio de 2024

ÍNDICE DE CONTENIDOS

RESUMEN	14
ABSTRACT.....	15
INTRODUCCIÓN	16
OBJETIVOS.....	17
ALCANCE DEL PROYECTO.....	18
1. REQUISITOS FUNCIONALES Y NO FUNCIONALES	18
A. Requisitos funcionales	18
B. Requisitos no funcionales	18
2. PROTOTIPO / MOCKUP.....	20
A. Navegación a través de la barra de navegación	20
B. Mi Espacio - Rol Administrador	21
1) Gestión de usuarios	21
2) Gestión de especialidades	22
3) Gestión de especialistas	23
C. Mi Espacio - Rol Especialista	24
1) Consultar agenda diaria.....	24
2) Gestión historia clínica - Medicación.....	24
3) Gestión historia clínica - Mediciones de glucosa y tensión arterial	25
4) Gestión historia clínica - Informes médicos	26
D. Mi Espacio - Rol Paciente.....	27
1) Solicitar cita	27
2) Historial de citas	28
3) Historial clínico	29
4) Editar perfil	30
3. TECNOLOGÍAS USADAS.....	31
A. Angular.....	31
B. Figma	31
C. Postman	32
D. Git	32
E. GitHub.....	33
F. Express.js	33
G. WebStorm.....	34
H. MySQL.....	34

I.	MySQL Workbench	34
J.	<i>Procedural Language / Structured Query Language (PL/SQL)</i>	35
K.	Bootstrap	35
L.	Handlebars.js	35
M.	<i>Sassy Cascading StyleSheets (SCSS)</i>	36
N.	<i>JavaScript Object Notation Web Tokens (JWT)</i>	36
O.	Swagger	37
P.	Swagger UI	37
Q.	JSDoc	37
R.	Mocha, Chai y Supertest	38
S.	Markdown	39
T.	Ubuntu Server	39
U.	Nginx	39
4.	DIAGRAMAS DE ENTIDAD-RELACIÓN.....	41
A.	Diagrama de Chen	41
B.	Diagrama de estructura de datos	42
5.	DIAGRAMA DE CASOS DE USO	43
A.	Herencia de actores	43
B.	Casos de uso del usuario	43
C.	Casos de uso del administrador	44
D.	Casos de uso del paciente	44
E.	Casos de uso del especialista	44
MARCO PRÁCTICO		45
1.	ESTRUCTURA DEL PROYECTO	45
A.	Directorio <i>database</i>	45
B.	Directorio <i>docs</i>	45
C.	Directorio <i>frontend</i>	46
D.	Directorio <i>server</i>	47
2.	AUTENTICACIÓN, AUTORIZACIÓN Y CONTROL DE ACCESO	50
A.	Inicio de sesión	50
1)	Pantalla de login	50
2)	Validación de datos en <i>frontend</i>	50
3)	Llamada al servicio de autenticación	51
4)	Activación del interceptor para el login	52
5)	Inicio de sesión en el servidor	53

6) Redirección al fichero de rutas.....	53
7) Validación de datos en <i>backend</i>	53
8) Llamada al controlador de usuarios	54
9) Llamada al servicio de usuarios.....	55
10) Llamada al modelo de usuarios.....	56
11) Firma de los tokens de acceso y refresco	57
12) Envío de los tokens al cliente	58
B. Protección de rutas en el cliente	59
C. Protección de rutas en el servidor	60
1) Verificación del token de acceso	61
2) Verificación del rol de usuario.....	62
3) Verificación del id del usuario	62
D. Manejo de la cabecera <i>authorization</i> por el cliente	62
E. Tratamiento del token de refresco	63
1) Intercepción del error 401 desde el servidor.....	63
2) Llamada al servicio de autenticación.....	65
3) Comunicación con el servidor	66
4) Manejo del token por el controlador	67
5) Funcionamiento del servicio.....	67
F. Reinicio de contraseña.....	68
1) Servicio del cliente.....	69
2) Manejo por el servidor	69
3) Continuar el proceso tras el correo electrónico.....	71
4) Finalización del proceso en el servidor.....	72
G. Cierre de sesión	74
1) Cierre de sesión en el cliente.....	74
2) Interceptor de logout	74
3) Manejo del cierre de sesión en el servidor	75
4) Eliminación de los tokens del almacenamiento local.....	75
H. Política de <i>Cross-Origin Resource Sharing</i> (CORS)	75
3. PETICIONES CLIENTE - SERVIDOR.....	77
A. Método GET	78
1) Envío de datos desde el servidor.....	78
2) Búsqueda de datos en la base de datos	80
3) Generación de la paginación en el servidor	80
4) Recepción y presentación de los datos en el cliente.....	81
5) Manejo de la paginación en el cliente	83

B. Ruta POST	85
1) Recogida de datos en el cliente.....	85
2) Manejo de ficheros. Transformación a base64.	87
3) Generación del modelo de datos y comunicación al servidor.....	88
4) Manejo de datos en el servidor.....	89
C. Método PUT.....	91
1) Edición de datos en el cliente.....	91
2) Comunicación al servidor	92
D. Método DELETE	94
1) Manejo en el cliente	94
2) Eliminación en el servidor	95
4. GENERACIÓN Y ENVÍO DE CORREOS ELECTRÓNICOS	97
5. GENERACIÓN Y DESCARGA DE DOCUMENTOS PDF	100
1) Solicitud de descarga en el cliente	100
2) Manejo por parte del servidor	100
3) Generación del PDF	101
4) Destrucción del PDF.....	103
5) Descarga en el cliente.....	103
6. GENERACIÓN DE CÓDIGOS QR	104
7. AUTOMATIZACIÓN DE PROCESOS	105
A. Evento para la eliminación de registros en la tabla token.....	105
B. Evento para eliminar las tomas vencidas	105
8. GENERACIÓN DE ARCHIVOS DE LOG.....	107
9. GENERACIÓN DE DOCUMENTACIÓN AUTOMÁTICA	108
A. Swagger / Swagger UI	108
1) Archivo de configuración	108
2) Definición de la ruta en el archivo app.js	109
3) Componentes de Swagger	109
4) Comentarios @swagger	109
5) Interfaz web.....	110
B. JSDoc	111
1) Configuración de JSDoc	111
2) Creación de comentarios JSDoc	111
3) Generación de la documentación	112
4) Visualización en web	113

10. DESPLIEGUE DE LA APLICACIÓN WEB	114
A. Configuración común de los servidores.....	114
1) Especificaciones de las máquinas virtuales	114
2) Configuración de netplan	114
3) Configuración del firewall.....	114
B. Despliegue del servidor web.....	115
1) Generación de la build de Angular	115
2) Instalación y configuración de Nginx.....	116
C. Despliegue del servidor de aplicaciones	116
D. Despliegue del servidor de base de datos	117
1) Instalación y configuración de MySQL Server.	117
2) Generación del usuario que se conectará a la base de datos	118
3) Configuración del firewall.....	118
4) Conexión a la base de datos desde el anfitrión	118
E. Demostración de funcionamiento	119
1) Cambios en el servidor web	119
2) Cambios en el servidor de aplicaciones.....	120
3) Cambios en el servidor de base de datos	120
4) Acceso desde el anfitrión	121
CONCLUSIONES.....	122
WEBGRAFÍA.....	123

ÍNDICE DE FIGURAS

Fig 1 Prototipo de la navegación a través de la barra de navegación de la web.....	20
Fig 2 Prototipo para la gestión de usuarios por parte del administrador.....	21
Fig 3 Prototipo para la gestión de especialidades por parte del administrador.....	22
Fig 4 Prototipo para la gestión de especialistas por parte del administrador.....	23
Fig 5 Prototipo para la consulta de la agenda diaria por parte del especialista.....	24
Fig 6 Prototipo para la gestión de la medicación por parte del especialista.....	24
Fig 7 Prototipo para consulta de las mediciones de glucosa y tensión arterial del paciente.	25
Fig 8 Prototipo para la consulta y generación de nuevos informes médicos.....	26
Fig 9 Prototipo para la solicitud de citas con especialistas por parte del paciente.....	27
Fig 10 Prototipo para la gestión de citas por parte del paciente.....	28
Fig 11 Prototipo para la inserción de datos de mediciones de tensión arterial y glucosa por parte del paciente.....	29
Fig 12 Prototipo para la edición del perfil por parte del paciente.....	30
Fig 13 Logo de Angular.....	31
Fig 14 Logo de Figma.....	31
Fig 15 Logo de Postman.....	32
Fig 16 Logo de Git.....	32
Fig 17 Logo de GitHub.	33
Fig 18 Logo de Express.js.....	33
Fig 19 Logo de WebStorm.	34
Fig 20 Logo de MySQL.	34
Fig 21 Logo de MySQL Workbench.	34
Fig 22 Logo de PL/SQL.	35
Fig 23 Logo de Bootstrap.	35
Fig 24 Logo de Handlebars.js.....	35
Fig 25 Logo de SASS-SCSS.....	36
Fig 26 Logo de JSON Web Token.....	36
Fig 27 Logo de Swagger.....	37
Fig 28 Logo de Swagger UI.	37
Fig 29 Logo de JsDoc.	37
Fig 30 Logos de Mocha, Chai y Supertest.....	38
Fig 31 Logo de Markdown.....	39

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

Fig 66 Ejemplo de respuesta desde el servidor ante un login correcto.....	58
Fig 67 Métodos del servicio de autenticación para almacenar los tokens.....	58
Fig 68 Almacenamiento local del navegador.....	58
Fig 69 Pantalla de opciones del usuario en función del rol.....	59
Fig 70 Ejemplo de ruta protegida en Angular.	59
Fig 71 Guardia de rol de paciente.	59
Fig 72 Método del servicio de autenticación para obtener el rol del usuario.	60
Fig 73 Ejemplo de protección de rutas en Node.JS.....	60
Fig 74 Funcionalidad para la verificación del token de acceso.	61
Fig 75 Mecanismo que impide que un paciente (role 2) pueda acceder a datos ajenos a los de su cuenta.....	61
Fig 76 Middleware para la verificación del rol del usuario.	62
Fig 77 Middleware para la verificación del identificador del usuario.....	62
Fig 78 Generación de la cabecera Authorization en el cliente.....	63
Fig 79 Captura de las herramientas de desarrollador donde se puede ver la cabecera Authorization.	63
Fig 80 Método principal del interceptor encargado del refresco de token.....	64
Fig 81 Método encargado de gestionar el error 401.	64
Fig 82 Método encargado de reciclar la petición original que provocó el 401.....	65
Fig 83 Método del servicio de autenticación encargado de solicitar un token de refresco ..	66
Fig 84 Enrutador para el refresco del token.	66
Fig 85 Método del controlador encargado del refresco del token.	67
Fig 86 Método del modelo encargado de devolver el token de refresco.....	67
Fig 87 Método del servicio encargado de realizar el proceso de refresco del token.	68
Fig 88 Formulario de recuperación de contraseña.	68
Fig 89 Mensaje de confirmación de envío del email.....	69
Fig 90 Servicio del cliente encargado de la funcionalidad de reinicio de contraseña.	69
Fig 91 Enrutador para la contraseña olvidada.	69
Fig 92 Controlador encargado del reinicio de contraseña.....	70
Fig 93 Método del servicio de Token encargado de crear el token de reinicio.	70
Fig 94 Servicio encargado del reinicio de contraseña.....	70
Fig 95 Método encargado de enviar un correo electrónico para continuar el proceso de reinicio.....	71
Fig 96 Email de recuperación de contraseña.	71

Fig 97 Ruta del enrutador de Angular para continuar el proceso.....	71
Fig 98 Captura del parámetro del token.	71
Fig 99 Formulario para la recuperación de contraseña. Introducción de nueva contraseña.	72
Fig 100 Servicio de Angular encargado de continuar el reinicio de contraseña.....	72
Fig 101 Ruta del servidor que continua el reinicio de contraseña.....	72
Fig 102 Método del controlador para continuar el proceso de reinicio de contraseña.	73
Fig 103 Método del servicio para continuar el proceso de reinicio de contraseña.....	73
Fig 104 Método de utilidades para la encriptación de contraseñas.....	73
Fig 105 Botón de cierre de sesión.....	74
Fig 106 Método de cierre de sesión en el componente sidebar del cliente.....	74
Fig 107 Interceptor para el cierre de sesión.	74
Fig 108 Método de actualización del valor del token de refresco a nulo.	75
Fig 109 Eliminación de los tokens del almacenamiento local.....	75
Fig 110 Configuración de CORS del servidor.....	76
Fig 111 Variables de entorno para las CORS.	76
Fig 112 Error de CORS si se realiza una petición desde un origen no permitido.	76
Fig 113 Ejemplo de ruta GET.	77
Fig 114 Ejemplo de ruta POST.....	77
Fig 115 Ejemplo de ruta DELETE.	77
Fig 116 Ejemplo de ruta PUT.....	77
Fig 117 Petición GET de citas.	78
Fig 118 Comprobación de los parámetros de búsqueda.	78
Fig 119 Petición para actualizar las especialidades existentes.	79
Fig 120 Parámetros de 'page', 'limit' y 'search'.....	79
Fig 121 Parámetros 'fechaInicioCita' y 'fechaFinCita'.	79
Fig 122 Funcionamiento del méto readCitas().....	80
Fig 123 Consulta de citas.....	80
Fig 124 Paginación en el servidor.....	81
Fig 125 Listado con paginación y filtrado.....	81
Fig 126 Uso de objeto Subject en la función ngOnInit para recoger las citas nada más cargue el componente.	82
Fig 127 Llamada al método getCitas() del servicio de citas.	82
Fig 128 Método getCitas().....	83
Fig 129 Método showResults().....	83

Fig 130 Funcionamiento de paginate en bucle for de Angular.....	83
Fig 131 Componente pagination-controls.....	84
Fig 132 Método changePage().....	84
Fig 133 Método updateFilters().....	84
Fig 134 Inputs de filtrado.....	84
Fig 135 Funciones relativas a listado y creación de usuarios.....	85
Fig 136 Formulario creación especialista.....	85
Fig 137 Formulario creación paciente.....	85
Fig 138 Estructura FormGroup con FormControls.....	86
Fig 139 FormControl que comprueba si los campos son válidos.....	86
Fig 140 FormControl comprueba si cada campo es invalido e imprime su mensaje correspondiente.....	86
Fig 141 Mensajes de error de cada campo al ser inválidos.....	87
Fig 142 Atributos de la interfaz SpecialistModel.....	87
Fig 143 Input que recoge las imágenes subidas al formulario.....	87
Fig 144 Método onFileSelect().....	87
Fig 145 Método toBase64().....	88
Fig 146 Construcción objeto SpecialistModel.....	88
Fig 147 Llamada al servicio de AuthService para registrar un especialista.....	88
Fig 148 Método del AuthService que llama a la ruta de registro de especialistas.....	89
Fig 149 Métodos de verificación de token del usuario y su rol.....	89
Fig 150 Validación de los campos del objeto SpecialistModel enviado al backend.....	89
Fig 151 Método sanitizeInput().....	90
Fig 152 Método postRegistro() encargado de llamar al servicio de usuarios para que guarde al usuario en cuestión.....	90
Fig 153 Creación de la transacción y comprobación que el especialista no exista con anterioridad.....	91
Fig 154 Formulario de edición de especialista.....	91
Fig 155 Llamada al método updateSpecialist().....	92
Fig 156 Método updateSpecialist() del servicio de autenticación.....	92
Fig 157 Verificación de token y rol en la ruta PUT del back.....	92
Fig 158 Comprobación ID del especialista.....	93
Fig 159 Establecimiento de la conexión y creación de la transacción.....	93

Fig 160 Comprobación de existencia del correo, contraseña y número de colegiado del especialista.....	93
Fig 161 Actualización del usuario por medio del objeto UsuarioModel.....	94
Fig 162 Vista componente listado de especialidades.	94
Fig 163 Modal de confirmación de eliminación.....	94
Fig 164 Método confirmarCancelacion().....	95
Fig 165 Método eliminateSpeciality() del servicio de especialidades.....	95
Fig 166 Comprobación token y rol de la ruta DELETE del back.....	95
Fig 167 Validación del tipo y existencia de ID de la especialidad.	95
Fig 168 Método deleteEspecialidad() del servicio de especialidades del back.	96
Fig 169 Respuesta de estado 200 si se ha eliminado correctamente la especialidad.	96
Fig 170 Uso del servicio de mails en la creación de usuarios paciente.....	97
Fig 171 Método encargado de mandar el mail de bienvenida.	97
Fig 172 Generación del transportador.....	97
Fig 173 Compilación de la plantilla handlebars.	98
Fig 174 Plantilla handlebars para el mail de bienvenida.....	98
Fig 175 Generación del objeto con los detalles del mail.	98
Fig 176 Mail de bienvenida.	99
Fig 177 Vista de la medicación del paciente.	100
Fig 178 Método de Angular que maneja la descarga.....	100
Fig 179 Servicio de Node.JS para el manejo de las prescripciones del paciente.	100
Fig 180 Método del servicio de PDF encargado de generar el PDF de prescripciones.....	101
Fig 181 Generación del árbol de directorios.....	101
Fig 182 Método encargado de generar el PDF utilizando puppeteer.....	102
Fig 183 Fragmento de la plantilla de handlebars.....	102
Fig 184 Respuesta de descarga desde el servidor.....	103
Fig 185 Método encargado de la destrucción del fichero.	103
Fig 186 Ventana de descarga del PDF en el navegador.	103
Fig 187 Fragmento del PDF resultante.....	103
Fig 188 Función encargada de la generación del código QR.....	104
Fig 189 Fragmento de código donde se utiliza la funcionalidad de QR.	104
Fig 190 Código QR generado y visible en el PDF de cita.	104
Fig 191 Evento para la eliminación de registros en la tabla token.	105
Fig 192 Evento para la eliminación de tomas vencidas	105

Fig 193 Procedimiento para la eliminación de tomas vencidas. Generación del cursor.	106
Fig 194 Procedimiento para la eliminación de tomas vencidas. Bucle de eliminación.	106
Fig 195 Directorio logs.	107
Fig 196 Configuración de rotatig-file-stream.	107
Fig 197 Configuración de morgan.	107
Fig 198 Función para la generación del nombre del fichero de log.	107
Fig 199 Ejemplo de fichero de log.	107
Fig 200 Configuración de Swagger.	108
Fig 201 Ruta del servidor que permitirá el acceso a Swagger UI.	109
Fig 202 Ejemplo de esquema en swagger.	109
Fig 203 Ejemplo de comentario @swagger.	109
Fig 204 Swagger UI de la ruta GET /api/provincia.	110
Fig 205 Respuesta de la API.	110
Fig 206 Esquema de Provincia en Swagger UI.	110
Fig 207 Archivo de configuración de JSDoc.	111
Fig 208 Fichero namespaces.js.	111
Fig 209 Ruta a JSDoc en app.js.	111
Fig 210 Ejemplo de comentario JSDoc.	112
Fig 211 Script de ejecución de JSDoc.	112
Fig 212 Directorio de JSDoc.	112
Fig 213 Visualización de documentación automática en web.	113
Fig 214 Instalación de Ubuntu Server 24.04.	114
Fig 215 Configuración de Netplan para el servidor web.	114
Fig 216 Estado del firewall en el servidor web.	114
Fig 217 Ejecución de ng build.	115
Fig 218 Contenido del directorio /dist.	115
Fig 219 Contenido de /var/www/html/ tras la copia del contenido del proyecto.	116
Fig 220 Fichero de configuración de Nginx.	116
Fig 221 Sitio web alojado en el servidor.	116
Fig 222 Lanzamiento del servidor de NodeJS.	116
Fig 223 Situación del firewall en el servidor de aplicaciones.	117
Fig 224 Acceso a la web de documentación de la API desde el anfitrión.	117
Fig 225 Parámetro bind-adress.	117
Fig 226 Generación del usuario de la base de datos.	118

Fig 227 Generación de la base de datos.	118
Fig 228 Privilegios al usuario 'clinica_user'.	118
Fig 229 Situación del firewall en el servidor de base de datos.	118
Fig 230 Creación de conexión en MySQL Workbench.	118
Fig 231 Conexión exitosa a la base de datos desde el anfitrión.	119
Fig 232 Búsqueda de las tablas de la base de datos 'clinica' en el servidor.....	119
Fig 233 Modificación del fichero de entorno de Angular.	119
Fig 234 Netplan del servidor de aplicaciones.	120
Fig 235 Fichero .env de NodeJS.....	120
Fig 236 Netplan del servidor de base de datos.....	120
Fig 237 Configuración de ufw para escuchar sólo al servidor de aplicaciones.....	120
Fig 238 Web de especialidades en el anfitrión.	121
Fig 239 Web de paciente en el anfitrión.	121

RESUMEN

El proyecto *MediApp* consiste en el diseño y desarrollo de una aplicación web para la gestión médica basado en 3 tipos de usuario: administradores, especialistas y pacientes. Para ello se han utilizado los *frameworks* de Angular para *frontend* y de ExpressJS para *backend*.

Las funcionalidades principales de la aplicación la gestión de especialidades médicas y usuarios por parte de los usuarios administradores, la gestión de informes médicos y prescripciones de medicamentos por parte de los usuarios especialistas y la solicitud de nuevas citas, almacenamiento de mediciones de glucosa y tensión arterial, así como, consulta de sus datos médicos por parte de los pacientes.

Durante estas semanas hemos afrontado diferentes dificultades, la mayoría de ellas debido al reto que supone desarrollar el propio proyecto a la vez que se aprende el manejo de tecnologías completamente nuevas como los dos *frameworks* elegidos para el proyecto, el uso de JSON Web Tokens para la autenticación y autorización, el manejo de SCSS o el uso de nuevas herramientas como *Postman*.

Es por ello por lo que como principal conclusión podemos decir que el proyecto nos ha servido como método de aprendizaje no sólo de estas herramientas y tecnologías sino también de una nueva forma de arquitectura y organización de software.

También hemos aprendido que, durante el ciclo de vida del desarrollo de software, la implementación de nuevas funcionalidades o de nuevas versiones de las versiones y subversiones de los *frameworks* puede provocar la aparición de *bugs* en funcionalidades ya desarrolladas o incompatibilidades con librerías ya instaladas que deben ser arreglados a la vez que se desarrollan nuevas funcionalidades para que la aplicación continúe con su desarrollo de forma óptima.

El código fuente de este proyecto puede ser encontrado en este [repositorio de GitHub](#) bajo licencia GNU GPLv3.

ABSTRACT

The MediApp project involves the design and development of a web application for medical management based on three types of users: administrators, specialists, and patients. For this, we used two frameworks: Angular for the frontend and ExpressJS for the backend.

The main functionalities of the application include the management of medical specialties and users by administrator users, the management of medical reports and medication prescriptions by specialist users, and the scheduling of new appointments, storage of glucose and blood pressure measurements, as well as consultation of their medical data by patients.

During these weeks, we have faced various difficulties, most of them due to the challenge of developing the project while simultaneously learning to handle completely new technologies such as the two frameworks chosen for the project, the use of JSON Web Tokens for authentication and authorization, handling SCSS, and using new tools like Postman.

Therefore, we can conclude that the project has served as a learning method not only for these tools and technologies but also for a new form of software architecture and organization.

We have also learned that, during the software development lifecycle, the implementation of new functionalities or new versions and subversions of the frameworks can cause bugs in already developed functionalities or incompatibilities with installed libraries that must be fixed while developing new functionalities to ensure the application continues its development optimally.

The source code of this project can be found in this [GitHub repository](#) under the GNU GPLv3 license.

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

A. Requisitos 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.

B. Requisitos 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 al *backend*.
2. La interfaz debe ser amigable y fácil de utilizar.
3. La interfaz debe ser responsive 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

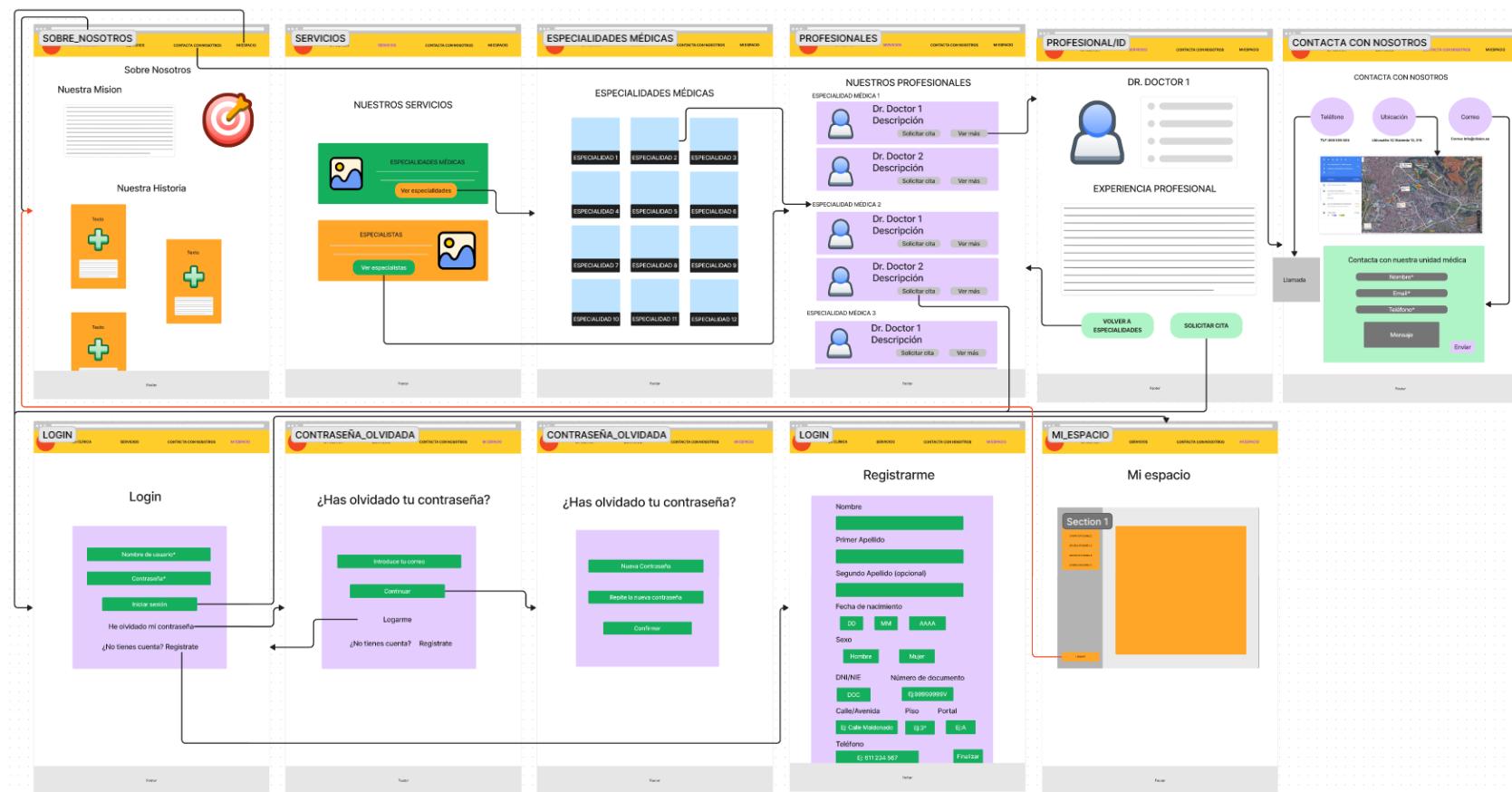


Fig 1 Prototipo de la navegación a través de la barra de navegación de la web.

B. Mi Espacio - Rol Administrador

1) Gestión de usuarios

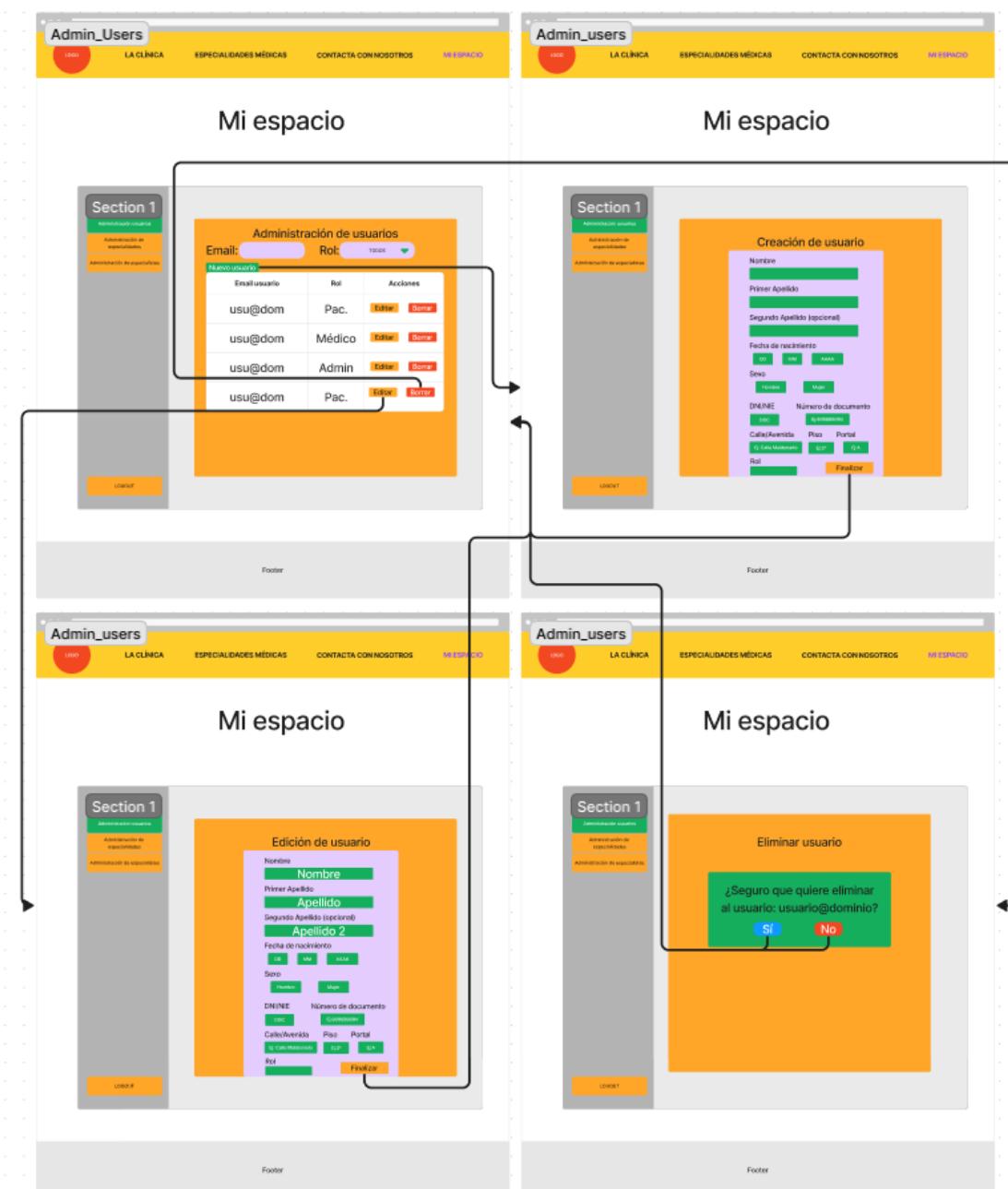


Fig 2 Prototipo para la gestión de usuarios por parte del administrador.

2) Gestión de especialidades

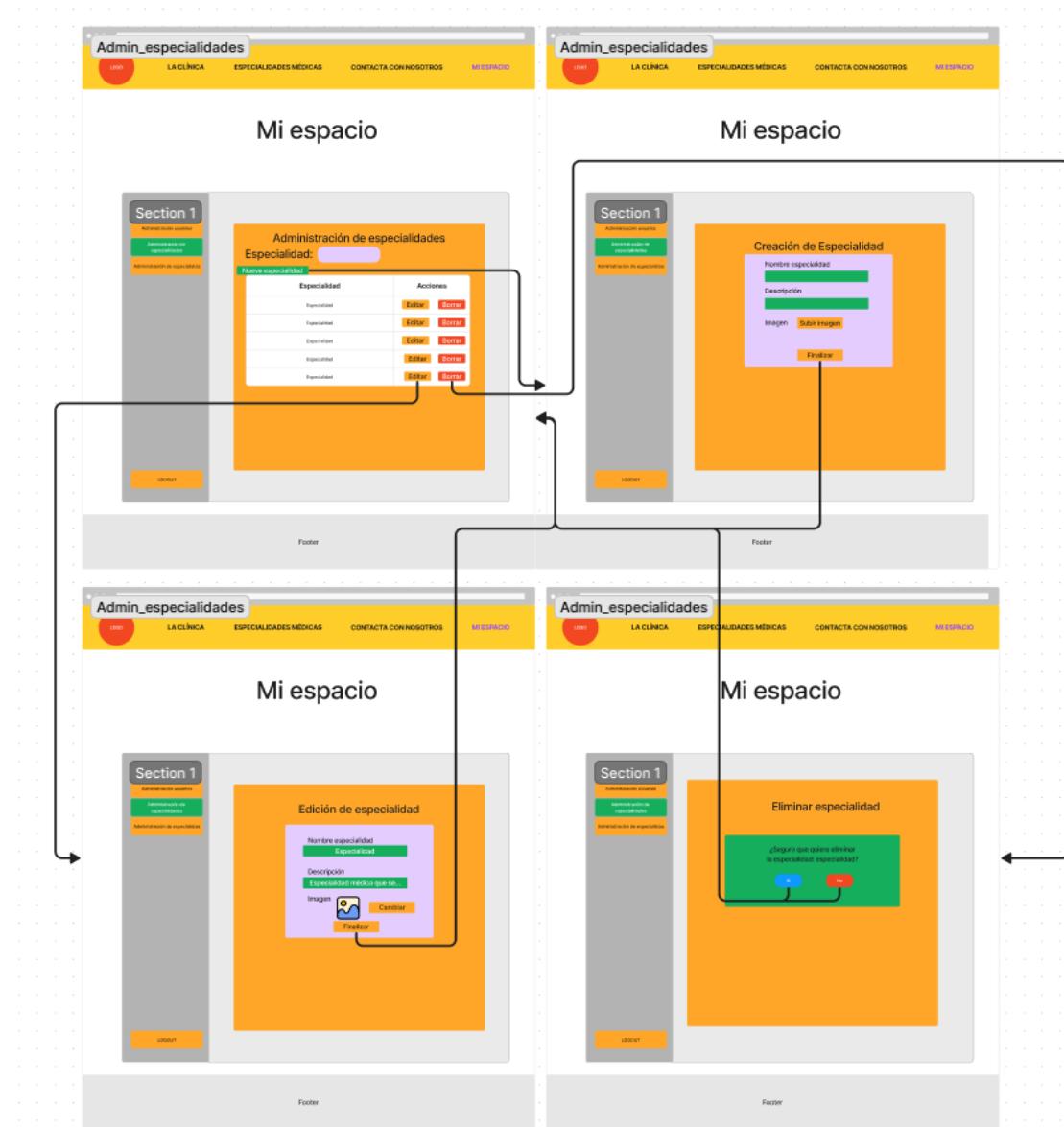


Fig 3 Prototipo para la gestión de especialidades por parte del administrador.

3) Gestión de especialistas

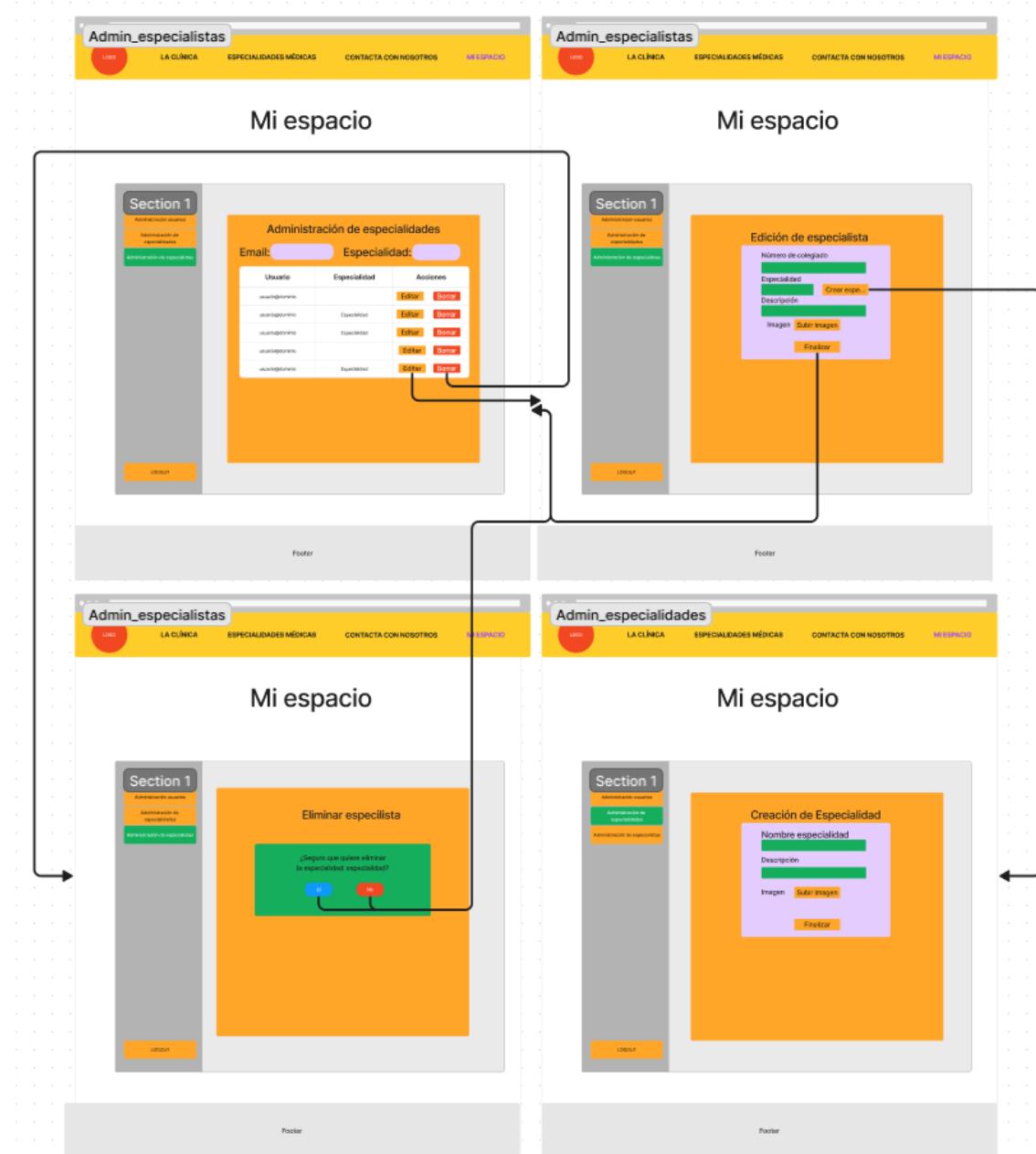


Fig 4 Prototipo para la gestión de especialistas por parte del administrador.

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

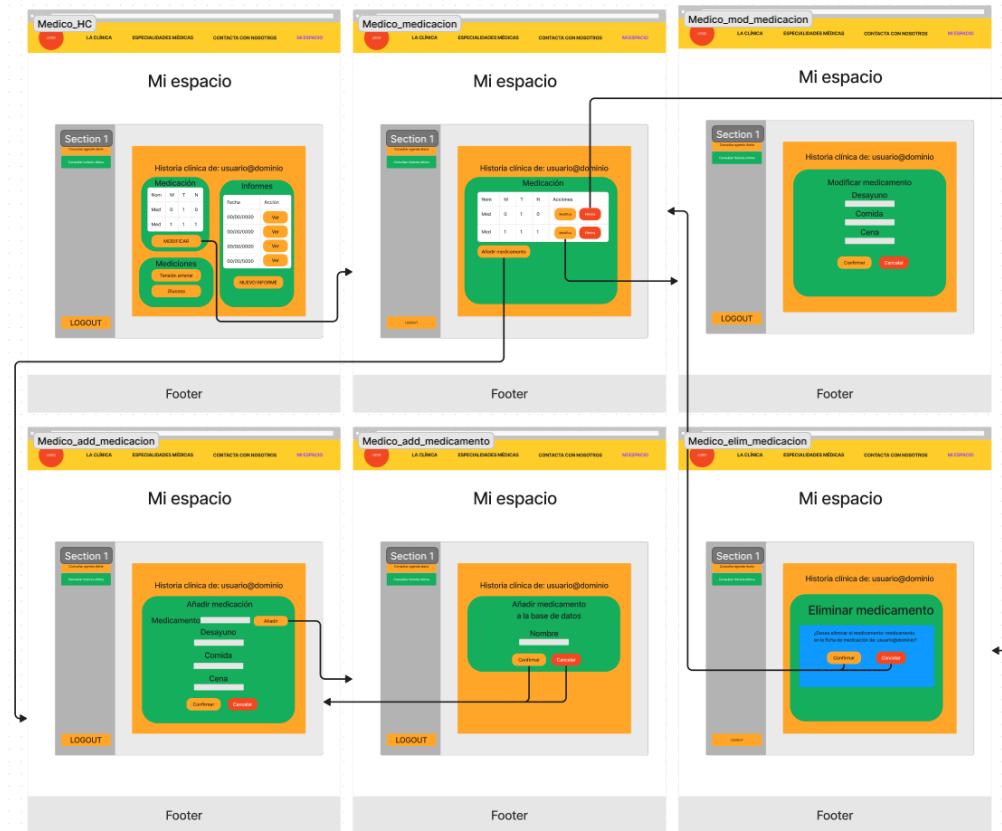


Fig 6 Prototipo para la gestión de la medicación por parte del especialista.

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

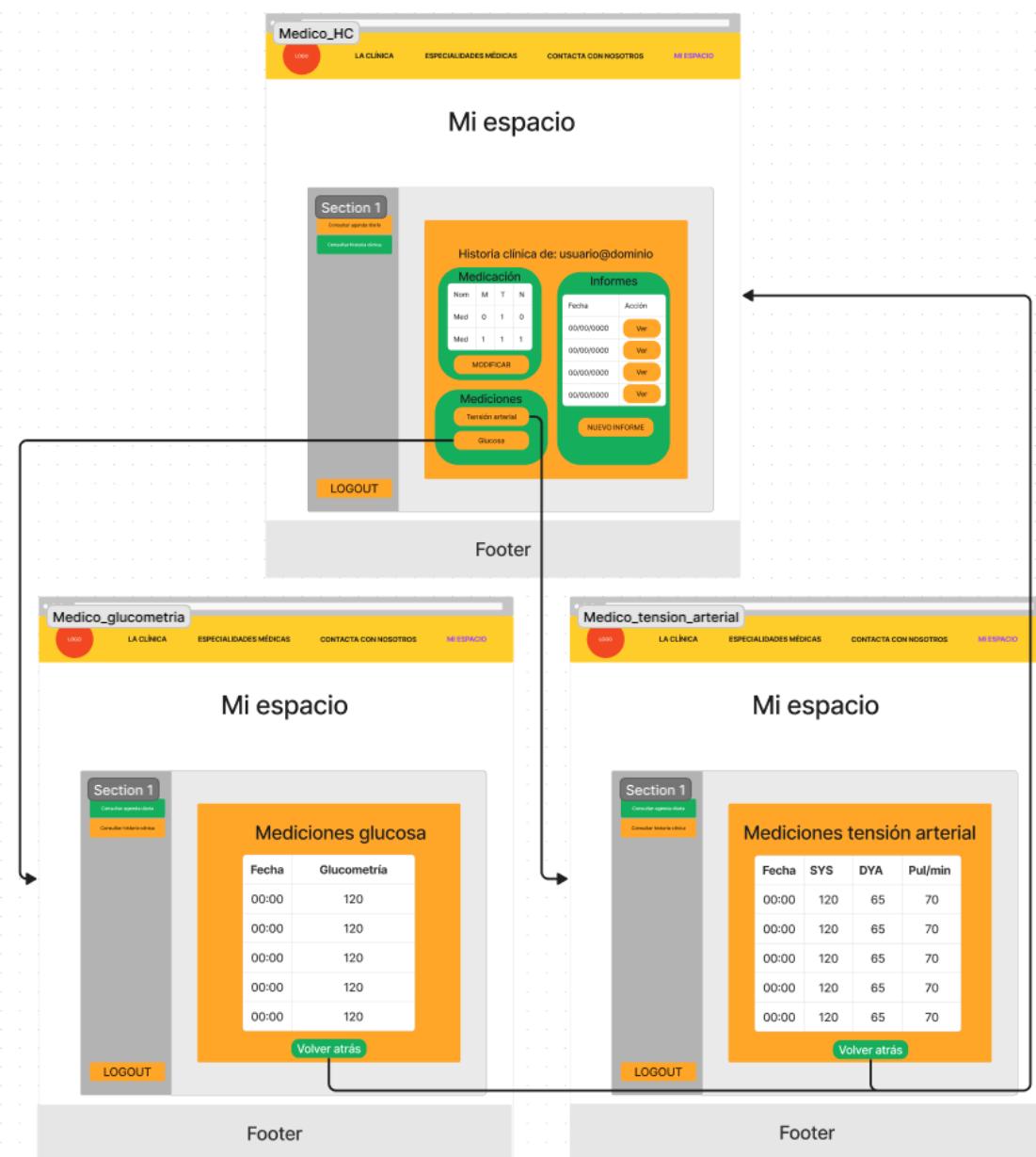


Fig 7 Prototipo para consulta de las mediciones de glucosa y tensión arterial del paciente.

4) Gestión historia clínica - Informes médicos

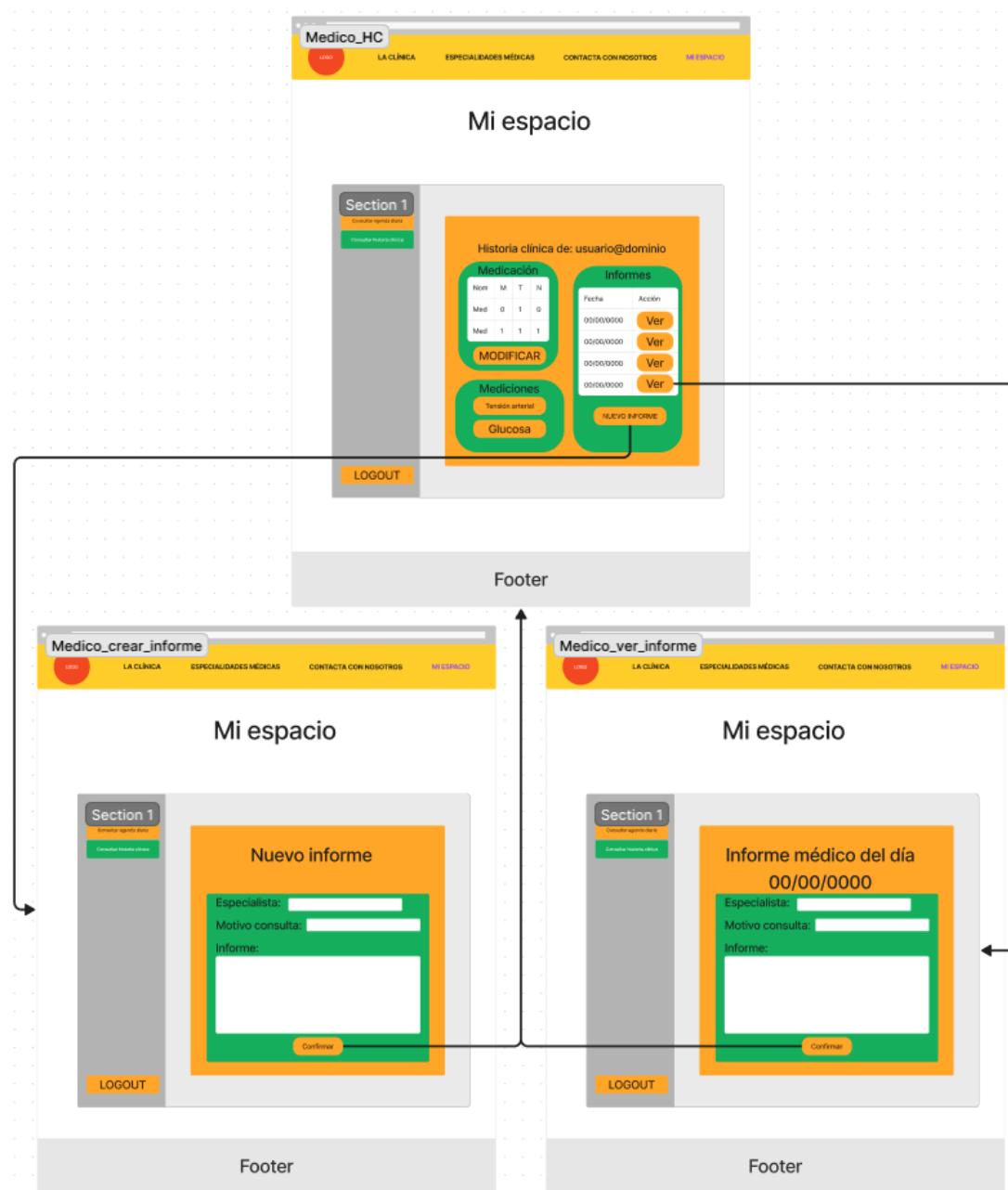


Fig 8 Prototipo para la consulta y generación de nuevos informes médicos.

D. Mi Espacio - Rol Paciente

1) Solicitar cita

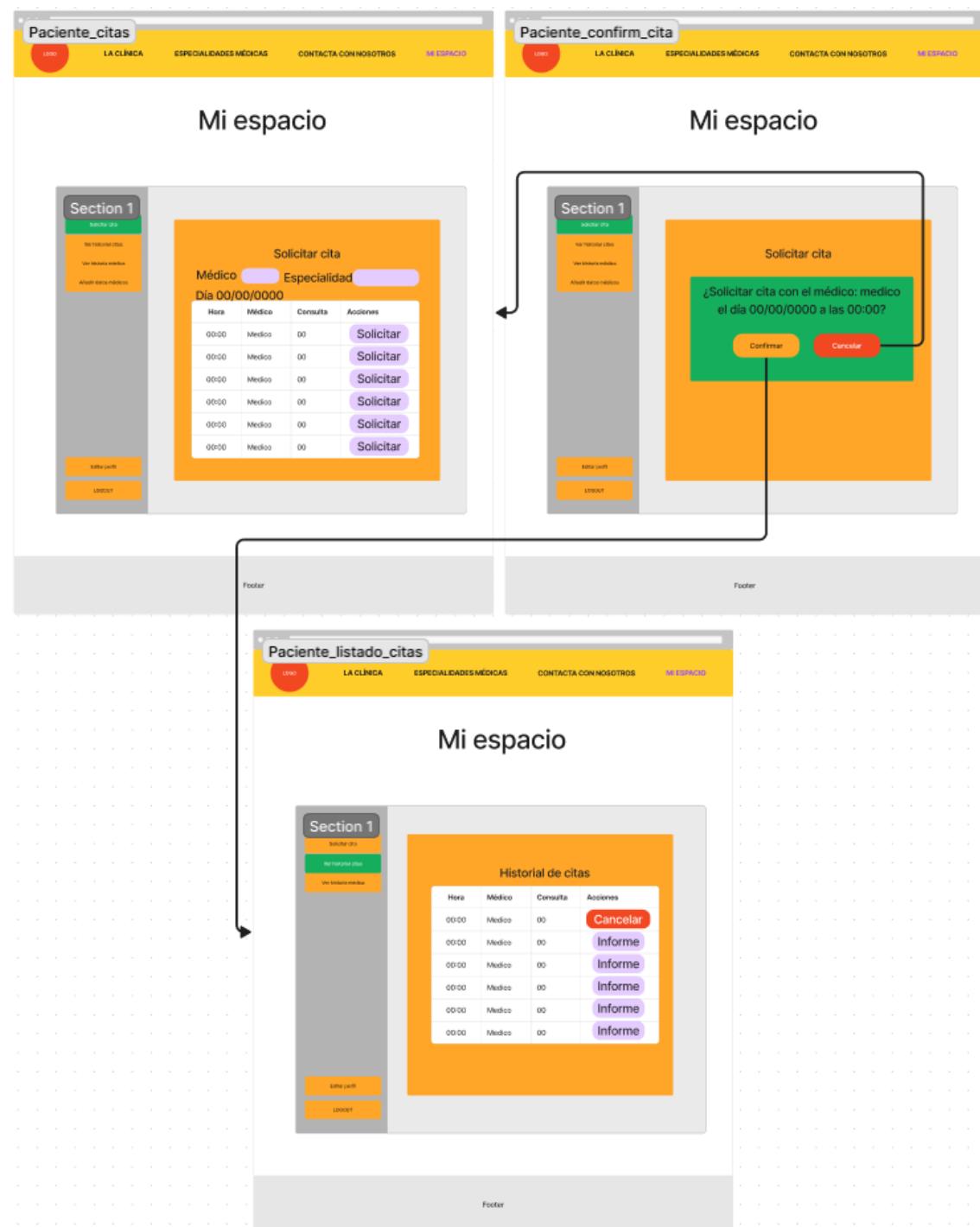


Fig 9 Prototipo para la solicitud de citas con especialistas por parte del paciente.

2) Historial de citas

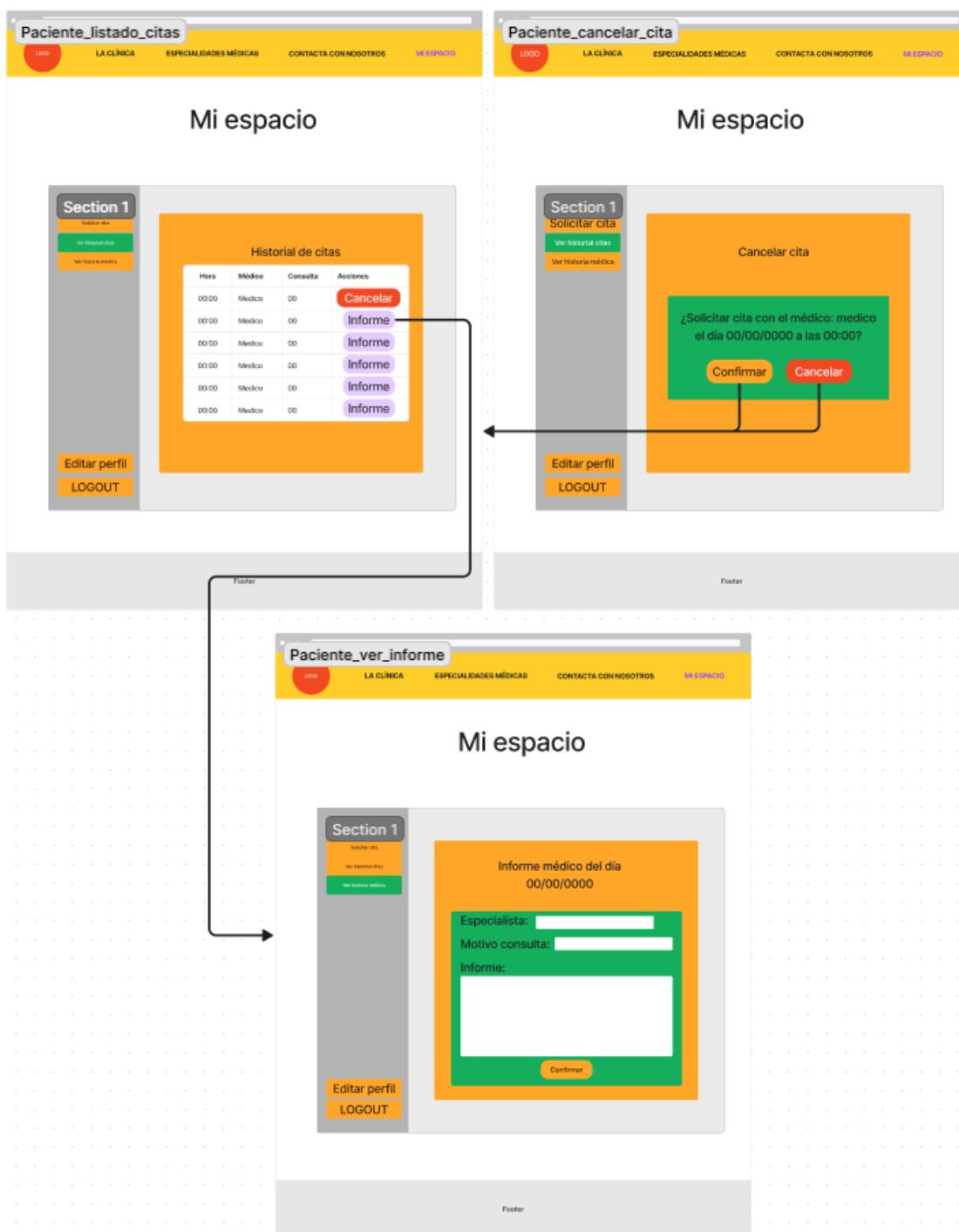


Fig 10 Prototipo para la gestión de citas por parte del paciente.

3) Historial clínico

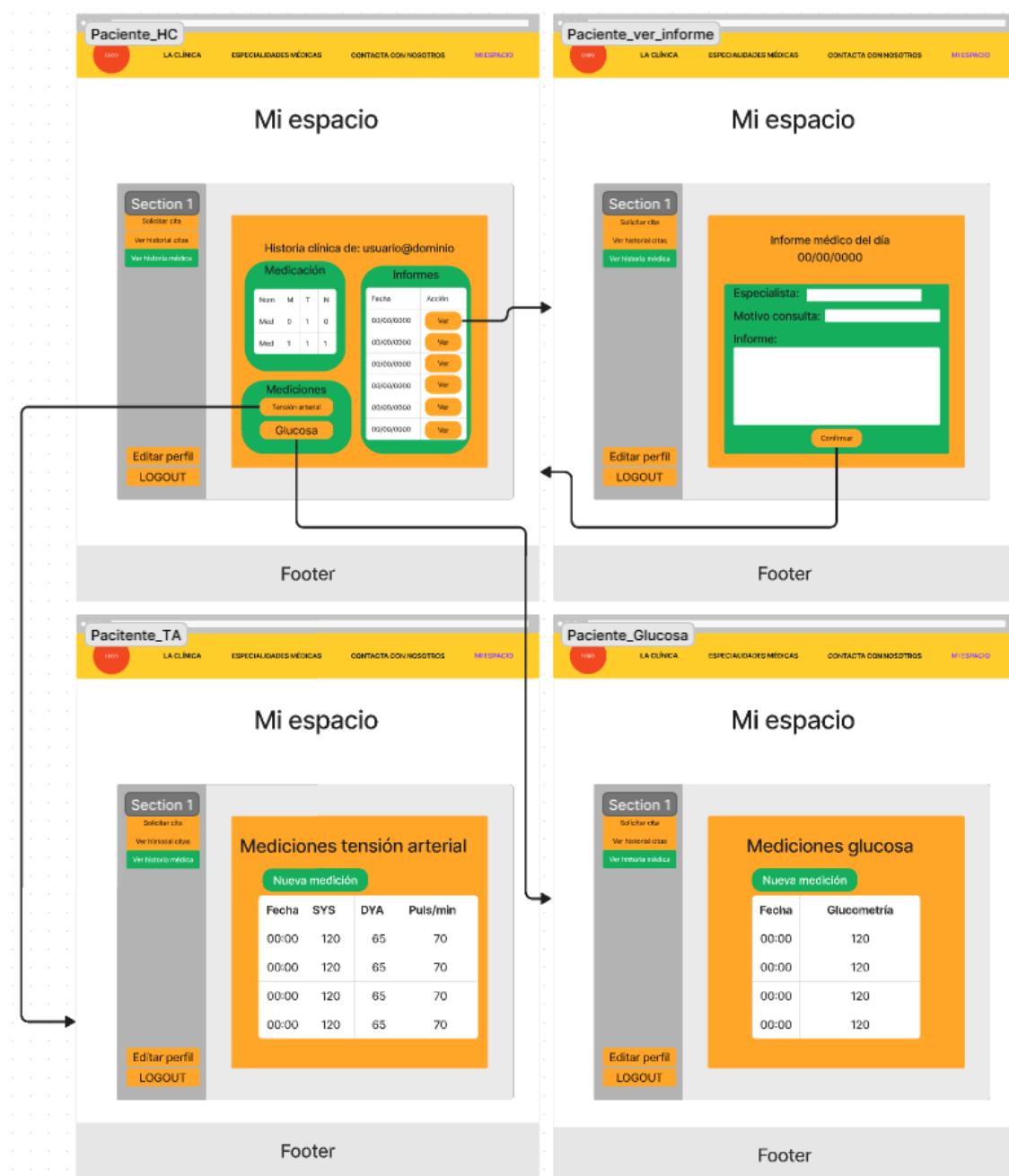


Fig 11 Prototipo para la inserción de datos de mediciones de tensión arterial y glucosa por parte del paciente.

4) Editar perfil

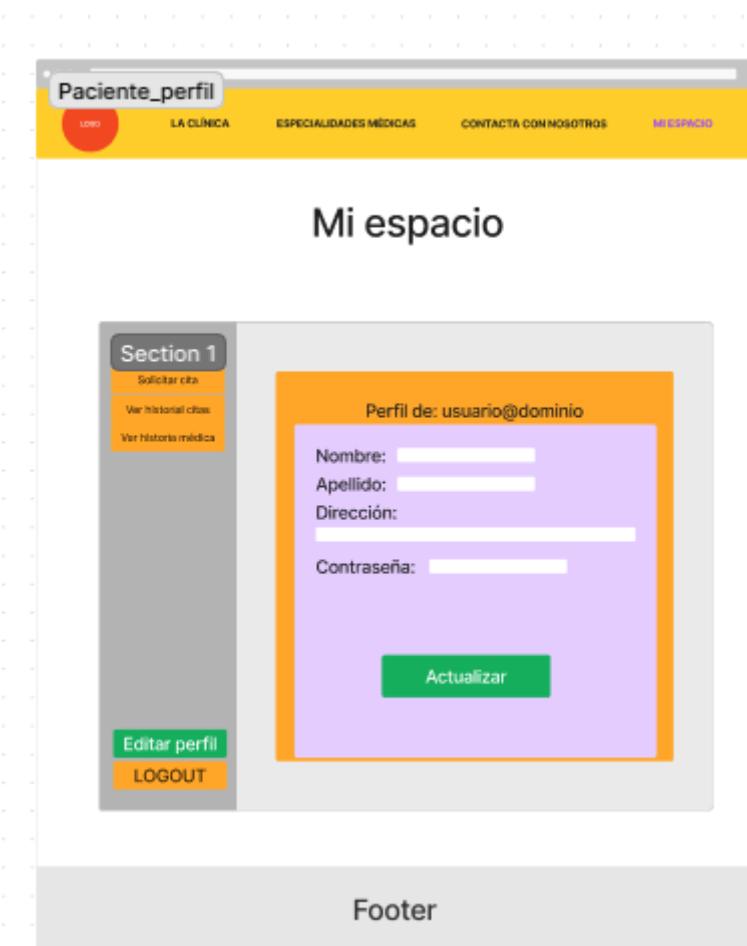


Fig 12 Prototipo para la edición del perfil por parte del paciente.

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.

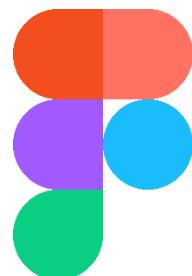


Fig 14 Logo de Figma.

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.

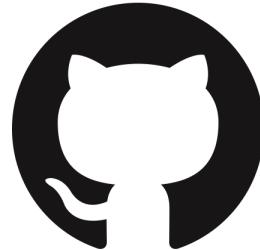


Fig 17 Logo de GitHub.

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.



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.

Fig 18 Logo de Express.js.

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,



Fig 20 Logo de MySQL.

confiabilidad y facilidad de uso, lo que lo convierte en una opción popular en el desarrollo de aplicaciones.

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.



Fig 21 Logo de MySQL Workbench.

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.

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 PL/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.



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 estilados.

Fig 23 Logo de Bootstrap.

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.

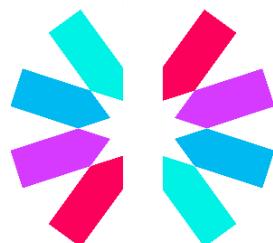


Fig 26 Logo de JSON Web Token.

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*.



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



Fig 28 Logo de Swagger UI.

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 rellenando 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



Fig 29 Logo de JSDoc.

del 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 compresión 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.

T. Ubuntu Server

Ubuntu Server es una distribución de Linux diseñada específicamente para servidores. Basada en Ubuntu, proporciona una plataforma sólida, segura y de alto rendimiento para la administración de servicios y aplicaciones en entornos de red. Es ampliamente utilizada en entornos empresariales, servidores web y en la nube debido a su fiabilidad y soporte a largo plazo.



Fig 32 Logo de Ubuntu Server.

La decisión de incluir Ubuntu Server en el proyecto se debe a su capacidad para manejar cargas de trabajo intensivas y su compatibilidad con una amplia gama de hardware y software.

U. Nginx

Nginx es un servidor web de alto rendimiento y un equilibrador de carga inverso que se utiliza ampliamente para servir contenido web estático y



Fig 33 Logo de Nginx.

dinámico. Diseñado para ofrecer una alta eficiencia y rendimiento, Nginx es conocido por su capacidad para manejar una gran cantidad de conexiones concurrentes con un uso mínimo

de recursos del sistema. Es una elección popular para la implementación de sitios web y aplicaciones web debido a su flexibilidad y robustez.

La decisión de incluir Nginx en el proyecto se debe a su capacidad para mejorar el rendimiento y la eficiencia de nuestra aplicación web, así como para proporcionar un equilibrio de carga y una distribución de tráfico efectivos. En nuestro caso, hemos utilizado Nginx como servidor web para poder desplegar la aplicación de Angular y realizar peticiones HTTP¹.

¹ NOTA: Se intentó realizar un despliegue que implicara la utilización del protocolo HTTPS, pero debido al uso de un certificado auto firmado, la comunicación entre el servidor web Nginx y el servidor de aplicaciones Node.JS se veía interrumpida provocando que el sistema de comunicación se rompiera. En el caso de realizar un despliegue real de la aplicación, habría que conseguir un certificado expedido por una agencia competente.

4. DIAGRAMAS DE ENTIDAD-RELACIÓN

A. Diagrama de Chen

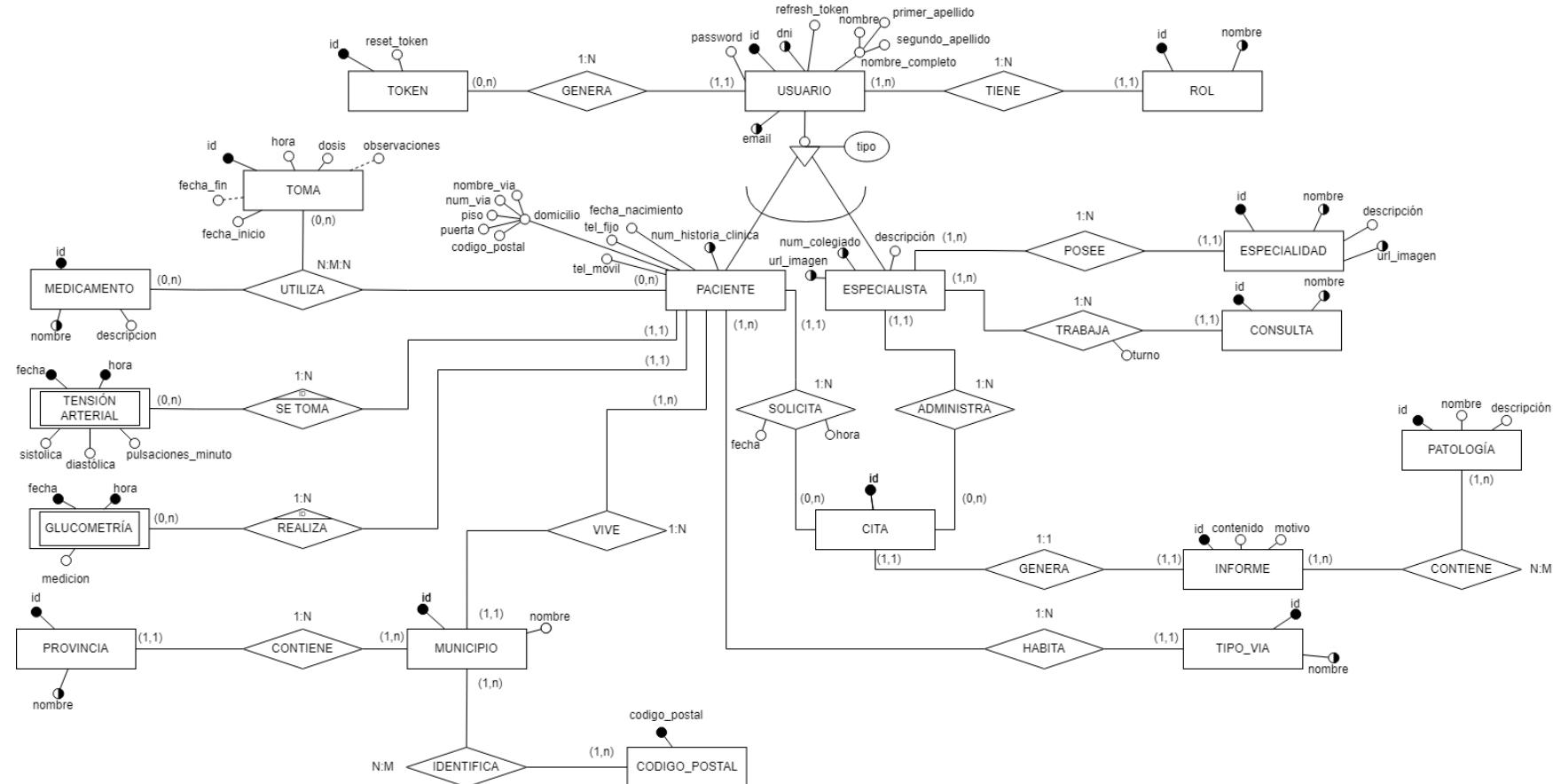


Fig 34 Diagrama de entidad - relación (diagrama de Chen).

B. Diagrama de estructura de datos

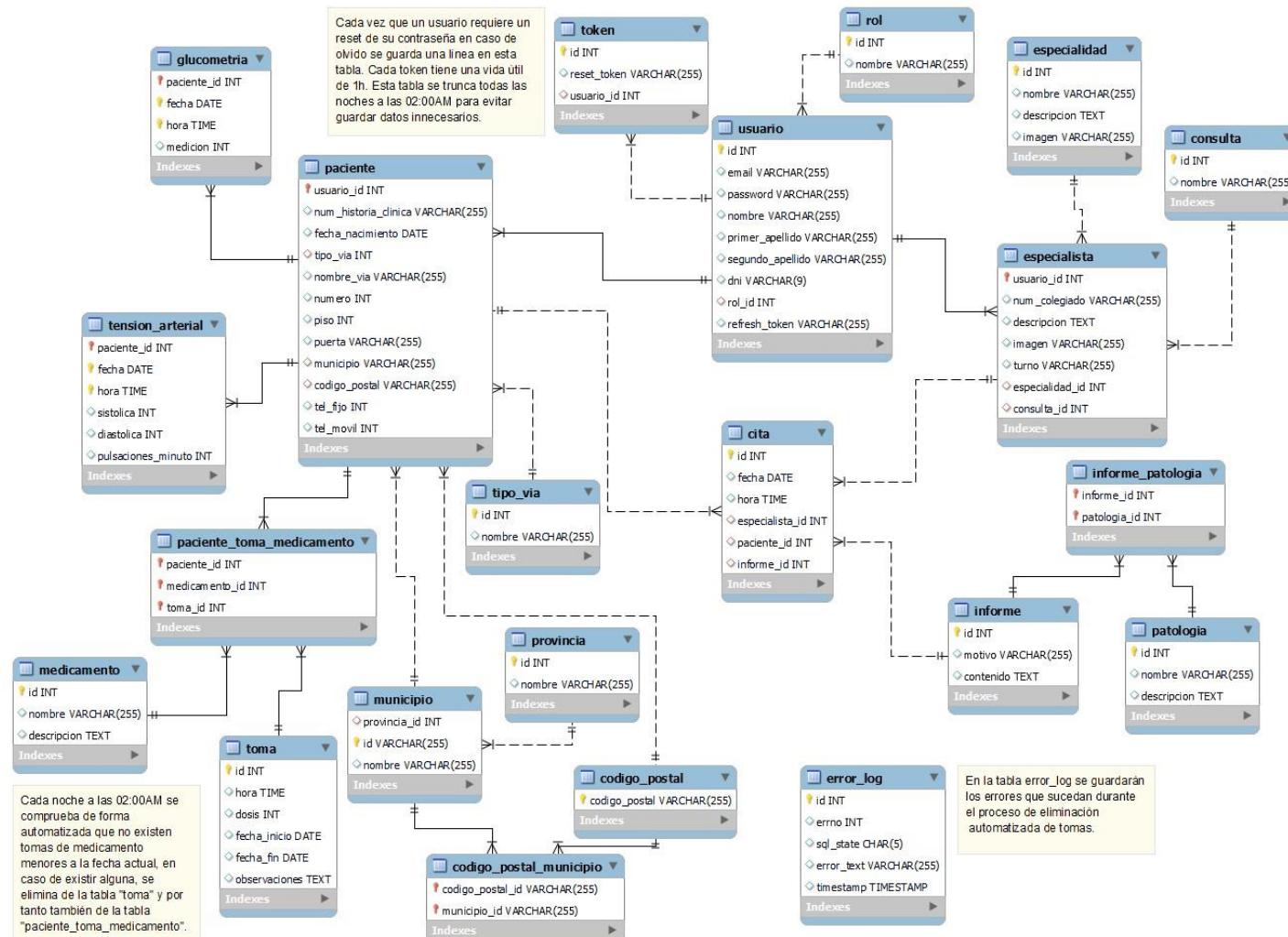


Fig 35 Diagrama de entidad-relación (diagrama de estructura de datos).

5. DIAGRAMA DE CASOS DE USO

A. Herencia de actores

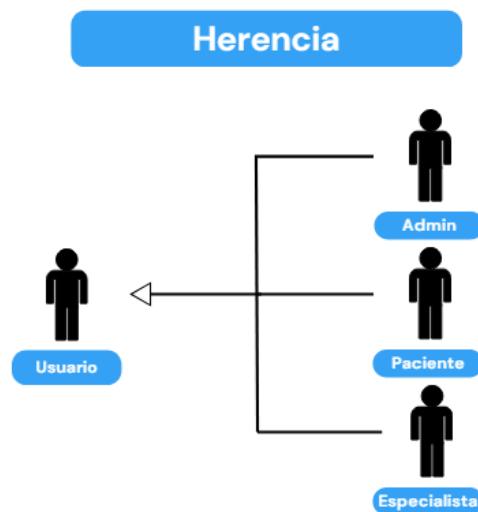


Fig 36 Herencia de actores del diagrama de casos de uso.

B. Casos de uso del usuario

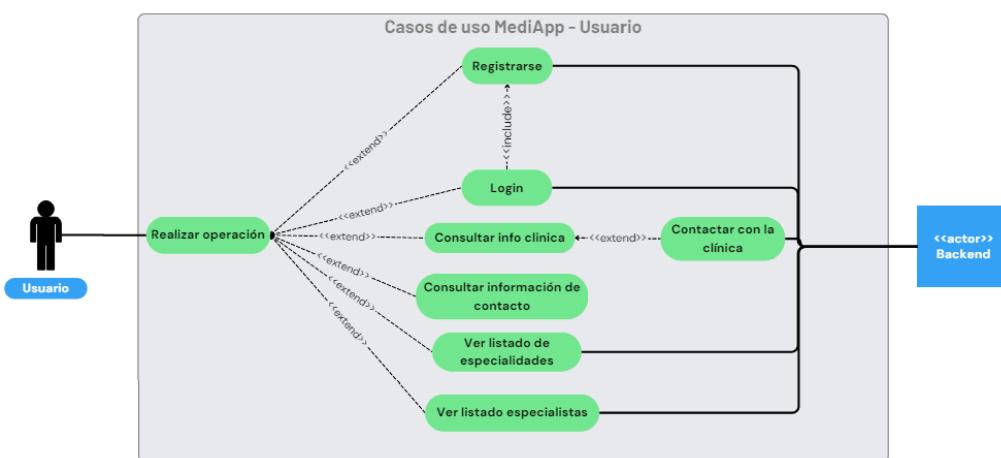


Fig 37 Diagrama de casos de uso del usuario.

C. Casos de uso del administrador

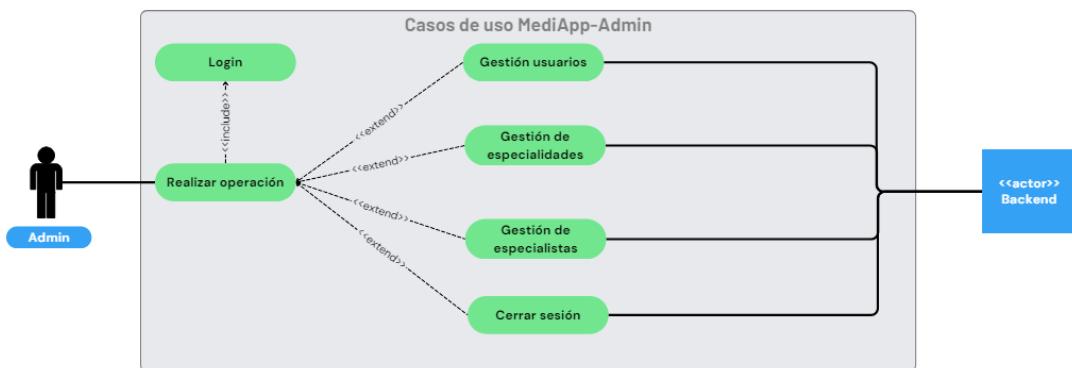


Fig 38 Diagrama de casos de uso del administrador.

D. Casos de uso del paciente

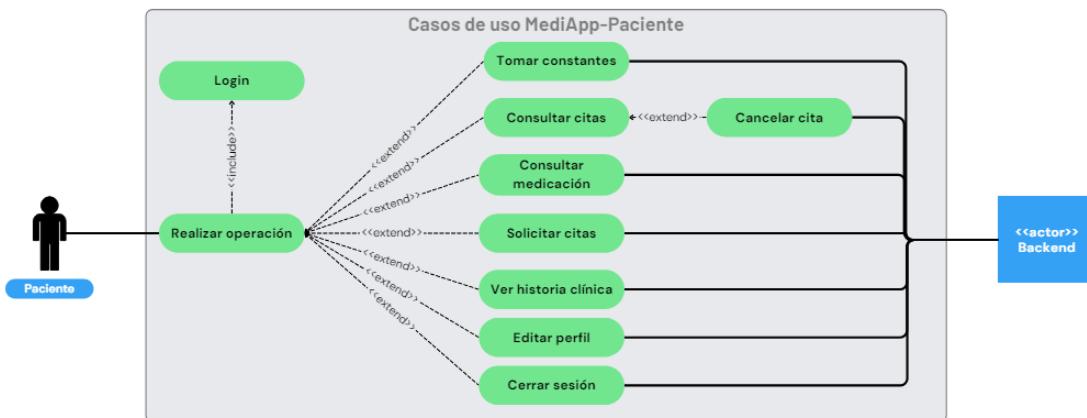


Fig 39 Casos de uso del paciente.

E. Casos de uso del especialista

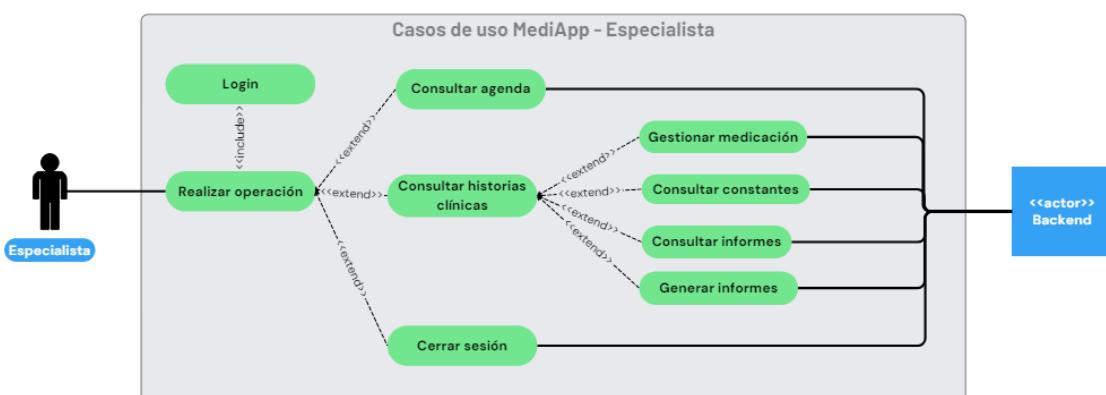


Fig 40 Casos de uso del especialista

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.

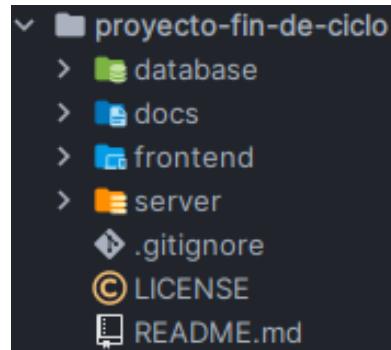


Fig 41 Estructura general del proyecto.

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 42 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.

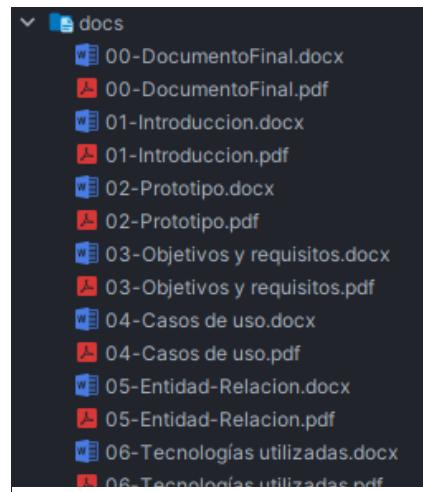


Fig 43 Estructura directorio 'docs'.

C. Directorio *frontend*

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

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

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

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

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 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

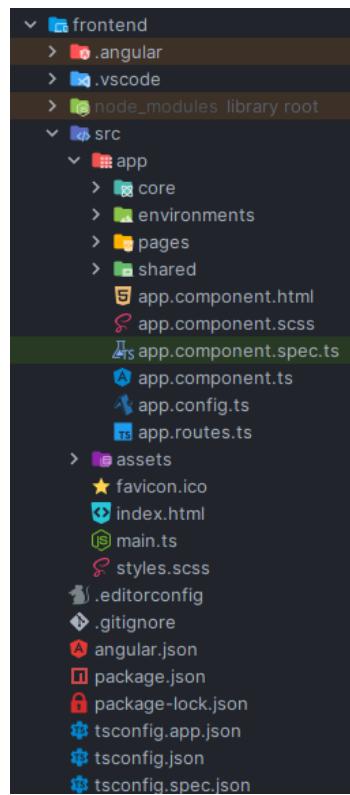


Fig 44 Estructura del directorio 'frontend'.

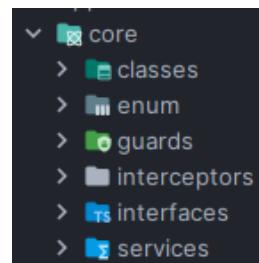


Fig 45 Subdirectorios del directorio 'core'.

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.

- 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:

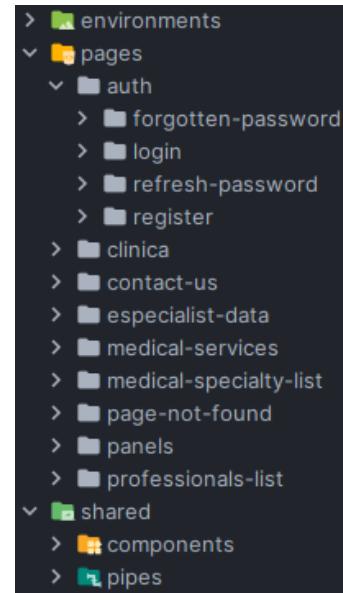


Fig 46 Subdirectorios 'enviroments', 'pages' y 'shared'.

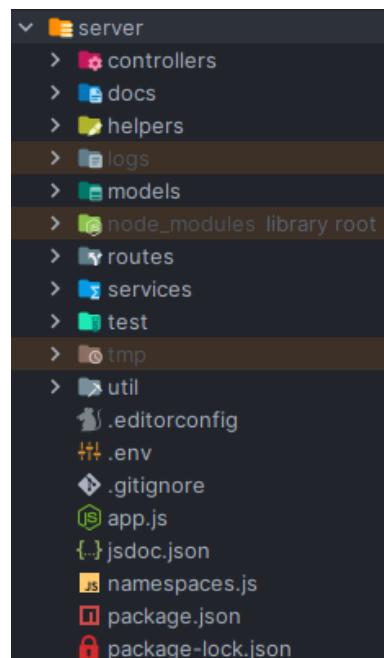


Fig 47 Estructura del directorio 'server'.

- a) **controllers:** En esta carpeta se almacenan los controladores 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) **logs:** Directorio que almacena los ficheros de log generados gracias a las librerías **morgan** y **rotating-file-stream**.
- e) **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.
- f) **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.
- g) **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.
- h) **test:** Contiene los archivos para las pruebas del servidor.

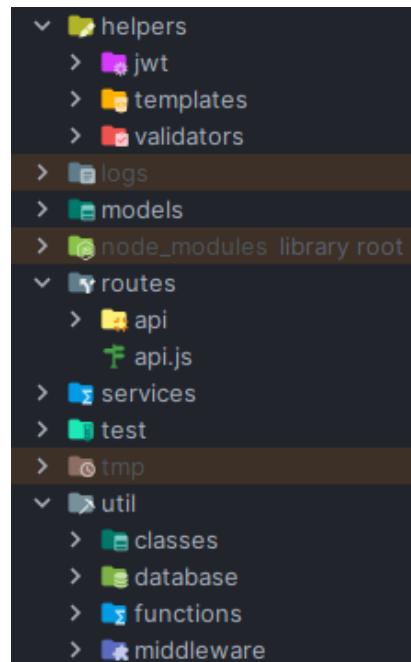


Fig 48 Estructura de los subdirectorios 'helpers', 'routes' y 'util'.

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

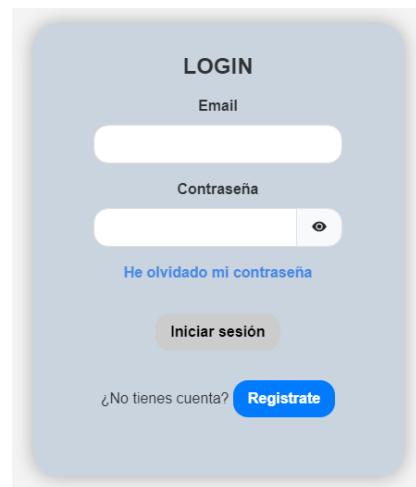
2. AUTENTICACIÓN, AUTORIZACIÓN Y CONTROL DE ACCESO

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).



LOGIN

Email

Contraseña

He olvidado mi contraseña

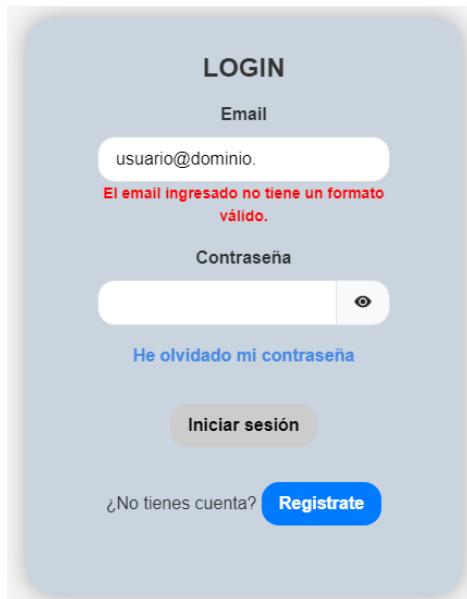
Iniciar sesión

¿No tienes cuenta? [Regístrate](#)

Fig 49 Formulario de login.

2) Validación de datos en frontend

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



LOGIN

Email

El email ingresado no tiene un formato válido.

Contraseña

He olvidado mi contraseña

Iniciar sesión

¿No tienes cuenta? [Regístrate](#)

Fig 50 Error en la introducción del email.

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

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

Fig 51 Generación del formulario reactivo de Angular para el login.

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

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

  return null;
}
```

Fig 52 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

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

  this.isLoading = true;

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

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

Fig 53 Método encargado de gestionar el login en el cliente.

petición HTTP a la API y generará un observable que contendrá la respuesta generada por el servidor.

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

Fig 54 Método encargado de comunicarse con el backend y llevar a cabo el login.

4) Activación del interceptor para el login

Esta petición HTTP dispara un interceptor, concretamente del LoginInterceptor.

```
export class LoginInterceptor implements HttpInterceptor {

  constructor(private authService: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req)
      .pipe(
        tap(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 55 Interceptor para el login del usuario.

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

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

5) Inicio de sesión en el servidor

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

6) Redirección al fichero de rutas

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

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

Fig 56 Ruta del servidor para el login.

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

7) Validación de datos en backend

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

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

```
export const validateUserLogin = [
  body('email')
    .trim()
    .notEmpty()
    .withMessage('El correo es requerido.')
    .custom((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.' });
      }

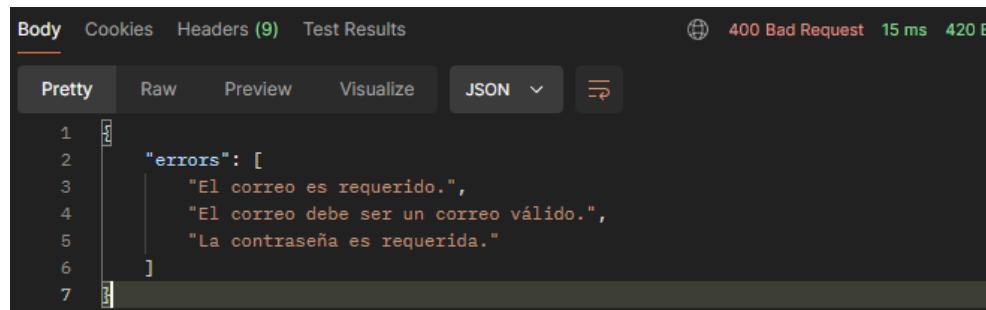
      return true;
    }),
  body('password')
    .trim()
    .notEmpty()
    .withMessage('La contraseña es requerida.')
    .isString()
    .withMessage('La contraseña debe ser una cadena de texto.')
    .escape(),
]

(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 57 Función de validación del servidor.

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



```

1
2   "errors": [
3     "El correo es requerido.",
4     "El correo debe ser un correo válido.",
5     "La contraseña es requerida."
6   ]
7
  
```

Fig 58 Ejemplo de una petición fallida utilizando Postman.

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

8) Llamada al controlador de usuarios

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

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

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

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

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

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

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

Fig 59 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.

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

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

    const validPassword = await compare(password, user.datos_personales.password);

    if (!validPassword) {
      throw new Error('Correo o contraseña incorrectos.');
    }

    const accessToken = TokenService.createAccessToken(user);
    const refreshToken = TokenService.createRefreshToken(user);

    await UserModel.updateRefreshToken(user.usuario_id, refreshToken, conn);

    return {
      accessToken,
      refreshToken
    };
  } catch (err) {
    throw err;
  }
}
```

Fig 60 Método de login del servicio.

```
// 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 61 Función de utilidad para la creación de un pool de conexiones a la base de datos.

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 62 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.

```
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 64 Función encargada de la firma del token de acceso.

Fig 65 Función encargada de la firma del token de refresco.

Cada uno de estos tokens tiene una función concreta:

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



Fig 63 Inicio de sesión fallido.

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

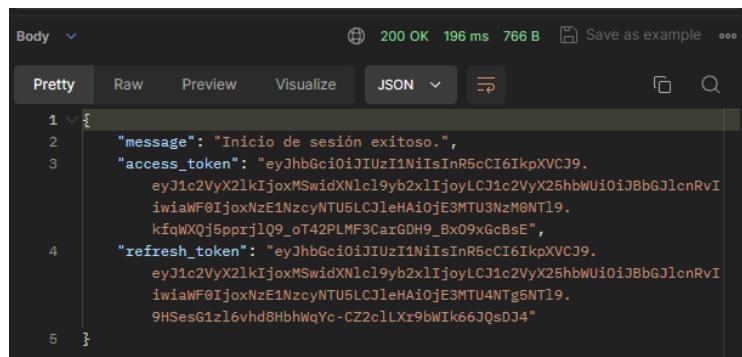
  return sign(payload, process.env.JWT_REFRESH_SECRET_KEY, options: {
    expiresIn: '1d',
  });
}
```

acceso (este proceso será comentado más adelante cuando se explique el funcionamiento de otros interceptores del cliente). En nuestro caso el token de refresco tiene un tiempo de vida de 1 día, durante ese tiempo el token puede ser utilizado para obtener nuevos tokens de acceso.

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

12) Envío de los tokens al cliente

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



```

Body
200 OK 196 ms 766 B Save as example ...
Pretty Raw Preview Visualize JSON ...
1 {
  "message": "Inicio de sesión exitoso.",
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJlc2VyX2lkIjoxMSwidXNlc19yb2xIjoyLCJ1c2VyX25hbWUiOjBbGJlcRvI
    iwiWF0IjoxNzE1NzcyNTU5LCJleHAIoje3MTU3NzM0NTl9.
    kfqWXQj5pprjlQ9_oT42PLMF3CarGDH9_Bx09xGcBsE",
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJlc2VyX2lkIjoxMSwidXNlc19yb2xIjoyLCJ1c2VyX25hbWUiOjBbGJlcRvI
    iwiWF0IjoxNzE1NzcyNTU5LCJleHAIoje3MTU4NTg5NTl9.
    9HSesG1z16vhd8HbwQYc-CZ2c1LXr9bWIk66JqsD4"
}

```

Fig 66 Ejemplo de respuesta desde el servidor ante un login correcto.

Y esta respuesta será interceptada por el LoginInterceptor tal y como se detalló en el punto 4 del presente apartado. Este interceptor se encargará de almacenar a nivel local del navegador tanto el token de acceso como el token de refresco utilizando los métodos **storeAccessToken()** y

storeRefreshToken() del servicio de autenticación.

```

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

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

```

Fig 67 Métodos del servicio de autenticación para almacenar los tokens.

http://localhost:4200	
Origin	Value
Key	Value
access_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2VyX2lkIjoxMSwidXNlc19yb2xIjoyLCJ1c2VyX3JvbGUQjEsInVzZXJfbmFtZSI6Ikp...
refresh_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2VyX2lkIjoxMSwidXNlc19yb2xIjoyLCJ1c2VyX3JvbGUQjEsInVzZXJfbmFtZSI6Ikp...

Fig 68 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 69 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 70 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

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

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

  return comprobarPatient;
};
```

Fig 71 Guardia de rol de paciente.

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

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

  if (!accessToken) {
    return null;
  }

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

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

Fig 72 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 usuario que estaba contenido en el token se encuentra entre los posibles que

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

Fig 73 Ejemplo de protección de rutas en Node.JS.

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 74 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 75 Mecanismo que impide que un paciente (role 2) pueda acceder a datos ajenos a los de su cuenta.

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

2) Verificación del rol de usuario

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

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

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

Fig 76 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 77 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 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.

```
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 78 Generación de la cabecera Authorization en el cliente.

Request Headers	Raw
Accept:	application/json, text/plain, */*
Accept-Encoding:	gzip, deflate, br, zstd
Accept-Language:	es-ES,es
Authorization:	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9eyJ1c2VyX2lkjoxLCJ1c2VyX3JvbGUJOjEsInVzZXRfbmFtZS16Ikpv2UiCjpxXQjOjE3MTU3ODkzMDoismV4cCl6MTcxNTc5MDlwN30.0XEP482sdo9xHSSzHNhqDHxfannWa1E-iXNN-KeMuC
Cache-Control:	no-cache
Connection:	keep-alive
Host:	localhost:3000
Origin:	http://localhost:4200
Pragma:	no-cache
Referer:	http://localhost:4200/
Sec-Ch-Ua:	"Chromium";v="124", "Brave";v="124", "Not-A.Brand";v="99"
Sec-Ch-Ua-Mobile:	?0
Sec-Ch-Ua-Platform:	"Windows"
Sec-Fetch-Dest:	empty
Sec-Fetch-Mode:	cors
Sec-Fetch-Site:	same-site
Sec-Gpc:	1

Fig 79 Captura de las herramientas de desarrollador donde se puede ver la cabecera Authorization.

E. Tratamiento del token de refresco

Un punto importante de todo el proceso de autenticación del usuario es el mantenimiento de su sesión para ello se utiliza el token de refresco, un token que como se vio en el apartado de inicio de sesión es largo y que en nuestro caso tiene una vida útil de 1 día, con esto se consigue que cuando el token de acceso expira, la sesión del usuario pueda continuar activa de forma completamente transparente para este y sin sufrir un cierre de sesión continuo que le obligue a iniciar la sesión cada poco tiempo.

1) Intercepción del error 401 desde el servidor.

Para conseguir que este proceso funcione de forma correcta vuelve a ser importante la acción de los interceptores de Angular que, en este caso, lo que harán será manejar las respuestas

401 respondidas por el servidor cuando se produzca una expiración del token de acceso tal y como se explicó en la sección correspondiente a la verificación del token de acceso por parte del servidor.

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

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

Fig 80 Método principal del interceptor encargado del refresco de token.

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

```
private handle401Error(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    if (!this.isRefreshing) {
        this.isRefreshing = true;
        this.refreshTokenSubject.next({ value: null });

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

Fig 81 Método encargado de gestionar el error 401.

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

Si esta solicitud es respondida de forma exitosa, se ejecuta el bloque de código que se encuentra dentro del **switchMap** emitiendo un nuevo token de acceso a través del sujeto

refreshTokenSubject, a continuación, se maneja la solicitud original que falló debido al error 401 utilizando el método **addToken()**.

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

Fig 82 Método encargado de reciclar la petición original que provocó el 401.

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

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

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

2) Llamada al servicio de autenticación

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

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

```

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

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

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

```

Fig 83 Método del servicio de autenticación encargado de solicitar un token de refresco.

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

3) Comunicación con el servidor

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

```

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

```

Fig 84 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

```
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 85 Método del controlador encargado del refresco del token.

404 y una vez que sea manejado por el cliente se producirá el cierre de sesión.

5) Funcionamiento del servicio

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

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

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

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

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

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

Fig 86 Método del modelo encargado de devolver el token de refresco.

Si todo ha sido correcto se generarán los nuevos tokens y se actualizará el token de refresco en el registro del usuario, por último, los nuevos tokens serán devueltos al controlador el cual los devolverá al cliente que se encargará de almacenarlos de forma segura en el navegador.

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

    const decodedToken = TokenService.verifyRefreshToken(refreshToken);

    if (!decodedToken) {
      throw new Error(message: 'El token no es válido.');
    }

    const userId = decodedToken.user_id;
    const userToken = await UsuarioModel.getRefreshTokenById(userId, conn);

    if (!userToken) {
      throw new Error(message: 'El usuario no existe.');
    }

    if (userToken.refresh_token !== refreshToken) {
      throw new Error(message: 'El token no es válido.');
    }

    const user = await UsuarioModel.findById(userId, conn);

    const newAccessToken = TokenService.createAccessToken(user);
    const newRefreshToken = TokenService.createRefreshToken(user);

    await UsuarioModel.updateRefreshToken(decodedToken.user_id, newRefreshToken, conn);

    return {
      new_access_token: newAccessToken,
      new_refresh_token: newRefreshToken,
    };
  } catch (err) {
    throw err;
  }
}
```

Fig 87 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.

FORMULARIO RECUPERACIÓN CONTRASEÑA

Correo electrónico

Introduce tu correo electrónico

Confirmar

Fig 88 Formulario de recuperación de contraseña.

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

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



Enhorabuena

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

OK

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 90 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 recibido.

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

Fig 91 Enrutador para la contraseña olvidada.

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 92 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 93 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 94 Servicio encargado del reinicio de contraseña.

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

```
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 95 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.

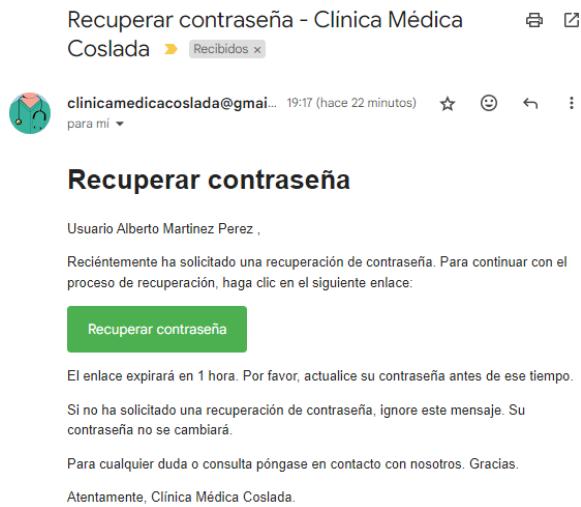


Fig 96 Email de recuperación de contraseña.

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

Fig 97 Ruta del enrutador de Angular para continuar el proceso.

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

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

Fig 98 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.



FORMULARIO RECUPERACIÓN CONTRASEÑA

Nueva contraseña

Introduce la nueva contraseña

Repite la contraseña

Repite la contraseña

Fig 99 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(`/${this.apiUrl}/usuario/contrasena-reset`, passwordRefresh)
      .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 100 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 101 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 102 Método del controlador para continuar el proceso de reinicio de contraseña.

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

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

    const encryptedPassword = await createEncryptedPassword(password);
    await UsuarioModel.updatePassword(user.datos_personales.email, encryptedPassword, conn);
  } catch (err) {
    throw err;
  }
}
```

Fig 103 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

```
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 104 Método de utilidades para la encriptación de contraseñas.

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

G. Cierre de sesión

1) Cierre de sesión en el cliente

A la hora de cerrar la sesión del usuario, este pulsará el botón de cierre de sesión de su panel de usuario, el cual lanzará el método asociado en el componente que será el encargado de comunicarse con el servicio y realizar la petición POST al servidor para cerrar la sesión.

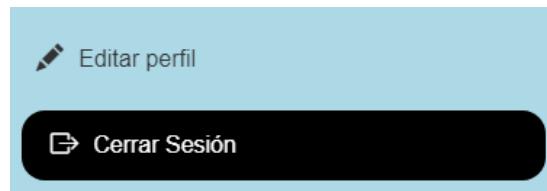


Fig 105 Botón de cierre de sesión.

```
onLogout(): void {
  this.auth.logout().subscribe( observerOrNext: {
    next: (): void => {
      this.router.navigate( commands: ['L'] ) Promise<boolean>
        .then((): void => {} ) Promise<void>
        .catch((): void => {} );
    },
    error: (error): void => {
      console.error(error);
    }
  });
}
```

Fig 106 Método de cierre de sesión en el componente sidebar del cliente.

2) Interceptor de logout

Para llevar a cabo de forma efectiva el cierre de sesión del usuario hemos creado un interceptor que se encargará de recibir todas las respuestas que provengan de la URL con terminación /usuario/logout.

```
export class LogoutInterceptor 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/logout')) {
          this.authService.removeTokens();
          this.authService.loggedInUser.next( value: false );
        }
      })
    );
  }
}
```

Fig 107 Interceptor para el cierre de sesión.

3) Manejo del cierre de sesión en el servidor

En el servidor una vez que la petición es recibida por el fichero app.js y es redirigida al enrutador correspondiente, el controlador comunica al servicio de usuario que se quiere cerrar la sesión desde el cliente y, usando el ID que se recupera del token de acceso, se comunica con el modelo de usuario para actualizar el registro del usuario en cuestión asignando el valor NULL al campo refresh_token de la tabla.

```
static async deleteRefreshToken(userId, dbConn) {
  const query =
    'UPDATE `usuario` '
    .SET `refresh_token = NULL`
    .WHERE `id = ?`;

  try {
    return await dbConn.execute(query, [userId]);
  } catch (err) {
    throw new Error(`message: 'Error al eliminar el token de refresco.'`);
  }
}
```

Fig 108 Método de actualización del valor del token de refresco a nulo.

4) Eliminación de los tokens del almacenamiento local

Si la respuesta del servidor es positiva, el interceptor de logout se comunica con el servicio de autenticación e

```
removeTokens(): void {
  localStorage.removeItem('access_token');
  localStorage.removeItem('refresh_token');
}
```

Fig 109 Eliminación de los tokens del almacenamiento local.

invoca el método **removeTokens()** y, además, cambiará el valor del sujeto de comportamiento **loggedInUser** a false para comunicar a toda la aplicación (todos los componentes que estén suscritos a él) del cambio de estado en la situación de login del usuario.

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

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

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

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

Fig 111 Variables de entorno para las CORS.

```
✖ 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 112 Error de CORS si se realiza una petición desde un origen no permitido.

3. PETICIONES CLIENTE - SERVIDOR

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

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

```

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

```

Fig 113 Ejemplo de ruta GET.

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

```

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

```

Fig 114 Ejemplo de ruta POST.

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

```

442
450 router.delete(
451   '/especialidad/:especialidad_id',
452   verifyAccessToken,
453   verifyUserRole([1]),
454   validateEspecialidadIdParam,
455   EspecialidadController.deleteEspecialidad,
456 );
457

```

Fig 115 Ejemplo de ruta DELETE.

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

```

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

```

Fig 116 Ejemplo de ruta PUT.

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

A. Método GET

1) Envío de datos desde el servidor

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

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

Fig 117 Petición GET de citas.

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

Tras esto, en el *backend* comprobaremos que el usuario está logado y que cumple con el rol necesario para poder desempeñar la acción de recogida de datos. A su vez comprobaremos que el id del usuario es

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

Fig 118 Comprobación de los parámetros de búsqueda.

enviado en el formato correcto al igual que los parámetros de filtrado para la paginación.

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

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

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

Fig 119 Petición para actualizar las especialidades existentes.

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

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

Fig 120 Parámetros de 'page', 'limit' y 'search'.

Volviendo a nuestro ejemplo de recogida de citas con paginación, este tiene un filtro que permite buscar citas entre un rango de fechas.

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

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

Si no hemos recibido la

Fig 121 Parámetros 'fechaInicioCita' y 'fechaFinCita'.

fecha de inicio, su valor será el primer día del año actual, es decir, el 01/01/XXXX. Por otra parte, si no hemos recibido una fecha máxima límite , recogeremos todas las citas que ya han

sucedido o planificado hasta el día actual de dentro de 3 años (con esto nos aseguramos de mostrar todas las citas previstas para corto y largo plazo de un paciente). Luego, devolvemos un objeto con todos los filtros para tener en cuenta para buscar las citas.

```

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

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

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

Fig 122 Funcionamiento del método `readCitas()`.

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

2) Búsqueda de datos en la base de datos

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

3) Generación de la paginación en el servidor

Por último, preparamos los parámetros a utilizar en la paginación del *frontend* los cuales son:

```

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

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

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

Fig 123 Consulta de citas.

- **Next:** Contiene la URL con la llamada a la siguiente página y filtros actuales (en este caso el rango de fechas). Si esta no es posible tomará un valor null.
- **Prev:** Lo mismo que el parámetro next pero en este caso contiene la llamada a la página anterior. Si esta es menor o igual que cero devuelve null ya que no existen páginas negativas.
- **Página_actual:** Indica en qué página estamos ubicados actualmente.
- **Páginas_totales:** Muestra el número total de páginas.
- **Cantidad_citas:** Indica el número total de citas en la llamada realizada.
- **Items_pagina:** Nos permite establecer distinto número de ítems por página, aunque el por defecto sea 10.
- **Fecha_inicio:** Fecha inicial del filtrado por fechas.
- **Fecha_fin:** Fecha máxima / límite del filtrado por fechas.
- **Resultados:** El conjunto de citas recogidas en la página.

```

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

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

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

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

```

Fig 124 Página en el servidor.

4) Recepción y presentación de los datos en el cliente

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

The screenshot shows a user interface for managing appointments. At the top, there is a search form with fields for 'Fecha inicio' (dd/mm/aaaa) and 'Fecha fin' (dd/mm/aaaa), both with calendar icons, and a dropdown for 'Items' set to 10. Below the form, a message says 'Estas son las citas, Anais'. A table displays 12 appointment entries, each with columns: Hora, Fecha, Consulta, Especialista, Especialidad, and Acción. The table shows entries for Patricia Medina Navarro from 09:00 to 12:00 on various dates in August and May. At the bottom, there is a pagination control with links for 'Anterior', 'Siguiente', and page numbers 1 and 2.

Hora	Fecha	Consulta	Especialista	Especialidad	Acción
12:00:00	03-09-2024	5-E	Patricia Medina Navarro	Hematología	...
12:00:00	01-09-2024	5-E	Patricia Medina Navarro	Hematología	...
09:00:00	23-08-2024	5-E	Patricia Medina Navarro	Hematología	...
09:00:00	23-05-2024	5-E	Patricia Medina Navarro	Hematología	...
09:00:00	22-05-2024	5-E	Patricia Medina Navarro	Hematología	...

Fig 125 Listado con paginación y filtrado.

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

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

```
ngOnInit(): void {
  this.actualPage = 1;

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

Fig 126 Uso de objeto **Subject** en la función `ngOnInit` para recoger las citas nada más cargue el componente.

```
getCitas(fechaInicioCita: string, fechaFinCita: string, perPage: number, page: number) {
  let query: string = `?page=${page}`;

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

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

  if (perPage) {
    query += `&limit=${perPage}`;
  }

  return this.http.get<CitasListModel>(` ${this.baseUrl}/cita${query}`);
}
```

Fig 127 Llamada al método `getCitas()` del servicio de citas.

Este método es ligeramente distinto a los vistos previamente ya que, al tener filtros , tenemos que comprobar que si la llamada manda parámetros que no sean nulos (es decir , cuando el usuario haya seleccionado un filtro) que sean enviados al *backend* para tener en cuenta esos

filtros como criterios de búsqueda ya que si son nulos no tiene sentido tenerlos como parámetros de búsqueda. Por otro lado, algo importante a destacar es que al tratarse de una ruta GET esta no puede enviar objetos como tal al *backend*, sino que tiene que hacerlo a través de variables de URL.

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

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

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

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

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

Fig 128 Método `getCitas()`.

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

Fig 129 Método `showResults()`.

5) Manejo de la paginación en el cliente

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

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

Fig 130 Funcionamiento de `paginate` en bucle `for` de Angular.

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

```
<div *ngIf="previousPageUrl || nextPageUrl" class="pagination">
    <pagination-controls (pageChange)="actualPage = $event; changePage(actualPage)"
        id="citaslistPagination"
        maxSize="5"
        directionLinks="true"
        previousLabel="Anterior"
        nextLabel="Siguiente"></pagination-controls>
</div>
```

Fig 131 Componente pagination-controls.

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

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

Fig 132 Método changePage().

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

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

Fig 133 Método updateFilters().

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

Fig 134 Inputs de filtrado.

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

B. Ruta POST

1) Recogida de datos en el cliente

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

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

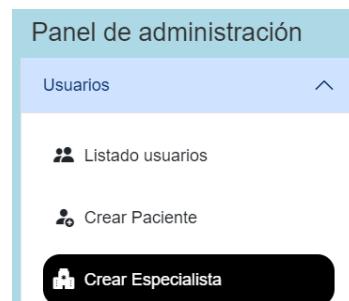


Fig 135 Funciones relativas a listado y creación de usuarios

Fig 136 Formulario creación especialista

Fig 137 Formulario creación paciente.

Estos formularios, a su vez cuentan con lo conocido como **FormsGroup** y **FormsControl**, los cuales son unas directivas de Angular que facilita el manejo de formularios al poder acceder

directamente a los elementos de cada input al igual que permite comprobar si los datos insertados son correctos entre otras funcionalidades.

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

Fig 138 Estructura FormGroup con FormControls.

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

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

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

Fig 139 FormControl que comprueba si los campos son válidos.

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

    @if (registerForm.get('primer_apellido').invalid) {
      <li>
        <strong>Primer apellido:</strong> Debe empezar por mayúscula y
        el resto de letras en minúsculas, salvo en el caso de apellidos
        compuestos.
      </li>
    }
  </ul>
</section>
```

Fig 140 FormControl comprueba si cada campo es invalido e imprime su mensaje correspondiente.

Por favor, rellena todos los campos correctamente:

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

Fig 141 Mensajes de error de cada campo al ser inválidos.

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

2) Manejo de ficheros. Transformación a base64.

Llegados a este punto es importante comentar como hemos transformado la imagen que sube el usuario a **base64** para su posterior almacenamiento en la base de datos.

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

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

Fig 142 Atributos de la interfaz **EspecialistModel**.

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

Fig 143 Input que recoge las imágenes subidas al formulario.

Para ello, hemos creado el método **onFileSelect()** el cual comprueba que hemos subido una imagen y a continuación, llamaremos al método **toBase64()** del servicio

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

Fig 144 Método **onFileSelect()**.

FileUpload el cual se encargará de transformar la imagen en una URL de metadatos mediante

el método **reader.readAsDataURL**. Con esto habremos conseguido que en la variable **imageBase64** se guarde una cadena con los metadatos necesarios para almacenar en la base de datos y luego reconstruirlos en el lado del cliente.

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

Fig 145 Método toBase64().

3) Generación del modelo de datos y comunicación al servidor

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

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

Fig 146 Construcción objeto EspecialistaModel.

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

Fig 147 Llamada al servicio de AuthService para registrar un especialista.

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

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

Fig 148 Método del AuthService que llama a la ruta de registro de especialistas.

4) Manejo de datos en el servidor

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

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

Fig 149 Métodos de verificación de token del usuario y su rol.

```
export const validateEspecialistaRegister = [
  validateUserRegister,
  body('datos_especialista.num_colegiado')
    .trim()
    .notEmpty()
    .withMessage('El número de colegiado es requerido.')
    .isNumeric()
    .withMessage('El número de colegiado debe ser un valor numérico.')
    .custom((value) => {
      const regex = /^\d{9}$/;

      if (!regex.test(value)) {
        throw new Error({ message: 'El número de colegiado debe tener 9 dígitos.' });
      }

      if (value < 1) {
        throw new Error({ message: 'El número de colegiado no puede ser 0 o negativo.' });
      }

      return true;
    })
    .escape(),
  body('datos_especialista.descripcion')
    .trim()
    .notEmpty()
    .withMessage('La descripción es requerida.')
    .isString()
    .withMessage('La descripción debe ser una cadena de texto.'),
  body('datos_especialista.turno')
    .trim()
    .notEmpty()
    .withMessage('El turno es requerido.')
    .isString()
    .withMessage('El turno debe ser una cadena de texto.')
    .escape(),
];
```

Fig 150 Validación de los campos del objeto EspecialistaModel enviado al backend.

Mención especial al método **escape()** de los validadores que elimina los caracteres <, >, &, ', " y / que vayan acompañados de entidades HTML para evitar de esta manera la inyección de HTML.

Debido a que se ha utilizado la librería **ngx-quill** en el lado del cliente para conseguir unos componentes tipo procesador de textos, ciertos campos como “descripción” no son sanitizados con escape() en el validador sino con una función util que permite ciertas entidades HTML cuyo nombre es **sanitizeInput()**.

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

```
export const sanitizeInput = (input) => {
  return sanitizeHtml(input, options: {
    allowedTags: [
      'p',
      'br',
      'a',
      'b',
      'strong',
      'i',
      'em',
      'u',
      's',
      'ol',
      'ul',
      'li',
      'h1',
      'h2',
      'h3',
      'h4',
      'h5',
      'h6',
      'blockquote',
      'code',
      'pre',
      'sub',
      'sup',
      'small',
    ],
    allowedAttributes: {
      a: [ 'href' ]
    }
  });
}
```

Fig 151 Método **sanitizeInput()**.

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

```
static async postRegistro(req, res) {
  const errors = [];
  try {
    await UsuarioService.createUsuario(req.body);

    return res.status(200).json({ message: 'Usuario creado exitosamente.' });
  } catch (err) {
    if (err.message === 'El correo electrónico ya está en uso.') {
      errors.push(err.message);
    }

    if (err.message === 'El DNI ya está en uso.') {
      errors.push(err.message);
    }

    if (err.message === 'El número de colegiado ya está en uso.') {
      errors.push(err.message);
    }

    if (errors.length > 0) {
      return res.status(409).json({ errors });
    }
  }
  return res.status(500).json({ errors: [err.message] });
}
```

Fig 152 Método **postRegistro()** encargado de llamar al servicio de usuarios para que guarde al usuario en cuestión.

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

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

```

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

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

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

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

    if (dniExists) {
        throw new Error('El DNI ya está en uso.');
    }

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

        if (colegiadoExists) {
            throw new Error('El número de colegiado ya está en uso.');
        }
    }
}

```

Fig 153 Creación de la transacción y comprobación que el especialista no exista con anterioridad.

C. Método PUT

1) Edición de datos en el cliente

Como ya se ha mencionado con anterioridad, el método PUT se suele utilizar para la actualización de datos , en este caso nos volvemos a encontrar con el mismo componente del formulario para el registro de especialistas ya que se ha

Fig 154 Formulario de edición de especialista.

reutilizado para también usarlo para su edición. Es por ello por lo que nada más clicar en la opción de editar especialista nos sacará ya el formulario con los datos rellenos de dicho especialista (haciendo una petición GET al servidor para coger los datos del especialista).

Al igual que en el caso de la ruta POST, si clicamos en editar, el formControl comprobará que todos los datos son correctos y si es correcto, se creará el objeto

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

Fig 155 Llamada al método updateSpecialist().

que contenga todos los datos de especialista (Model). Luego, en vez de llamar al método registerEspecialist() como sucedía con el registro de Especialista , pasaremos a llamar al método updateEspecialist() al querer actualizar sus datos.

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

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

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

Fig 156 Método updateSpecialist() del servicio de autenticación.

2) Comunicación al servidor

Tras esto, el servicio le pasa los datos a la ruta correspondiente en el *backend*.

Como siempre , tras comprobar que el usuario está logado y que

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

Fig 157 Verificación de token y rol en la ruta PUT del back

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

```

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

```

Fig 158 Comprobación ID del especialista.

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

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

```

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

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

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

```

Fig 159 Establecimiento de la conexión y creación de la transacción.

operaciones en la base de datos con total seguridad.

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

```

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

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

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

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

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

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

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

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

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

```

Fig 160 Comprobación de existencia del correo, contraseña y número de colegiado del especialista.

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

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

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

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

const usuario = ObjectFactory.updateUserObject(data);

await UsuarioModel.updateUsuario(usuario, conn);

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

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

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

Fig 161 Actualización del usuario por medio del objeto UsuarioModel.

D. Método DELETE

1) Manejo en el cliente

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

Mostrando 1 - 10 de 15 especialidades.			
Nombre	Imagen	Descripción	Acciones
Cardiología		Especialidad médica dedicada al estudio del corazón	⋮
Dermatología		Especialidad médica que se ocupa de la piel	editar eliminar
Endocrinología		Especialidad médica que estudia las glándulas que producen las hormonas	⋮
Racteranterior		Fenómeno óptico que ocurre al efectuar rotación	⋮

Fig 162 Vista componente listado de especialidades.

En cuanto a la lógica del frontend, tenemos asignado un evento que al pulsar en el botón “eliminar” salga un modal que confirme si el



Fig 163 Modal de confirmación de eliminación.

administrador está seguro de querer eliminar dicho registro.

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

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

Fig 164 Método confirmarCancelacion().

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

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

Fig 165 Método eliminateSpeciality() del servicio de especialidades.

2) Eliminación en el servidor

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

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

Fig 166 Comprobación token y rol de la ruta DELETE del back.

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

Fig 167 Validación del tipo y existencia de ID de la especialidad.

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

```
static async deleteEspecialidad(id, conn = dbConn) {
  try {
    const idExistente = await EspecialidadModel.findById(id, conn);

    if (!idExistente) {
      throw new Error('Especialidad no encontrada.');
    }

    return await EspecialidadModel.deleteById(id, conn);
  } catch (err) {
    throw err;
  }
}

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

Fig 168 Método deleteEspecialidad() del servicio de especialidades del back.

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

Fig 169 Respuesta de estado 200 si se ha eliminado correctamente la especialidad.

4. GENERACIÓN Y ENVÍO DE CORREOS ELECTRÓNICOS

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

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

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

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

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

Fig 170 Uso del servicio de mails en la creación de usuarios paciente.

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

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

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

Fig 171 Método encargado de mandar el mail de bienvenida.

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

Fig 172 Generación del transportador.

autenticarse con la cuenta y mandar el email correspondiente.

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

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

  return template(data);
}
```

Fig 173 Compilación de la plantilla handlebars.

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

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

Fig 174 Plantilla handlebars para el mail de bienvenida.

En tercer lugar, debemos crear los detalles del mail: remitente, destinatario, asunto, contenido y adjuntos (que en este caso será un array vacío, pero en otros casos como la confirmación de una cita médica será un PDF) para ello utilizamos el método **createMailDetails()** que devuelve un objeto con estos parámetros.

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

Fig 175 Generación del objeto con los detalles del mail.

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

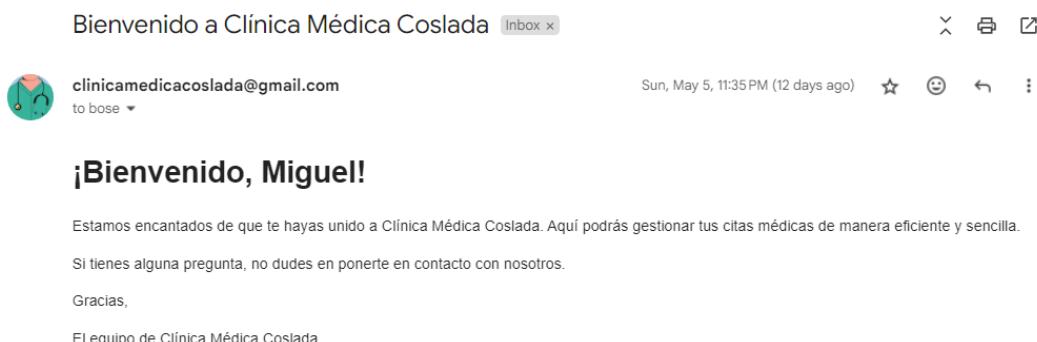


Fig 176 Mail de bienvenida.

5. GENERACIÓN Y DESCARGA DE DOCUMENTOS PDF

1) Solicitud de descarga en el cliente

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

Listado medicamentos						
Estas son tus medicaciones, Alberto						Descargar en PDF
	Nombre	Hora	Dosis	Inicio	Fin	Observaciones
Amoxicilina		07:00	1	03-05-2024	03-06-2024	Tomar antes del desayuno
		17:00	2	04-05-2024		Tomar durante la cena
		22:00	1	01-06-2024	01-06-2024	
		00:00	2	02-05-2024		
Atorvastatina		01:00	2	09-05-2024		
	Ibuprofeno	08:00	1	29-04-2024	29-05-2024	Tomar con el desayuno
		23:00	2	28-04-2024		
Loratadina		23:00	1	05-05-2024	05-06-2024	

Fig 177 Vista de la medicación del paciente.

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

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

2) Manejo por parte del servidor

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

Fig 178 Método de Angular que maneja la descarga.

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

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

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

Fig 179 Servicio de Node.JS para el manejo de las prescripciones del paciente.

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

3) Generación del PDF

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

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

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

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

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

Fig 180 Método del servicio de PDF encargado de generar el PDF de prescripciones.

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

Para ello se utiliza el método

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

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

  await PdfService.#generatePDF(bodyHtml, pdfPath);

  return pdfPath;
}
```

Fig 181 Generación del árbol de directorios.

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

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

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

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

    const headerHtml = compiledHeader({ context: {} });
    const footerHtml = compiledFooter({ context: {} });

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

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

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

    await browser.close();
}
```

Fig 182 Método encargado de generar el PDF utilizando puppeteer.

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

Fig 183 Fragmento de la plantilla de handlebars.

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

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

Fig 184 Respuesta de descarga desde el servidor.

4) Destrucción del PDF

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

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

Fig 185 Método encargado de la destrucción del fichero.

5) Descarga en el cliente

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

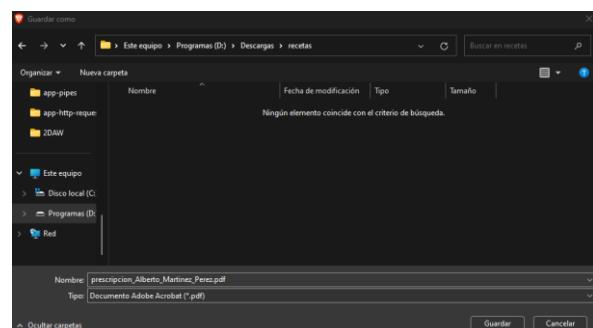


Fig 186 Ventana de descarga del PDF en el navegador.

Clinica Médica Coslada, SL

RECETA DE MEDICAMENTOS

Paciente: Alberto Martínez Pérez
Número de Historia Clínica: 2024042018551244

Medicamentos

Amoxicilina
Descripción: Antibiótico de amplio espectro

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

Atorvastatina
Descripción: Estatina para reducir el colesterol

Hora	Dosis	Fecha de inicio	Fecha de fin	Observaciones

Fig 187 Fragmento del PDF resultante.

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(`message: 'Error al generar el código QR.'`);
  }
};
```

Fig 188 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 189 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 190 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 191 Evento para la eliminación de registros en la tabla token.

B. Evento para eliminar las tomas vencidas

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

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

Fig 192 Evento para la eliminación de tomas vencidas

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

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

```
-- 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 193 Procedimiento para la eliminación de tomas vencidas. Generación del cursor.

En segundo lugar, tenemos toda la lógica de eliminación la cual se realiza a través de un LOOP de PL-SQL que cicla a través del cursor eliminando los registros en la tabla paciente_toma_medicamento y en toma con el id correspondiente al valor que tenga el cursor en la iteración.

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

```
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;
```

Fig 194 Procedimiento para la eliminación de tomas vencidas. Bucle de eliminación.

8. GENERACIÓN DE ARCHIVOS DE LOG

Para guardar un registro de las peticiones y respuestas del servidor se ha generado un sistema de generación de archivos de log utilizando las bibliotecas **morgan** y **rotating-file-stream**.

Morgan es una biblioteca que se encarga de forma automática de generar registros log cada vez que el servidor recibe una petición desde un cliente. Mientras que rotating-file-stream cumple con la funcionalidad de crear cada día un nuevo fichero .log que almacena las solicitudes de las siguientes 24h y que se guarda en el directorio logs.

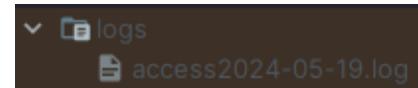


Fig 195 Directorio logs.

Para que esto sea posible hay que realizar unas configuraciones previas en el fichero app.js.

- Definir el directorio donde se guardarán los ficheros (en nuestro caso se almacenan en el directorio logs):

```
const accessLogStream = rfs.createStream(generateLogFileName, { options: {
    interval: '1d',
    path: path.join(__dirname, 'logs'),
}});
```

Fig 196 Configuración de rotatig-file-stream.

- Hacer consciente a app.js de que debe usar Morgan para el manejo de logs:

```
app.use(morgan('combined', { options: { stream: accessLogStream }}));
```

Fig 197 Configuración de morgan.

Con estas sencillas configuraciones, cada vez que pasen 24h se generará un nuevo fichero de log (para generar el nombre del fichero se utiliza la función **generateLogFileName()** que a partir de la fecha actual genera un nombre de fichero)

```
export const generateLogFileName = () => {
  const time = new Date();

  const year = time.getFullYear();
  const month = (time.getMonth() + 1).toString().padStart(2, '0');
  const day = time.getDate().toString().padStart(2, '0');

  return "access" + year + "-" + month + "-" + day + ".log";
}
```

Fig 198 Función para la generación del nombre del fichero de log.

en el cual se guardará información sobre las diferentes peticiones y respuestas que se den (IP de origen, navegador, código de estado, ruta de acceso, etc.).

```
access2024-05-19.log
1 ::1 - - [19/May/2024:08:31:47 +0000] "OPTIONS /api/cita?page=1&limit=15 HTTP/1.1" 204 0 "http://localhost:4200"
2 ::1 - - [19/May/2024:08:31:47 +0000] "GET /api/cita?page=1&limit=15 HTTP/1.1" 200 2344 "http://localhost:4200"
3 ::1 - - [19/May/2024:08:31:51 +0000] "OPTIONS /api/prescripcion HTTP/1.1" 204 0 "http://localhost:4200"
4 ::1 - - [19/May/2024:08:31:51 +0000] "GET /api/prescripcion HTTP/1.1" 200 2149 "http://localhost:4200"
5 ::1 - - [19/May/2024:08:32:26 +0000] "OPTIONS /api/cita?page=1&limit=10 HTTP/1.1" 204 0 "http://localhost:4200"
6 ::1 - - [19/May/2024:08:32:26 +0000] "GET /api/cita?page=1&limit=10 HTTP/1.1" 304 - "http://localhost:4200"
7 ::1 - - [19/May/2024:08:32:29 +0000] "OPTIONS /api/usuario HTTP/1.1" 204 0 "http://localhost:4200"
8 ::1 - - [19/May/2024:08:32:29 +0000] "OPTIONS /api/tipo-via HTTP/1.1" 204 0 "http://localhost:4200"
```

Fig 199 Ejemplo de fichero de log.

9. GENERACIÓN DE DOCUMENTACIÓN AUTOMÁTICA

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

A. Swagger / Swagger UI

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

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

1) Archivo de configuración

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

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

Fig 200 Configuración de Swagger.

Estas opciones básicas se componen de:

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

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

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

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

Fig 201 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 procesado 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 202 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 identificar **@swagger** en los ficheros de rutas (como alternativa se pueden definir objetos **path** en el fichero de configuración).

Estos comentarios se compondrán de información sobre el tipo de ruta, un resumen de lo que se espera de esta ruta, si requiere de autenticación y de los tipos de respuestas (códigos de estado) que se pueden producir con

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

Fig 203 Ejemplo de comentario @swagger.

su ejecución. Cada uno de estos códigos tendrá una descripción y el tipo de esquema que devolverá.

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`.

The screenshot shows the Swagger UI interface for the `GET /provincia` endpoint. At the top, it says "Obtiene las provincias". Under "Parameters", there are none. In the "Responses" section, there are two entries:

- Code: 200**: Description: Las provincias fueron obtenidas exitosamente. Media type: application/json. Example Value: A JSON object with properties `id` and `nombre`. Schema: A JSON object with properties `id` and `nombre`.
- Code: 404**: Description: Las provincias no fueron encontradas. Media type: application/json. Example Value: A JSON object with a single property `errors` containing a string value.

La potencia de Swagger UI es que *Fig 204 Swagger UI de la ruta GET /api/provincia.*

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**.

The screenshot shows the curl command being run in a terminal window:

```
curl -X 'GET' \
  'http://localhost:3000/api/provincia' \
  -H 'accept: application/json'
```

Below the command, the "Request URL" is shown as `http://localhost:3000/api/provincia`. The "Server response" section shows the API's response:

Code: 200

Response body:

```
[
  {
    "id": 2,
    "nombre": "Albacete"
  },
  {
    "id": 3,
    "nombre": "Alicante / Alacant"
  },
  {
    "id": 4,
    "nombre": "Almería"
  },
  {
    "id": 5,
    "nombre": "Araba / Álava"
  },
  {
    "id": 33,
    "nombre": "Asturias"
  },
  {
    "id": 6,
    "nombre": "Ávila"
  },
  {
    "id": 8,
    "nombre": "Badajoz"
  }
]
```

Response headers:

```
access-control-allow-origin: http://localhost:4200
content-length: 340
content-type: application/json; charset=utf-8
date: Fri, 17 May 2024 18:39:17 GMT
etag: W/"3d5-0e3b4e0001q95tp0K4/gbw"
vary: Origin
x-powered-by: Express
```

Fig 205 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.

The screenshot shows the schema definition for the `Provincia` entity:

```
Provincia ▼ [Provincia ▼ {
  id
  nombre
}]]
```

Under `id`, it says "string" and "El id de la provincia". Under `nombre`, it says "string" and "El nombre de la provincia".

Fig 206 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.
- En el caso de declarar espacios de nombres será necesario crear un fichero namespaces.js donde se detallen estos espacios de nombres.

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

Fig 207 Archivo de configuración de JSDoc.

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

```

Fig 208 Fichero namespaces.js.

- Para declarar un espacio de nombres en el fichero simplemente tendremos que crear un comentario JSDoc y añadir la etiqueta `@namespace` seguida del nombre del espacio de nombres.
- Por último, al igual que ocurría con Swagger, en el fichero app.js habrá que definir la ruta que se utilizará para acceder a esta documentación.

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

Fig 209 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), `@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.

```
/**
 * @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 210 Ejemplo de comentario JSDoc.

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 211 Script de ejecución de JSDoc.

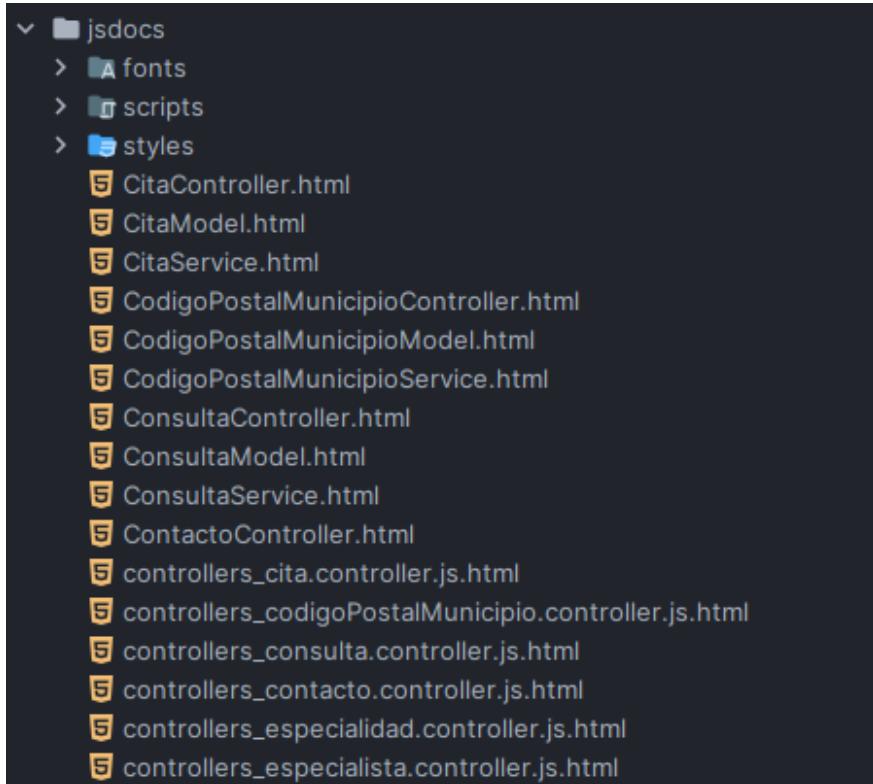


Fig 212 Directorio de JSDoc.

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 213 Visualización de documentación automática en web.

10. DESPLIEGUE DE LA APLICACIÓN WEB

Para el despliegue de la aplicación hemos decidido crear una red de máquinas virtuales que simulen un ecosistema de servidores que se compone de un servidor web que aloja el proyecto de Angular, un servidor de aplicaciones que aloja el proyecto de Node.JS-Express.JS y un servidor de base de datos que aloja el sistema gestor de base de datos MySQL.

A. Configuración común de los servidores

1) Especificaciones de las máquinas virtuales

Las tres máquinas virtuales utilizan una configuración idéntica:

- Sistema operativo: Ubuntu Server 24.04 LTS Noble Numbat.
- Memoria RAM: 3072 MB.
- Espacio en disco: 25 GB.
- Número de procesadores: 2.
- Tipo de red: Adaptador puente

Fig 214 Instalación de Ubuntu Server 24.04.

2) Configuración de netplan

Con el fin de mantener unas IPs fijas se realiza una configuración de netplan para ello se realiza un nuevo fichero de tipo YAML con el nombre 01-netcfg.yml que, entre otras cosas, almacena la versión de formato de configuración, las interfaces Ethernet, la dirección IP estática, si se permite o no DHCP para IPv4, etc.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: no
      addresses: [192.168.1.44/24]
      gateway4: 192.168.1.1
      nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
```

Fig 215 Configuración de Netplan para el servidor web.

Una vez realizado si utilizamos el comando sudo netplan apply se procesará el archivo YAML y se aplicará la configuración de red especificada.

3) Configuración del firewall

Además de Netplan es necesario configurar el firewall del servidor para mejorar la seguridad y gestión de la red.

```
serv-angular@servweb:~$ sudo ufw status
Status: active
To                         Action      From
--                         --         --
Nginx Full                 ALLOW      Anywhere
Nginx Full (v6)             ALLOW      Anywhere (v6)
```

Fig 216 Estado del firewall en el servidor web.

Para ello utilizamos UFW.

B. Despliegue del servidor web

1) Generación de la build de Angular

Aunque Angular incorpora un servidor para ejecutarse, no es recomendable utilizar este servidor en producción al ser un servidor poco seguro y más orientado hacia crear un entorno de desarrollo que uno de producción, por ello necesitaremos instalar Nginx en el sistema como veremos más adelante, pero antes de ello debemos generar una build de nuestro proyecto.

Esta build servirá para compilar y construir una aplicación de Angular convirtiendo los archivos en ficheros JavaScript, HTML y CSS que pueden ser ejecutados por un navegador web.

Para ello se utiliza el comando `ng build` seguido de la opción `--configuration production`. Este proceso provocará la creación de un directorio `/dist` que almacena el proyecto compilado

```
serv-angular@servweb:~/frontend$ ng build --configuration production
Initial chunk files | Names | Raw size | Estimated transfer size
main-SJPQ3R3K.js | main | 2.17 MB | 122.74 kB
styles-4PHELYJV.css | styles | 379.48 kB | 36.91 kB
chunk-CPEN6542.js | - | 170.29 kB | 49.93 kB
scripts-BXT2P7N3.js | scripts | 166.91 kB | 49.26 kB
polyfills-RT5I6R6G.js | polyfills | 83.10 kB | 10.72 kB
| Initial total | 2.90 MB | 269.56 kB
Lazy chunk files | Names | Raw size | Estimated transfer size
chunk-P3UJ2MTK.js | browser | 65.75 kB | 17.08 kB
Output location: /home/serv-angular/frontend/dist/frontend
```

Fig 217 Ejecución de `ng build`.

```
serv-angular@servweb:~/frontend$ ls -la dist/
total 12
drwxrwxr-x 3 serv-angular serv-angular 4096 may 21 17:11 .
drwxrwxr-x 7 serv-angular serv-angular 4096 may 21 20:01 ..
drwxrwxr-x 3 serv-angular serv-angular 4096 may 22 16:46 frontend/
serv-angular@servweb:~/frontend$ ls -la dist/frontend/
total 36
drwxrwxr-x 3 serv-angular serv-angular 4096 may 22 16:46 .
drwxrwxr-x 3 serv-angular serv-angular 4096 may 21 17:11 ..
-rw-rw-r-- 1 serv-angular serv-angular 22840 may 22 16:46 3rdpartylicenses.txt
drwxrwxr-x 4 serv-angular serv-angular 4096 may 22 16:46 browser/
serv-angular@servweb:~/frontend$ ls -la dist/frontend/browser/
total 3140
drwxrwxr-x 4 serv-angular serv-angular 4096 may 22 16:46 .
drwxrwxr-x 3 serv-angular serv-angular 4096 may 22 16:46 ..
drwxrwxr-x 3 serv-angular serv-angular 4096 may 22 16:46 assets/
-rw-rw-r-- 1 serv-angular serv-angular 174378 may 22 16:46 chunk-CPEN6542.js
-rw-rw-r-- 1 serv-angular serv-angular 67323 may 22 16:46 chunk-P3UJ2MTK.js
-rw-rw-r-- 1 serv-angular serv-angular 67646 may 22 16:46 favicon.ico
-rw-rw-r-- 1 serv-angular serv-angular 9854 may 22 16:46 index.html
-rw-rw-r-- 1 serv-angular serv-angular 2271998 may 22 16:46 main-SJPQ3R3K.js
drwxrwxr-x 2 serv-angular serv-angular 4096 may 22 16:46 media/
-rw-rw-r-- 1 serv-angular serv-angular 33898 may 22 16:46 polyfills-RT5I6R6G.js
-rw-rw-r-- 1 serv-angular serv-angular 170917 may 22 16:46 scripts-BXT2P7N3.js
-rw-rw-r-- 1 serv-angular serv-angular 388587 may 22 16:46 styles-4PHELYJV.css
```

Fig 218 Contenido del directorio `/dist`.

2) Instalación y configuración de Nginx

Una vez hecho lo anterior debemos instalar un servidor web, en nuestro caso nos hemos decantado por Nginx.

Completada la instalación (la cual se puede realizar con sudo apt install nginx) habrá que configurar el servidor, aunque antes deberemos hacer una

```
serv-angular@servweb:~/frontend$ ls -la /var/www/html/
total 40
drwxr-xr-x 3 root root 4096 may 21 19:30 .
drwxr-xr-x 3 root root 4096 may 21 17:09 ..
-rw-r--r-- 1 root root 22840 may 21 20:20 3rdpartylicenses.txt
drwxr-xr-x 4 root root 4096 may 21 20:02 browser
-rw-r--r-- 1 root root 615 may 21 17:09 index.nginx-debian.html
```

Fig 219 Contenido de /var/www/html/ tras la copia del contenido del proyecto.

copia del contenido del directorio /dist/nombre_proyecto en /var/www/html/ para que de esa forma el proyecto pueda ser utilizado en la configuración de nginx.

Una vez hecho esto, podemos configurar el fichero del servidor, en nuestro caso hemos decidido editar el fichero default para que apunte al directorio /browser que es el que contiene el proyecto.

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    root /var/www/html/browser;
    index index.html index.htm index.nginx-debian.html;
    server_name _;
    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Fig 220 Fichero de configuración de Nginx.

Tras reiniciar el servicio Nginx si accedemos al servidor desde el equipo anfitrión a través de su IP (en este caso 192.168.1.44) podremos acceder al sitio web.



Fig 221 Sitio web alojado en el servidor.

C. Despliegue del servidor de aplicaciones

Una vez realizada la configuración del servidor web se debe configurar el despliegue del servidor de aplicaciones.

Debido a la naturaleza de Node simplemente es necesario que el Ubuntu server contenga el directorio con el proyecto de

```
serv-nodejs@servnodejs:~/server$ npm run start
> server@1.0.0 start
> node app.js
NodeJS Server listening on http://192.168.1.24:3000
```

Fig 222 Lanzamiento del servidor de NodeJS.

Node y comprobar su funcionamiento lanzando el comando de ejecución del servidor npm run start.

Como en este caso usaremos el puerto 3000 para la escucha, como se explicó en la configuración básica deberemos abrir este puerto en el firewall para la escucha.

```
serv-nodejs@servnodejs:~/server$ sudo ufw status
Status: active
To                         Action      From
--                         --          --
3000                       ALLOW      Anywhere
3000 (v6)                  ALLOW      Anywhere (v6)
```

Fig 223 Situación del firewall en el servidor de aplicaciones.

Para comprobar el funcionamiento correcto podemos acceder a la IP por el puerto 3000 a una de las rutas abiertas del servidor, por ejemplo, las rutas de documentación como /api-docs.

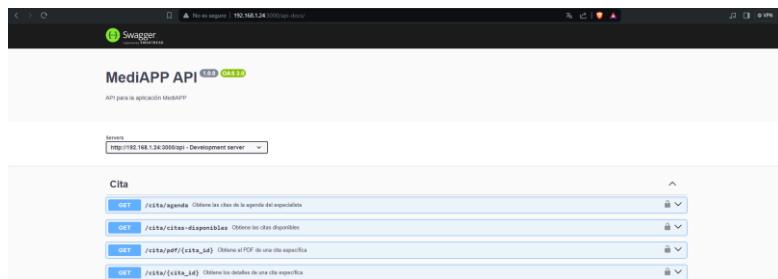


Fig 224 Acceso a la web de documentación de la API desde el anfitrión.

D. Despliegue del servidor de base de datos

1) Instalación y configuración de MySQL Server.

Por último, debemos desplegar el servidor que albergará el sistema gestor de base de datos, para ello debemos instalar MySQL server en la máquina con el comando sudo apt install mysql-server.

El siguiente paso será configurar el servidor para que pueda escuchar peticiones desde el exterior ya que por defecto escucha a localhost nada más, para ello debemos modificar la configuración del fichero /etc/mysql/mysql.conf.d/mysql.cnf y cambiar la dirección asociada a bind-address de 127.0.0.1 a 0.0.0.0, de esta manera podremos crear las tablas desde el equipo anfitrión usando la herramienta Workbench para facilitar la creación de tablas e inserción de datos de prueba. Posteriormente, en el siguiente apartado, se modificará este parámetro para que sólo reciba peticiones desde el servidor de NodeJS.

```
bind-address = 0.0.0.0
```

Fig 225 Parámetro bind-adress.

2) Generación del usuario que se conectará a la base de datos

Tras reiniciar el servidor para que estos cambios sean tenidos en cuenta, habrá que conectarse como root a MySQL (`sudo mysql -u root`) y crear un usuario que se pueda conectar desde fuera de la app.

```
mysql> CREATE USER 'clinica_admin'@'%' IDENTIFIED BY 'yHD63d9jnYfn';
Query OK, 0 rows affected (0,01 sec)
```

Fig 226 Generación del usuario de la base de datos.

Crear la base de datos:

```
CREATE DATABASE 'clinica';
```

Fig 227 Generación de la base de datos.

Y concederle permisos al usuario anterior sobre esta base de datos:

```
mysql> GRANT ALL ON clinica.* TO 'clinica_admin'@'%';
Query OK, 0 rows affected (0,02 sec)
```

Fig 228 Privilegios al usuario 'clinica_user'.

3) Configuración del firewall

Al igual que en los casos anteriores debemos habilitar el firewall y permitir las conexiones por el puerto 3306, el puerto por defecto para MySQL.

```
serv-mysql@servmysql:~$ sudo ufw status
Status: active
To                         Action      From
--                         --          --
3306                       ALLOW       Anywhere
3306 (v6)                  ALLOW       Anywhere (v6)
```

Fig 229 Situación del firewall en el servidor de base de datos.

4) Conexión a la base de datos desde el anfitrión

Para conectarnos a la base de datos podemos usar MySQL Workbench y crear una conexión.

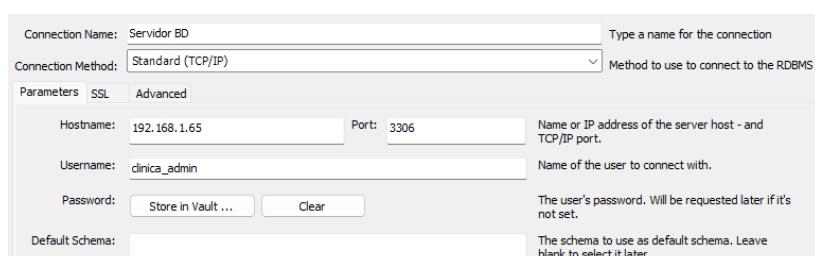


Fig 230 Creación de conexión en MySQL Workbench.

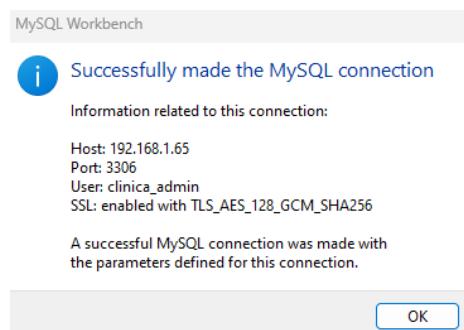


Fig 231 Conexión exitosa a la base de datos desde el anfitrión.

Tras la creación de tablas e inserción de datos de prueba podemos comprobar que en el servidor de base de datos estos son accesibles.

```
mysql> use clinica
Reading table information for co
You can turn off this feature to
Database changed
mysql> show tables;
+-----+
| Tables_in_clinica |
+-----+
| cita
| codigo_postal
| codigo_postal_municipio
| consulta
| error_log
| especialidad
| especialista
| glucometria
| informe
| informe_patologia
| medicamento
| municipio
| paciente
| paciente_toma_medicamento
| patologia
| provincia
| rol
| tension_arterial
| tipo_via
| token
| toma
| usuario
+-----+
22 rows in set (0,00 sec)
```

Fig 232 Búsqueda de las tablas de la base de datos 'clinica' en el servidor.

E. Demostración de funcionamiento

Una vez que tenemos los servidores funcionando de forma independiente y comunicándose con el anfitrión el último paso es conectarlos para que se vean entre sí.

1) Cambios en el servidor web

En el servidor web debemos modificar el fichero de variables de entorno para que cuando se hagan solicitudes a la API no se hagan a localhost como en el entorno de desarrollo sino a la IP de nuestro servidor de aplicaciones. De esta forma se harán llamadas a 192.168.1.24 por el puerto 3000 (tal y como se explicó en la configuración básica).

```
export const environment: { apiUrl: string } = {
  apiUrl: 'http://192.168.1.24:3000/api'
```

Fig 233 Modificación del fichero de entorno de Angular.

entorno para que cuando se hagan solicitudes a la API no se hagan a localhost como en el entorno de desarrollo sino a la IP de nuestro servidor de aplicaciones. De esta forma se harán llamadas a 192.168.1.24 por el puerto 3000 (tal y como se explicó en la configuración básica).

para asegurarse de que este valor nunca cambia habrá que hacer un netplan en el servidor de aplicaciones).

Para que los cambios que hemos generado en el fichero de variables de entorno surta efecto deberemos de repetir la generación de la build y copiar el directorio resultante en /var/www/html/ tal y como se detalló en el despliegue del servidor web.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: no
      addresses: [192.168.1.24/24]
      gateway4: 192.168.1.1
      nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
```

Fig 234 Netplan del servidor de aplicaciones.

2) Cambios en el servidor de aplicaciones

En el servidor de NodeJS debemos modificar el fichero .env para que escuche cualquier petición que le llegue (las CORS se encargarán de evitar accesos no autorizados, para ello habrá que modificar el ORIGIN a 192.168.1.44).

Además, se debe añadir la IP de nuestro servidor de base de datos que ahora se encuentra en 192.168.1.65. Al igual que en los otros dos casos, el conseguir esta IP estática se consigue gracias a netplan.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: no
      addresses: [192.168.1.65/24]
      gateway4: 192.168.1.1
      nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
```

Fig 236 Netplan del servidor de base de datos.

```
# Configuración de variables de entorno
# Configuración del servidor
SERV_HOST=0.0.0.0
SERV_PORT=3000
SERV_IP=192.168.1.24
SERV_API_URL=http://192.168.1.24:3000/api

# Configuración de la base de datos
DB_HOST=192.168.1.65
DB_PORT=3306
DB_USER=clinica_user
DB_PASS=uHD63d9jnYfn
DB_NAME=clinica
DB_TIMEZONE=2

# Configuración de JWT
JWT_SECRET_KEY='S3cr3tP4ssw0rd'
JWT_RESET_SECRET_KEY='R3s3tP4ssw0rd'
JWT_REFRESH_SECRET_KEY='R3fr3shP4ssw0rd'

# Configuración de CORS
CORS_ORIGIN=http://192.168.1.44
CORS_METHODS="GET,POST,PUT,DELETE"
CORS_ALLOWED_HEADERS="Content-Type,Authorization"

# Configuración de Angular
ANGULAR_HOST=http://192.168.1.44
ANGULAR_PORT=80

# Configuración de email
EMAIL_ACCOUNT=clinicamedicacoslada@gmail.com
EMAIL_PASS="cdztrusyberspqhj"
```

Fig 235 Fichero .env de NodeJS.

3) Cambios en el servidor de base de datos

En el servidor de base de datos debemos permitir únicamente las conexiones a la base de datos a través de 192.168.1.44. Para ello utilizaremos ufw.

```
serv-mysql@servmysql:~$ sudo ufw allow from 192.168.1.24 to any port 3306
```

Fig 237 Configuración de ufw para escuchar sólo al servidor de aplicaciones.

4) Acceso desde el anfitrión

Si accedemos desde el equipo anfitrión a por ejemplo <http://192.168.1.44/listado-de-especialidades> conseguiremos el listado de especialidades desde la base de datos.



Fig 238 Web de especialidades en el anfitrión.

Así mismo si nos logamos, accederemos a nuestros datos:

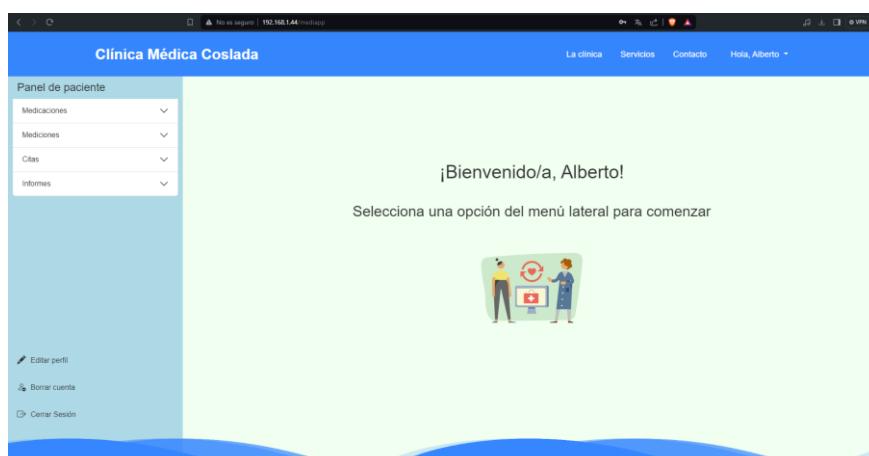


Fig 239 Web de paciente en el anfitrión.

CONCLUSIONES

En conclusión, gracias al conocimiento que hemos conseguido desarrollar durante estas semanas podemos decir que hemos cumplido con los objetivos que nos marcamos antes de iniciar el diseño de la aplicación, permitiendo que los tres tipos de usuarios puedan realizar las funciones que nos marcamos como principales, consiguiendo además una interfaz funcional, accesible e intuitiva siendo visible en dispositivos de pantalla pequeña, mediana y grande.

Obviamente al ser un desarrollo de pocas semanas el potencial de la aplicación, así como su margen de mejora es enorme. Por ejemplo, se podrían añadir nuevas funcionalidades como un chat en vivo con tu médico, la posibilidad de realizar video-consultas, un sistema de solicitud de pruebas médicas como pruebas radiológicas, analíticas, etc.

Además, se podrían añadir nuevos perfiles de profesiones sanitarias como enfermeros, fisioterapeutas, técnicos de laboratorio... lo que posibilitaría a su vez crear nuevas funcionalidades como crear tablas de ejercicios personalizados para cada paciente, generación de informes de resultados de analíticas, etc.

También debemos marcarnos como objetivo a futuro la migración del proyecto a la versión 18 de Angular para mantener el proyecto lo más actualizado posible con respecto a las tecnologías. Esta versión fue publicada el pasado mes de mayo pero nos ha sido imposible implementar debido a que algunas de las librerías que utilizamos no han actualizado sus códigos fuente para ser compatibles con la nueva versión del *framework*; así mismo el próximo mes de octubre está previsto el lanzamiento de la versión 22 en formato *active* de Node.JS, algo que también tendríamos que valorar de cara a implementar, durante el desarrollo del proyecto esta versión salió en su formato *current* pero decidimos no utilizarlo ya que se desaconseja su uso en proyectos de producción.

Por último, el código actual podría ser mejorado en cuanto a optimización, rendimiento y apariencia con un mayor conocimiento de las tecnologías utilizadas que han sido explicadas a lo largo de este documento. También sería conveniente una mejora de seguridad a través de llamadas HTTPS.

WEBGRAFÍA

Angular. (s. f.). <https://angular.io/docs>

Angular. (s. f.-b). Angular. <https://angular.dev/>

Bootstrap. (s. f.). <https://getbootstrap.com/>

Express - Node.js web application framework. (s. f.). <https://expressjs.com/>

Handlebars. (s. f.). <https://handlebarsjs.com/>

Node.js v20.13.1 Documentation. (s. f.). <https://nodejs.org/docs/latest-v20.x/api/index.html>

npm Docs. (s. f.). <https://docs.npmjs.com/>

Postman documentation overview | Postman Learning Center. (2023, 19 octubre). Postman Learning Center. <https://learning.postman.com/docs/introduction/overview/>

Reinman, A. (s. f.). Nodemailer :: Nodemailer. <https://www.nodemailer.com/>

Swagger. (s. f.). <https://swagger.io/>

Use JSDOC: Index. (s. f.). <https://jsdoc.app/>