Facultad
de
Ciencias

APLICACIÓN WEB SOBRE JAVA PARA LA GESTIÓN DE CITAS DE ATENCIÓN SANITARIA

(Java-based web application for the management of healthcare appointments)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: José Antonio Mingo Ortega

Director: Julio Luis Medina Pasaje

Julio – 2022

Agradecimientos

Me gustaría expresar mi total agradecimiento, en primer lugar, a mi familia, amigos y compañeros por todo el apoyo transmitido a lo largo de mi carrera.
En segundo lugar, a mi tutor D.Julio Luis Medina Pasaje por el trabajo realizado durante el desarrollo del proyecto.
Y, por último, mostrar mi agradecimiento tanto a la Universidad de Cantabria como a todos los profesores que me han aportado los conocimientos necesarios para poder llegar a este punto.

Resumen

Palabras clave: Back-end, Java, Spring Boot, MySql, React

La meta de este trabajo es desarrollar un entorno web para la reserva de citas médicas que le proporcione al usuario tanto seguridad en cuanto a la integridad de su información como un manejo sencillo de la web.

El desarrollo sigue un patrón habitual de tres capas:

La capa de persistencia se apoya en una base de datos relacional tradicional organizada de modo que permita a la aplicación acceder a la información de manera no redundante, lo que ayudará a maximizar la eficacia de los accesos a esta misma.
La capa de negocio se ofrece como un servicio web que se centrará en el uso seguro y rápido de la información, considerando la seguridad un requisito indispensable para el correcto funcionamiento de la aplicación.
La capa de presentación es un cliente web orientado a que el usuario pueda navegar de manera sencilla por la aplicación Esto permite que los usuarios utilicen la aplicación independientemente de su nivel de conocimiento y fluidez en este tipo de vistas.

En su realización se utilizan herramientas como MySQL para la implementación de la base de datos, Java para la implementación del servidor (back-end), y JavaScript, HTML y CSS para la vista de cliente (front-end).

Abstract

Keywords: Back-end, Java, Spring Boot, MySql, React

The goal of this work is to develop a web environment for the booking of healthcare appointments that brings such functionality to the user with both security over the integrity of the information and an ease usage of the web.

The development follows the traditional three layers pattern:

The persistence layer works over a relational database that allows the user to access information in a non-redundant way, which will help to maximize the efficiency of access to the information.
The business layer is offered as a web service that focuses on the secure and fast use of information, considering security an indispensable requirement for the correct operation of the application.
The presentation layer is a web client designed so that the user can easily navigate through the application. This allows users to use the application regardless of their level of knowledge and fluency in this type of view.

Tools such as MySQL for the implementation of the database, Java for the implementation of the server (back-end), and JavaScript, HTML and CSS for the client view (front-end) are used in its realization.

Índice

1. Introducción

1.1. Contexto

En los tiempos que corren, después de una pandemia, ha quedado demostrado que las aplicaciones informáticas han ganado, aún más, un papel indispensable en nuestras vidas.

Ya no solo se cuenta con las aplicaciones web a modo de ocio o trabajo privado, sino también para agilizar los servicios. Cada día más instituciones ofrecen servicios como el que propone esta aplicación y cuentan con grandes desarrolladores para sus plataformas, porque como es evidente, brindar un servicio eficiente y seguro a los ciudadanos hace que estos obtengan un nivel de confianza mayor hacía dichas instituciones. Particularmente, después de que el mundo sea consciente de los grandes problemas que existen con las filtraciones de datos personales.

Cuando se habla de salud no se debe escatimar lo mínimo y se tiene que proporcionar tanto al trabajador sanitario como al usuario externo un servicio digno que permita, en este caso, un empleo eficiente de la aplicación, de manera que se pueda centrar en los pacientes que acuden a él.

Por otra parte, somos conocedores de que mucha gente sigue sin utilizar estos medios informáticos para la gestión de los servicios ya sea debido al mal funcionamiento de estos o por su dificultad de uso, prefiriendo optar por medios más tradicionales como puede ser el teléfono. Esto último da que pensar a las personas que gestionan instituciones y cada vez más buscan actualizar la faceta informática en búsqueda de llegar con facilidad a la mayor parte de los usuarios que utilicen sus servicios.

1.2. Objetivo

El objetivo principal de este trabajo es el desarrollo de una aplicación web fullstack para la gestión de citas de atención sanitaria que permita a los usuarios externos poder realizar su propia gestión de las citas para acudir a su médico además de poder ver las recomendaciones y recetas hechas en citas anteriores. En este caso el cliente hipotético con el que se trabajará para el desarrollo de esta aplicación se considerará una institución que basa su trabajo principal en la atención sanitaria.

Para el desarrollo de este trabajo se han identificado dos tipos principales de usuarios mientras que a la vez este último está dividido en otro dos. Estos tipos de usuarios se explican en los siguientes apartados.

1.2.1. Usuario externo

El usuario que no pertenece a la organización (paciente) solo tendrá acceso a su información. Podrá consultar sus datos personales y algunos datos relacionados con la organización, como los centros de salud que se encuentran disponibles para acudir de emergencia en el horario que no corresponde al habitual de las citas. Por otra parte, el usuario además de poder controlar sus datos podrá realizar citas con su médico asignado y podrá ver sus citas anteriores y las indicaciones que se realizaron en las mismas.

1.2.2. Usuario interno: médico

El trabajador sanitario (médico) podrá ver una lista con las citas que tiene además de poder ver los datos de las personas que las concretaron, pudiendo acceder a sus correspondientes historiales médicos y modificando las citas para añadir datos como recomendaciones. Este usuario no tendrá permisos de acceso para ver el calendario de citas de otros usuarios de su mismo nivel.

1.2.3. Usuario interno: administrador

.

El rol de administrador podrá añadir usuarios, modificar usuarios y realizar citas para aquellas personas que decidan contactar vía telefónica en vez de utilizar la propia aplicación. Por otro lado, podrá actualizar ciertos datos como los centros que se encuentran abiertos para los servicios de urgencia.

1.3. Organización de la memoria

Las secciones que componen el resto de esta memoria se organizan como se enumera a continuación.

2. Tecnologías, herramientas y lenguajes utilizados: se describen las tecnologías utilizadas y el uso que han tenido en el proyecto.

3. Metodología y planificación: muestra el itinerario seguido dentro de las correspondientes fases del proyecto.

4. Especificación de requisitos y casos de uso: formaliza, tras la toma de necesidades, los requisitos tanto funcionales como no funcionales que se han extraído y los casos de uso que satisfarán los requisitos.

5. Diseño del sistema: muestra los lineamientos y modelos de diseño que se han definido para el correcto desarrollo de la aplicación divido en las fases que se han realizado.

6. Implementación: describe la codificación tanto del backend como del frontend en las correspondientes fases del proceso de desarrollo.

7. Pruebas y resultados: muestra el proceso llevado para la realización de pruebas para validar el correcto funcionamiento de la aplicación.

8. Conclusiones y trabajos futuros: se apuntan conclusiones tras el desarrollo del proyecto y algunas ideas que podrían mejorar el sistema en el futuro.

2. Tecnologías, herramientas y lenguajes utilizados

2.1. Tecnologías y herramientas

En este punto se proporciona una breve descripción de las tecnologías y herramientas que se han utilizado para el desarrollo de la aplicación.

- **MySQL Workbench.** MySQL Workbench es una herramienta visual de diseño de bases de datos que integra desarrollo de software, administración de bases de datos, diseño de bases de datos, gestión y mantenimiento para el gestor de base de datos MySQL [1]. En el proyecto se utiliza para realizar la implementación de la base de datos relacional que soporta la capa de persistencia.

- **Eclipse IDE.** Eclipse IDE es un entorno de desarrollo integrado de código abierto que facilita el desarrollo de aplicaciones basadas en Java [2]. En el proyecto se usa como entorno de desarrollo para implementar el back-end.

- **Spring Framework.** Spring es un framework para el desarrollo de aplicaciones y contenedor de inversión de control, de código abierto para la plataforma Java [3]. En este caso lo utilizamos para el desarrollo de la API REST.

- **Spring Boot.** Spring Boot permite compilar nuestras aplicaciones Web como un archivo .jar que podemos ejecutar como una aplicación Java normal consiguiéndolo gracias a que integra el servidor de aplicaciones en el propio .jar [4]. En el proyecto se utiliza para el despliegue de la aplicación.

- **Hibernate.** Hibernate es una herramienta de mapeo objeto-relacional (ORM) que facilita el mapeo de atributos en una base de datos tradicional, y el modelo de objetos de una aplicación mediante archivos declarativos o anotaciones en los beans (componentes Java) de las entidades que permiten establecer estas relaciones [5]. Se utiliza para mapear las clases y poder hacer la relación con la base de datos.

- **Maven.** Maven se utiliza en la gestión y construcción de software. Facilita tareas de procesamiento del código y configuración claramente definidas, como la compilación del código y su empaquetado. Además, gestiona dependencias entre librerías utilizadas e incluidas dentro de la estructura del JAR [6].

- **Swagger API.** Swagger es una especificación abierta para definir API REST. Un documento Swagger es el equivalente de API REST de un documento WSDL para un servicio web basado en SOAP. El documento Swagger especifica la lista de recursos disponibles en la API REST y las operaciones a las que se puede llamar en estos recursos [7]. Se emplea en la definición de la API de este proyecto.

- **Visual Studio Code.** Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y Web. Incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código [8]. En el proyecto se utiliza como entorno de desarrollo para el front-end.

- **React.** React ayuda a crear interfaces de usuario interactivas de forma sencilla. Diseña vistas simples para cada estado de la aplicación. Se encarga de actualizar y renderizar de manera eficiente los componentes correctos cuando los datos cambien [9].

- **Node.js.** Es Ideado como un entorno de ejecución de JavaScript orientado a eventos asíncronos, Node.js está diseñado para crear aplicaciones web escalables [10].

- **Bootstrap.** Es una biblioteca multiplataforma y un conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web [11].

- **Postman.** Postman es una aplicación que nos permite realizar pruebas de una API. Es un cliente HTTP que nos da la posibilidad de testear 'HTTP requests' a través de una interfaz gráfica de usuario, por medio de la cual obtendremos diferentes tipos de respuestas para su validación [12].

- **GitHub.** GitHub es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git [13].

2.2. Lenguajes

En este apartado se mencionan los lenguajes que se han utilizado para el desarrollo de la aplicación.

- **SQL.** SQL es un lenguaje de programación específico para actualizar, consultar y obtenerinformación de bases de datos relacionales.

- **Java.** Java es un sistema de programación muy usado para diversos fines. Un lenguaje de programación sencillo, multiplataforma y orientado a objetos.

- **JavaScript.** JavaScript es un lenguaje de scripts dinámico que admite la construcción de objetos basada en prototipos. Intencionalmente, la sintaxis básica es similar a Java y C++ para reducir la cantidad de conceptos nuevos necesarios para aprender el lenguaje [14].

- **HTML.** HTML es un lenguaje de marcado que permite indicar la estructura y de un documento web mediante etiquetas.

- **CSS.** CSS es un lenguaje que define la apariencia de un documento escrito en un lenguaje de marcado (usualmente, HTML). Así, a los elementos de la página web creados con HTML se les prefija una apariencia utilizando CSS [15]**.**

2.3. Definiciones

- Back-end. Parte de la aplicación que procesa el tratamiento de datos y realiza la conexión entre la base datos y el front-end.

- Front-end. Parte de la aplicación correspondiente con la intezfaz que facilitará el uso de esta al usuario desde un navegador web.

- API REST. Definición fundamental de los servicios ofrecidos por el back-end al front-end. Constituye el sustrato medular de la aplicación gracias a un amplio conjunto de funciones accesibles a través del protocolo HTTP.

- JWT (JSON Web Token). Es un estándar que permite la creación de tokens garantizando así que el uso de los servicios sea seguro [16].

- HTTP (Hypertext Transfer Protocol). Protocolo que permite hacer las llamadas para la transmisión de datos entre front-end y back-end.

2.4. Elección de tecnologías y lenguajes

Se han elegido las herramientas mencionadas previamente por las siguientes razones:

- Los entornos de desarrollo han sido seleccionados por la familiarización que tenía el desarrollador con ellos previo al comienzo del proyecto. Son lenguajes que permiten al usuario un desarrollo rápido gracias a la gran cantidad de ayudas y herramientas internas que proporcionan. MySQL

cuenta con una interfaz muy intuitiva que ayuda tanto a desarrollar la base de datos como a generar modelos de una manera sencilla.

- Los frameworks utilizados se han elegido por encima de otros por la gran compatibilidad que tenían con el tipo de desarrollo que se pedía implementar además de que en la gran mayoría de los casos son herramientas que el desarrollador había visto o utilizado previamente. Tanto los frameworks de Spring como Hibernate han hecho el desarrollo de la API REST y de la base de datos más liviano gracias a las facilidades que nos brindan con sus librerías prediseñadas.

- Maven ha sido seleccionado debido a que tiene una gran integración con el entorno de desarrollo con el que se ha realizado el back-end facilitando así la tarea del desarrollador. Al igual que en los anteriores casos, se estaba previamente familiarizado con la compilación y ejecución del código, así como con la gestión que hace de las dependencias.

- React y Node.JS son tecnologías con las que no se estaba familiarizado en gran medida, pero tras una pequeña investigación se decidió que gracias a las oportunidades que brindan con su facilidad de uso eran las dos mejores herramientas con las que desarrollar el código de la parte front-end frente a otras como podría ser Angular. De esta misma manera se eligió la librería de Bootstrap para añadir los componentes frente a PrimeReact [20], aunque en este caso ambas podrían ser totalmente válidas el desarrollador decidió la utilización de la primera.

- Para realizar pruebas sobre la API REST se optó por Postman debido a la facilidad de uso de su interfaz gráfica para hacer llamadas HTTP además de que se estaba familiarizado con la herramienta.

- Para el control de versiones se decidió utilizar GitHub por la familiaridad que se tiene con la herramienta además de la gran integración que tiene con los entornos de desarrollo, permitiendo consolidar todos los cambios directamente a través de la IDE o desde unla terminal.

- Por último, los lenguajes utilizados se seleccionaron realizando una valoración entre lo que mejor se adecuaba a la aplicación en cuestión y el conocimiento del desarrollador sobre ellos. Java y SQL son lenguajes muy utilizados a nivel de base de datos y front-end y, a su vez, encajaban a la perfección para la realización de la aplicación. Por otra parte, aunque los lenguajes JavaScript, HTML y CSS no eran tan conocidos por el desarrollador son punteros a nivel mundial a la hora de la programación de front-end lo cual hizo que se tomase la decisión de usarlos.

3. Metodología y planificación

En esta secciónse muestra la metodología que se seguido y la planificación empleada para el desarrollo de la aplicación.

3.1. Metodología

Como metodología de desarrollo se ha decidido utilizar una metodología incremental lo cual permite subdividir la evolución del proyecto en fases, de modo que se obtiene una versión actualizada de la aplicación en cada una de estas fases. Se optó por esta metodología pues conlleva que el contacto con el cliente se realizará mucho más frecuentemente (fase a fase) y se adecua mejor para desarrollos individuales.

En cada una de estas fases se realizará un análisis, un diseño, un desarrollo y un testeo por parte del stakeholder. Gracias al uso de esta metodología se proporciona al cliente la capacidad de ver fase a fase el desarrollo y avance de la aplicación haciendo más sencilla la captación de nuevos requisitos y funcionalidades necesarias que podrían no haber sido detectadas en fases previas del proyecto.

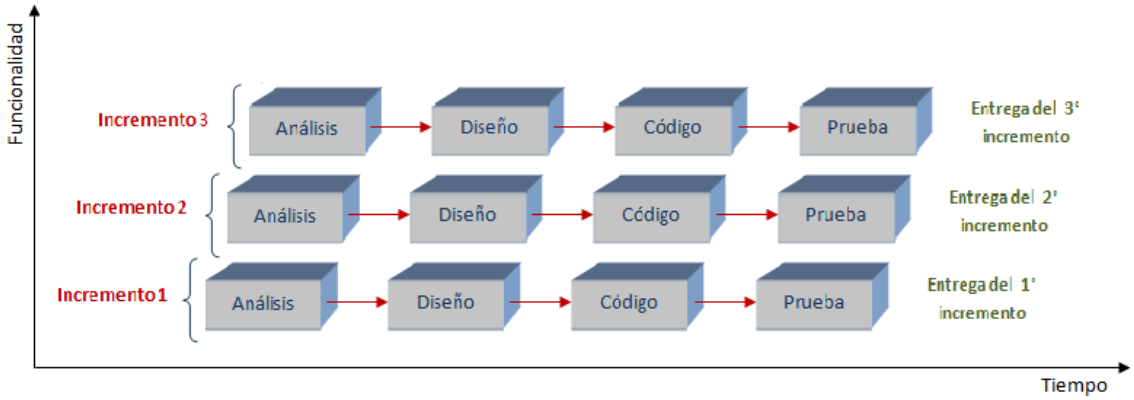

Imagen 1: Modelo de la metodología incremental.

3.2. Planificación

Para el desarrollo de este proyecto se ha decidido dividir la planificación en varias etapas que se explicarán con mayor detalle en los siguientes apartados. Estas etapas son la toma de requisitos, el diseño principal del sistema, la correspondiente investigación y formación sobre las herramientas a utilizar en la aplicación y el desarrollo de esta.

3.2.1. Validación de requisitos

Esta es primera de las etapas de la planificación, se centra en las reuniones con el stakeholder y la investigación de otras aplicaciones de carácter similar con el fin de hacer una captura exhausta de los requisitos tanto funcionales como no funcionales que el sistema debe cumplir para obtener el funcionamiento esperado.

3.2.2. Investigación y formación

Previo al desarrollo de la aplicación se ha requerido un tiempo en una investigación de las tecnologías y lenguajes a utilizar, evaluando cuales podrían ser aquellas que se adecúan mejor a los requisitos del sistema que se detectaron en el punto anterior. Una vez seleccionadas dichos lenguajes se procederá a la formación en ellos, ya fuese con la intención de reforzar ciertos conocimientos sobre aquellos que se dominan como Java o aprender nuevos aspectos y mejorar las bases sobre el lenguaje como es el caso de React.

3.2.3. Desarrollo

Como se mencionó anteriormente el desarrollo sigue una metodología incremental, en este caso divida en 2 fases. Para el desarrollo se crearán dos repositorios en GitHub, uno dedicado a la implementación del back-end y otro para el front-end donde se ha seguido la siguiente misma idea. En la rama master se mantendrá el desarrollo funcional de la aplicación. En la rama develop se mantendrá una copia de master que será utilizada para realizar las pruebas de integración una vez se ha implementado una nueva funcionalidad con el fin de no llevar a la rama master un código que no esté completamente probado y sea integrable con el resto de la aplicación. Por último, cada vez que se quiere trabajar en una funcionalidad nueva se crea una rama para su desarrollo, una vez el desarrollo está completo se hace el merge de dicha rama con develop para comprobar su correspondiente integración con el resto de la aplicación. Por último, cuando se corrobora al completo que la nueva funcionalidad es correcta se realiza el paso a master.

A continuación, se muestra cómo se han estructurado y qué se ha llevado a cabo en cada fase:

- **Fase 1**: Esta fase corresponde a la implementación de la base de datos y los métodos CRUD sencillos para el desarrollo de la API REST. Para la base de datos SQL se realiza un esquema conceptual de las tablas y relaciones para posteriormente crearla a través de MySQL Workbench. Por otra parte, una vez contemos con dicha base de datos se procederá a crear el back-end con Java. Con la ayuda de ciertas tecnologías como Spring y H2 se conectará dicho back-end con la base de datos y se creará el modelo CRUD de la API REST y su correspondiente swagger. Esta API REST permitirá en

próximas fases que el front-end tenga la posibilidad de conectarse a la base de datos con llamadas HTTP. En este punto no existirá ningún front-end con el que probar el back-end por lo que se utilizará la herramienta Postman para probar la correcta funcionalidad de lo desarrollado.

- **Fase 2**: Esta fase corresponde al aumento de la funcionalidad del back-end y el desarrollo del front-end con React. En primer lugar, se implementar JWT de Spring Security para que las llamadas que posteriormente se realicen desde el front-end sean seguras. Acto seguido se procede a crear la primera versión de la interfaz de usuario probando su correcta integración con el back-end y las llamadas CRUD que se permitan hacer. Por último, en esta fase se implementan aquellas partes del front-end y servicios en el back-end para el resto de las funcionalidades que han sido solicitadas.

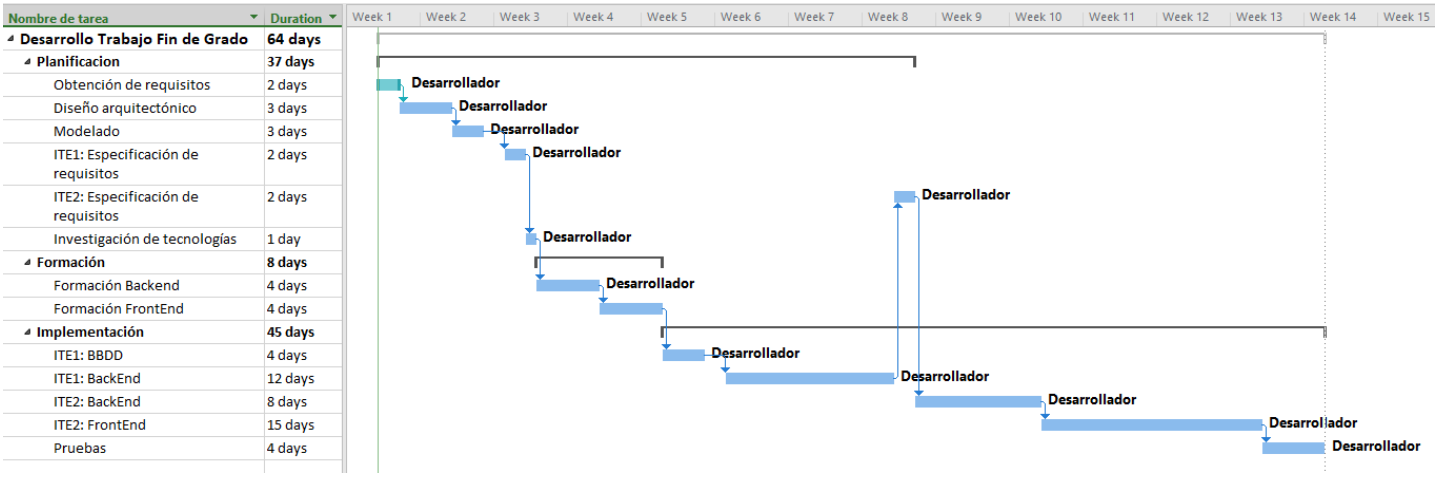

Imagen 2: Diagrama de Gantt correspondiente al desarrollo del proyecto

4. Especificación de requisitos y casos de uso

La especificación de requisitos y el análisis son el primer paso a seguir para realizar un desarrollo de este carácter. Para ello debemos distinguir entre dos tipos de requisitos: funcionales y no funcionales.

Para este proyecto el propio desarrollador será quien realice la correspondiente toma de requisitos para el sistema. Estos requisitos se obtienen de dos vías distintas. En primer lugar, se investigan aplicaciones web similares con la intención de encontrar requisitos básicos que debe contener la aplicación y, en segundo lugar, esto se ha amplia a través de reuniones con el stakeholder con el fin de detallar lo máximo posible los requisitos que debe cumplir esta aplicación respecto a las ideas del cliente.

Posteriormente, se describen las diferentes funcionalidades que tendrá el sistema según el rol del usuario que esté accediendo. Se muestra mediante diagramas de casos de uso el uso que cada usuario podrá dar a la aplicación y se analizarán algunos de ellos.

4.1. Actores del sistema

Se conoce como actores del sistema a los distintos roles o tipos de usuarios que usarán el sistema. Para este desarrollo se identificaron tres roles distintos entre los usuarios según las funciones que llevarán a cabo con el sistema.

Actor	Descripción
Usuario interno: Administrador	Tiene permiso para acceder a toda la información del sistema exceptuando los datos sensibles del usuario.
Usuario interno: Medico	Tiene permiso para ver sus datos y los datos médicos de los usuarios a los que es asignado, a su vez tiene permiso para ver las consultas que tendrá que realizar y modificarlas parcialmente.
Usuario externo	Tiene accesos para ver sus datos y actualizarlos, ver la información tanto de su médico asignado como de su centro de salud asignado y la posibilidad de creación de citas.

Tabla 1: Actores del sistema

4.2. Requisitos funcionales

Los requisitos funcionales son los que definen las funcionalidades que va a integrar el sistema.

ID	Descripción
RF01	El usuario deberá poder acceder a sus datos
RF02	El usuario deberá poder actualizar sus datos.
RF03	El usuario deberá poder realizar una cita telefónica.
RF04	El usuario deberá poder realizar una cita presencial.
RF05	El usuario deberá poder hacer anotaciones cuando realiza una cita.
RF06	El usuario deberá poder ver sus citas pasadas.
RF07	El usuario deberá poder ver sus próximas citas.
RF08	El usuario deberá poder ver los datos de su médico asignado.
RF09	El usuario deberá poder ver los datos de su centro de salud asignado.
RF10	El usuario deberá poder iniciar sesión.
RF11	El usuario deberá poder ver los centros de salud de urgencia.
RF12	El usuario deberá poder ver la disponibilidad de su médico.
RF13	El médico deberá poder ver sus citas.
RF14	El médico deberá poder añadir anotaciones en sus citas.
RF15	El médico podrá ver la información de sus usuarios asignados.
RF16	El médico deberá poder filtras sus citas.
RF17	El médico deberá poder iniciar sesión.
RF18	El administrador deberá poder dar de alta un usuario.
RF19	El administrador deberá poder filtrar por usuario.
RF20	El administrador deberá poder ver los datos no sensibles de los usuarios.
RF21	El administrador deberá poder ver los datos no sensibles de los médicos.
RF22	El administrador deberá poder realizar una cita.
RF23	El administrador deberá poder filtras las citas.
RF24	El administrador deberá poder ver las citas sin información sensible.
RF25	El administrador deberá poder ver los datos de su centro de salud.
RF26	El administrador deberá poder modificar los datos de su centro de salud.
RF27	El administrador deberá poder iniciar sesión.

Tabla 2: Requisitos funcionales

4.3.Requisitos no funcionales

Los requisitos no funcionales son aquellos que detallan las necesidades de la aplicación que no tienen que ver específicamente con las funcionalidades que se van a desarrollar.

ID	Descripción	Categoría	Importancia
RNF01	La aplicación deberá utilizar el idioma castellano.	Localización	Alta
RNF02	La aplicación deberá asegurar que el usuario está autentificado para poder utilizarla.	Seguridad	Alta
RNF03	La aplicación deberá contar con una interfaz de uso sencillo.	Usabilidad	Alta
RNF04	La aplicación debe contar con un mecanismo de gestión de datos fácilmente escalable.	Eficiencia	Alta
RNF05	La aplicación debe ser capaz de dar servicio a un gran número de conexiones simultáneamente sin perder rendimiento.	Eficiencia	Media

Tabla 3: Requisitos no funcionales

4.4. Consideraciones sobre seguridad

Sabemos que un sistema de este tipo nunca es seguro al cien por cien, pero siendo conocedores de que estamos tratando con una aplicación donde parte de la información que se contiene son datos sensibles de los usuarios se ha encontrado la necesidad de incluir ciertas herramientas que aseguren tanto la integridad como la seguridad de los datos que son manejados en ella con la intención de que los riesgos sean mínimos. Las medidas que se han tomado son las siguientes:

- La información sensible de los usuarios solo podrá ser accesible por aquellos usuarios que han sido identificados.

- Se dispone de tres roles gestionados al inicio de sesión con JWT lo que permite que los usuarios externos puedan acceder solamente a sus datos, los médicos solo puedan acceder a cierta información de los usuarios siempre y cuando el usuario tenga asignado dicho médico y, por último, los administradores puedan acceder a información de los usuarios que no sea considera sensible, por ejemplo, un administrador no podrá acceder al historial médico de un usuario.

- Se realizan copias de seguridad de la base de datos para prevenir perdidas o preservar la integridad de los datos almacenados.
- Las llamadas HTTP que pudiesen incluir información sensible se realizan vía peticiones POST no permitiendo así que se pueda añadir o eliminar información en ellas.

4.5. Diagramas de casos de uso

En este apartado se mostrarán los diagramas correspondientes a los diferentes roles de usuario.

4.5.1. Diagrama para el rol Usuario interno: Administrador

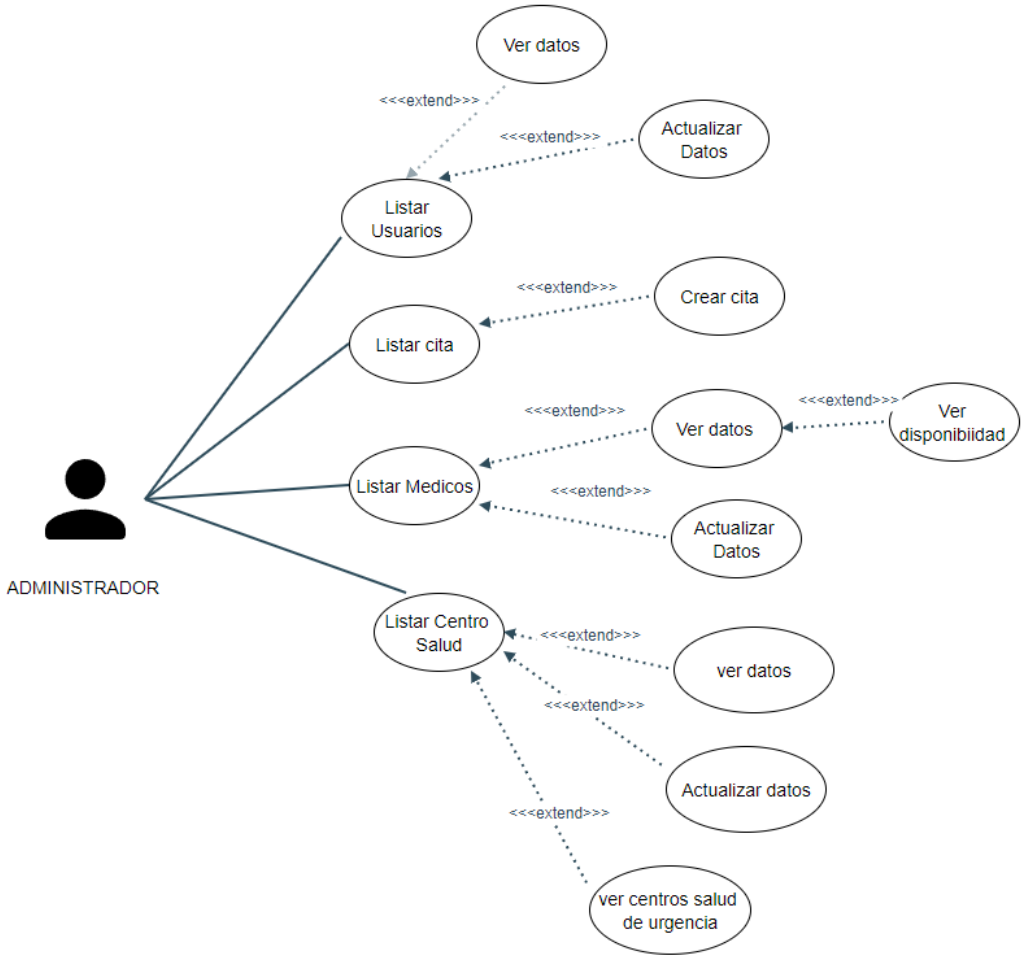

Imagen 3: Diagrama de casos de uso del rol Administrador

4.5.2. Diagrama para el rol Usuario interno: Medico

Imagen 4: Diagrama de casos de uso del rol Medico

4.5.3. Diagrama para el rol Usuario externo

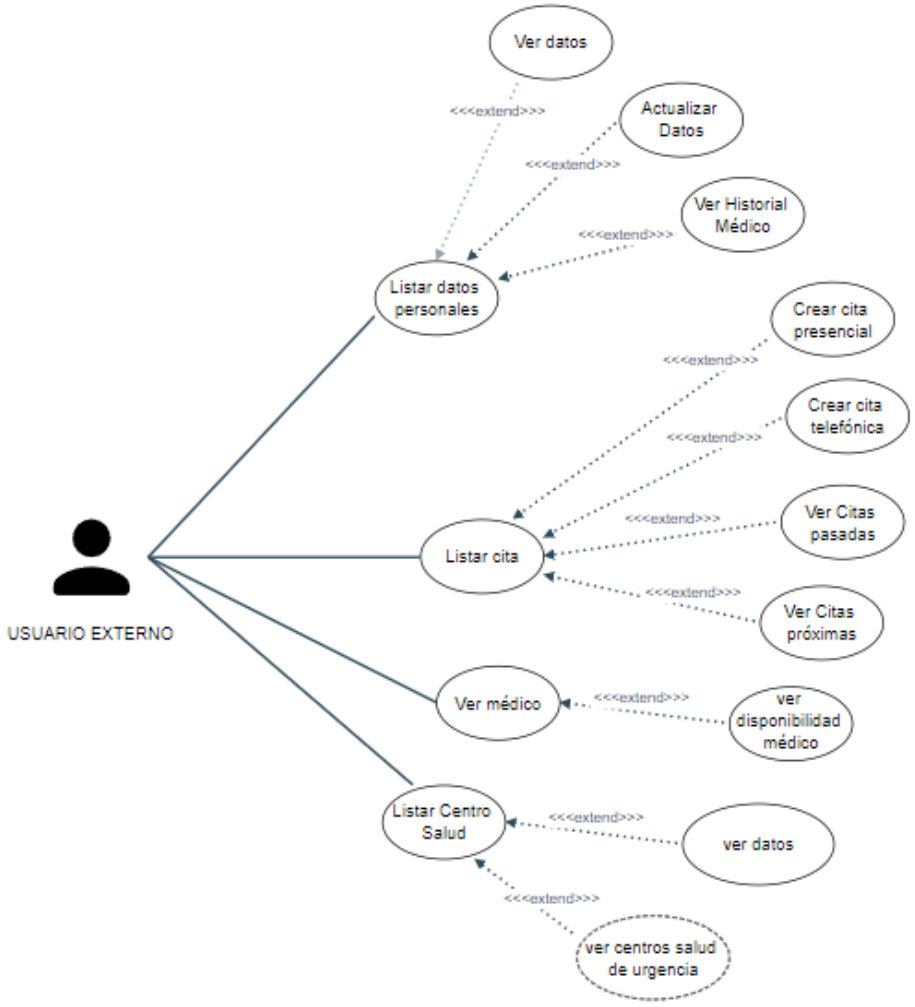

Imagen 5: Diagrama de casos de uso del rol Usuario externo

4.6. Plantillas de casos de uso

ID	P01
Caso de uso	Inicio de Sesión
Descripción	El usuario inicia sesión en la aplicación
Actores	Todos
Precondiciones	El usuario deber estar registrado en el sistema
Flujo principal	1. El usuario introduce las credenciales en los campos que se muestran en el formulario. 2. EL sistema reconoce al usuario y carga la interfaz.
Flujo alternativo	El sistema no reconoce al usuario y se muestra un mensaje de error.

Tabla 4: Casos de uso (1/6)

ID	P02
Caso de uso	Ver citas próximas
Descripción	El sistema muestra al usuario las citas que faltan por realizarse.
Actores	Usuario externo
Precondiciones	El usuario deber estar registrado en el sistema
Flujo principal	1. El usuario hace clic en el botón de citas. 2. El sistema muestra las citas del usuario que quedan por realizarse.
Flujo alternativo	El sistema no encuentra citas próximas para el usuario y no muestra la tabla

Tabla 5: Casos de uso (2/6)

ID	P03
Caso de uso	Crear cita presencial.
Descripción	El sistema registra una cita para el usuario.
Actores	Usuario externo
Precondiciones	El usuario deber estar registrado en el sistema.
Flujo principal	1. El usuario hace clic en el botón de citas. 2. El sistema muestra un calendario 3. El usuario selecciona un día 4. El sistema muestra la disponibilidad para ese día. 5. El usuario selecciona una hora 6. El usuario hace clic en el botón de hacer cita presencial o telefónica. 7. El sistema registra la cita
Flujo alternativo	

Tabla 6: Casos de uso (3/6)

ID	P04
Caso de uso	Dar de alta a un médico
Descripción	El sistema registrará un nuevo médico y lo mostrará.
Actores	Administrador
Precondiciones	El administrador deber estar registrado en el sistema y debe haber accedido a la página que gestiona los médicos.

Flujo principal	1. El administrador rellena el formulario. 2. El administrador selecciona el centro de salud al que estará adscrito el médico. 3. El administrador hará clic en el botón de registro. 4. El sistema registra el médico y refresca la lista de médicos.
Flujo alternativo	

Tabla 7: Casos de uso (4/6)

ID	P05
Caso de uso	Actualizar información centro de salud.
Descripción	El sistema actualizará el centro de salud con los datos que proporciona el administrador
Actores	Administrador
Precondiciones	El administrador deber estar registrado en el sistema y haber listado los centros de salud.
Flujo principal	1. El administrador selecciona el centro de salud 2. El sistema muestra un formulario con los datos del centro. 3. El administrador rellena los datos del formulario que desea cambiar y hace clic en el botón de registro. 4. El sistema actualiza los datos del centro de salud y refresca la lista.
Flujo alternativo	

Tabla 8: Casos de uso (5/6)

ID	P06
Caso de uso	Ver centros de salud de urgencia
Descripción	El sistema muestra los centros de salud que están disponibles para acudir fuera del horario habitual.
Actores	Usuario
Precondiciones	El usuario deber estar registrado en el sistema.
Flujo principal	1. El usuario accede a la opción de centros de salud. 2. El sistema muestra los centros de salud con el servicio de urgencias disponible.
Flujo alternativo	El sistema muestra un mensaje avisando de que no se encuentran centros de salud de urgencia en ese momento.

Tabla 9: Casos de uso (6/6)

5. Análisis y Diseño del sistema

Esta sección describe los modelos de análisis y diseño de la aplicación.

5.1. Modelo de análisis

5.1.1. Elección del patrón arquitectónico

El patrón arquitectónico es el encargado de definir la conexión entre las diferentes capas con las que cuenta el sistema y explica cómo se organiza y diseña la aplicación.

Vistos los diferentes patrones arquitecturales existentes, y considerando que lo primordial es la escalabilidad de la aplicación y la seguridad, se ha optado por utilizar un patrón de tres capas debido a que nos permite poder escalar independiente cada nivel y gestionar nosotros mismos la seguridad en la capa lógica ya que la capa de datos y presentación no podrán conectarse entre sí. Dichas capas serán: presentación, lógica de negocio y datos.

Cada una tendrá su función en la aplicación:

- Capa de presentación: es la interfaz de usuario y de comunicación de la aplicación, donde el usuario final interactúa con la aplicación. Su objetivo principal es mostrar información al usuario y recopilar datos de este.
- Capa de lógica de negocio: el núcleo de la aplicación. En esta capa se procesa y gestiona la información recopilada por la capa de presentación. Además, se encarga de realizar las conexiones entre la interfaz y la base de datos.
- Capa de datos: es la parte correspondiente a la base de datos. Es la capa donde se almacena y gestiona la información persistente procesada por la aplicación.

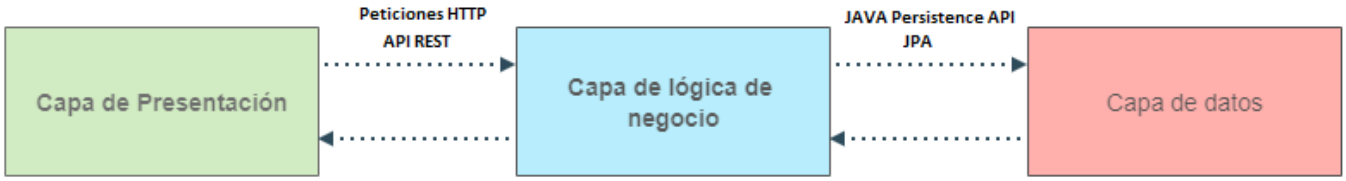

Imagen 6: Estructura del patrón arquitectural de 3 capas

5.1.2. Modelo de análisis arquitectural

Tal como se muestra en la Imagen 7, la aplicación cuenta con tres interfaces a implementar por los diferentes controladores con el fin de abordar todos los requisitos planteados en la especificación. El diagrama es el siguiente:

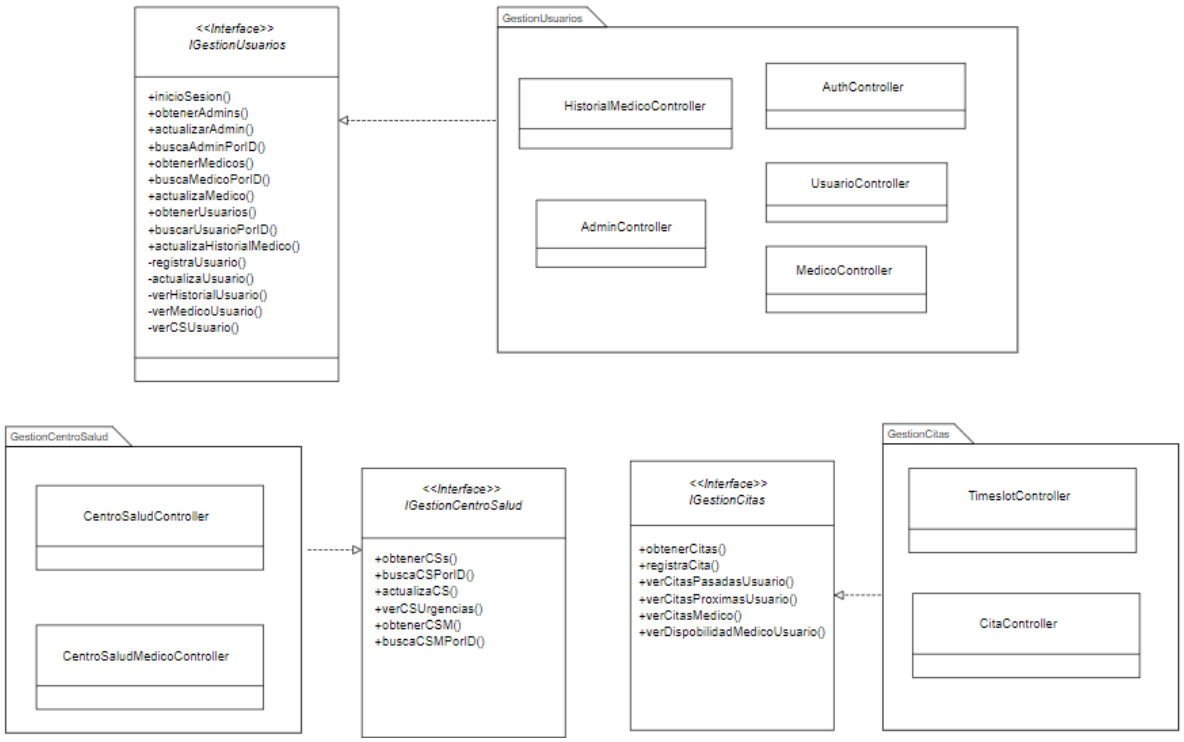

Imagen 7: Diagrama del análisis arquitectural del sistema

- IGestionUsuarios se encargará de cumplimentar los requisitos que tienen que ver con los usuarios, como pueden ser “RF01 - El usuario deberá poder acceder a sus datos” o “RF17 - El médico deberá poder iniciar sesión”.

- IGestiónCentroSalud se encargará de cumplimentar los requisitos relacionados con los centros de salud y las relaciones entre centro de salud y el médico, como puede ser “RF09 - El usuario deberá poder ver los datos de su centro de salud asignado”.

- IGestiónCitas se encargará de cumplimentar los requisitos relacionados con las citas y los timeslots, como pueden ser “RF12 - El usuario deberá poder ver la disponibilidad de su médico” o “RF13 - El médico deberá poder ver sus citas”.

5.2. Diseño arquitectónico

Como se ha explicado en el punto anterior se utiliza un modelo de tres capas para el desarrollo de este proyecto. Donde cada capa irá enlaza con las correspondientes como se puede observar en el siguiente diseño.

- Capa de datos: también conocida como capa de persistencia se basa en una base de datos relacional implementada con MySQL. Esta se encarga de guardar y modificar la información del sistema según las peticiones que reciba provenientes de la capa de lógica de negocio.

- Capa de lógica de negocio: es el núcleo de la aplicación y se basa en una API REST implementada con Java y Spring Boot. Esta capa recibe las llamadas HTTP de la capa de presentación, gestiona la información y realiza las peticiones pertinentes a la capa de datos.

- Capa de presentación: es la más visual de todas ya que desarrolla la interfaz de usuario que se implementa con React. Esta se encarga de recibir las peticiones del usuario y transmitirlas a la capa de negocio.

A continuación, se muestran los diagramas específicos de cada interfaz.

Imagen 8: Diagrama del diseño arquitectural del sistema (1/2)

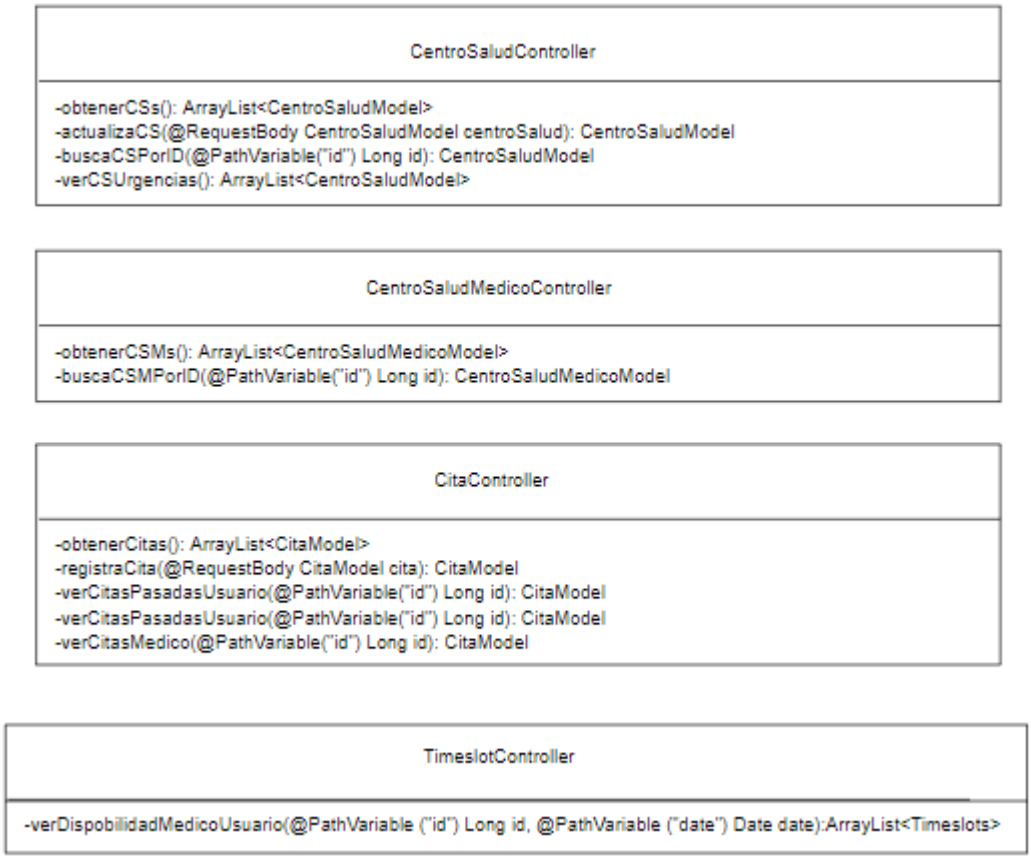

Imagen 9: Diagrama del diseño arquitectural del sistema (2/2)

5.3. Diseño de la capa de datos.

Como se explica en la fase de desarrollo (3.2.3) la capa de datos se diseña en la fase 1 del proyecto. Tras un análisis de las posibilidades que se ofrecen se decidió que la opción más viable para esto sería una base de datos relacional debido a su facilidad para ser escalada y su rendimiento, en este caso, implementada con MySql.

La Imagen 10 muestra el esquema relacional de la capa de datos el cual se puede observar que cuenta con una sola base de datos que es conformada por 11 tablas.

En primer lugar, se tiene generic_user_model que es la entidad principal diseñada para el usuario de la cual se especializan las entidades usuario, médico y admin en base al rol que desempeñan respecto al uso de la aplicación y almacena los datos principales de todos los tipos de usuarios.

La entidad centrosaludmedico proviene de una combinación de la entidad médico y la entidad centro de salud donde ambas relaciones son de tipo N a 1. Esta entidad se utilizará para realizar la conexión entre el médico y el centro de salud con los usuarios, la cual es de 1 a N. Esto es debido a que cada usuario del sistema debe tener un médico y un centro de salud asignados.

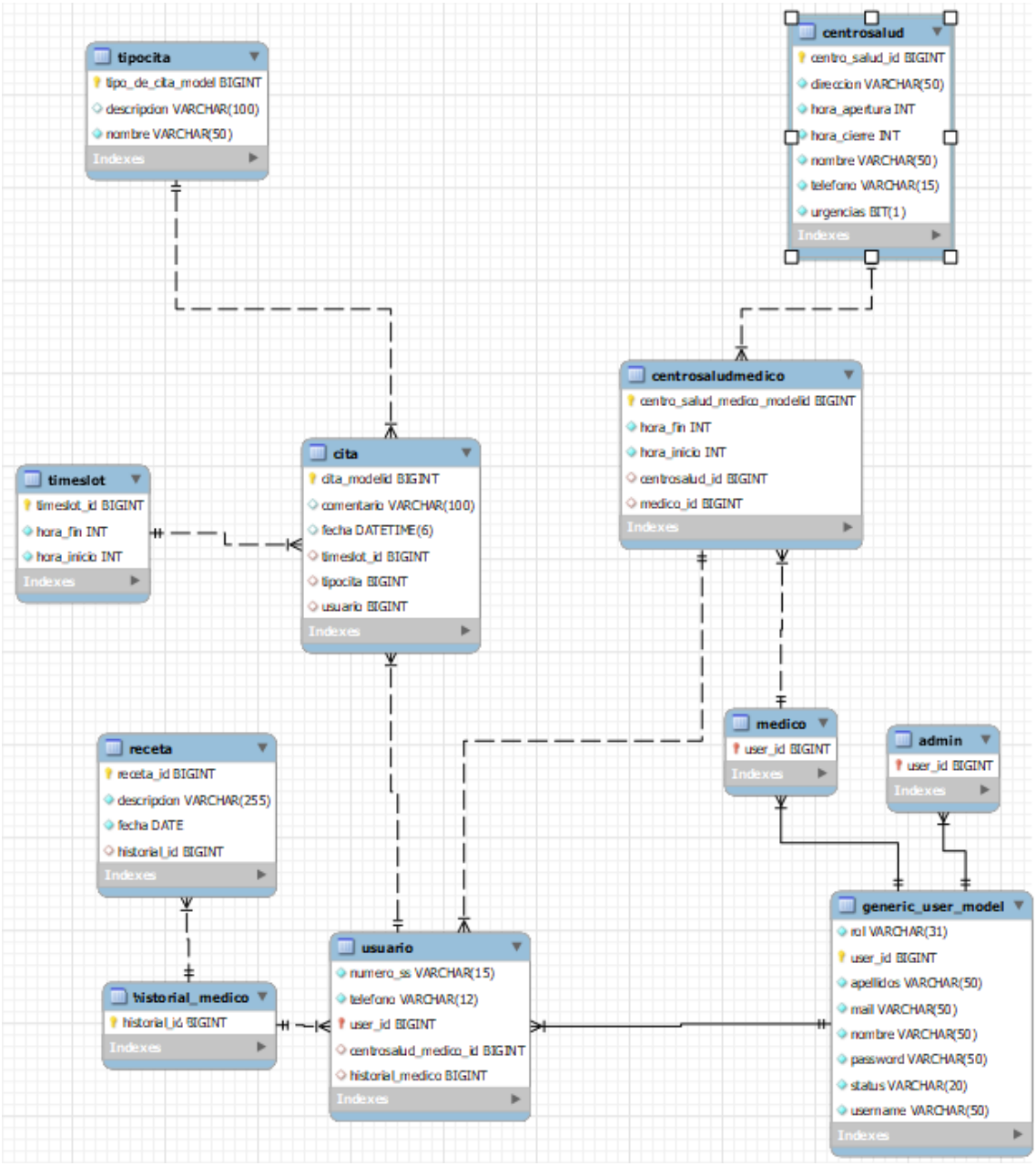

Imagen 10: Diagrama relacional de las entidades del sistema

La entidad heredada usuario cuenta con algunos datos específicos del rol de usuario externo como puede ser el número de seguridad social. Esta, además de contar con la relación explicada anteriormente cuenta con una relación de 1 a N con la entidad cita y una relación de 1 a 1 con la entidad historial_medico. Esta última tiene una relación 1 a N para poder incorporar las recetas dentro del historial médico de cada usuario.

Por último, la entidad cita también tiene una relación de 1 a 1 con tipocita, ya que esta puede ser tanto presencial como telefónica, y una relación 1 a 1 con la tabla timeslot, la cual cuenta con los horarios posibles en los que se pueden realizar citas. la adición de esta tabla es fundamental ya que permite que no se tengan que almacenar todos los

horarios disponibles de cada médico, lo que generaría una gran carga de datos en la base de datos.

5.4. Diseño de la capa de lógica de negocio.

La capa de lógica de negocio se ofrece a través de una API REST implementada con Java y Spring Boot.

La realización de esta capa se puede concebir también considerando el patrón conocido como “modelo, vista, controlador” que se basa en la separación de los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos donde:

- El modelo es una representación de los datos que contiene la capa de persistencia y será responsable de acceder a esta.
- La vista adecúa la información estructurada con los datos que provienen desde la capa de presentación y retorna los solicitados.
- El controlador actúa como intermediario gestionando el tráfico de información entre el modelo y la vista. El controlador contendrá las reglas para la gestión de los datos de la aplicación.

Esta capa, la API REST, recibe llamadas siguiendo el protocolo HTTP procedentes de la interfaz web, las procesa y gestiona la base de datos en función del contenido de esas llamadas. Para poder contemplar los errores o el buen funcionamiento durante la ejecución se utilizarán los correspondientes códigos HTTP. Algunos de estos códigos son:

- 200 OK. La petición ha sido correcta.
- 201 Created. La petición ha sido correcta y se crea un nuevo recurso.
- 204 No Content. La petición ha sido correcta y no devuelve contenido.
- 400 Bad Request. La petición contiene sintaxis errónea.
- 401 Authorization Required. La petición no se puede llevar a cabo porque no se ha iniciado sesión.
- 404 Not Found. El recurso pedido no ha sido encontrado.

Como se explicó en el apartado de planificación el desarrollo se divide en dos fases.

5.4.1. Fase 1

En esta fase se desarrollan todos los modelos e interfaces necesarias para el correcto funcionamiento de la API REST. A su vez, se desarrollan los controladores y servicios para los métodos CRUD más sencillos que se necesitan en la aplicación.

5.4.2. Fase 2

En esta fase la API REST será ampliada con aquellos servicios que requieran mayor lógica de negocio que los métodos CRUD propiamente dichos. Además, se incorporan aquellos servicios necesarios para cumplimentar los requisitos funcionales que surgen tras la revisión correspondiente a la fase 1.

5.5. Diseño de la capa web

Este apartado muestra el diseño que sigue la implementación final de la interfaz web. Esta capa web se implementa con HTML y JavaScript, más concretamente con React. Este desarrollo se planifica en la fase 2 del proyecto.

La interfaz web debe contar con un inicio de sesión genérico independientemente del tipo de usuario que intente realizar dicha actividad. Una vez el usuario inicie sesión se deberá redireccionar al usuario a la página principal que le corresponde según el rol que ocupa en la aplicación. Para obtener una idea más visual de la interfaz de la aplicación se cuenta con los siguientes mockups

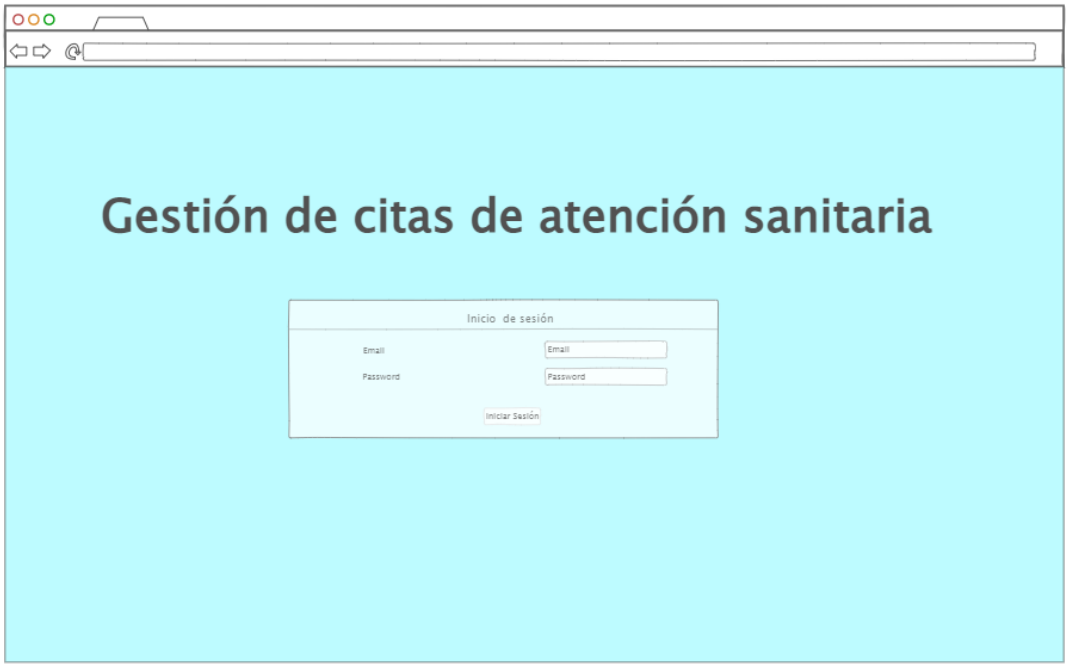

Imagen 11: Mockup de la página de inicio de sesión

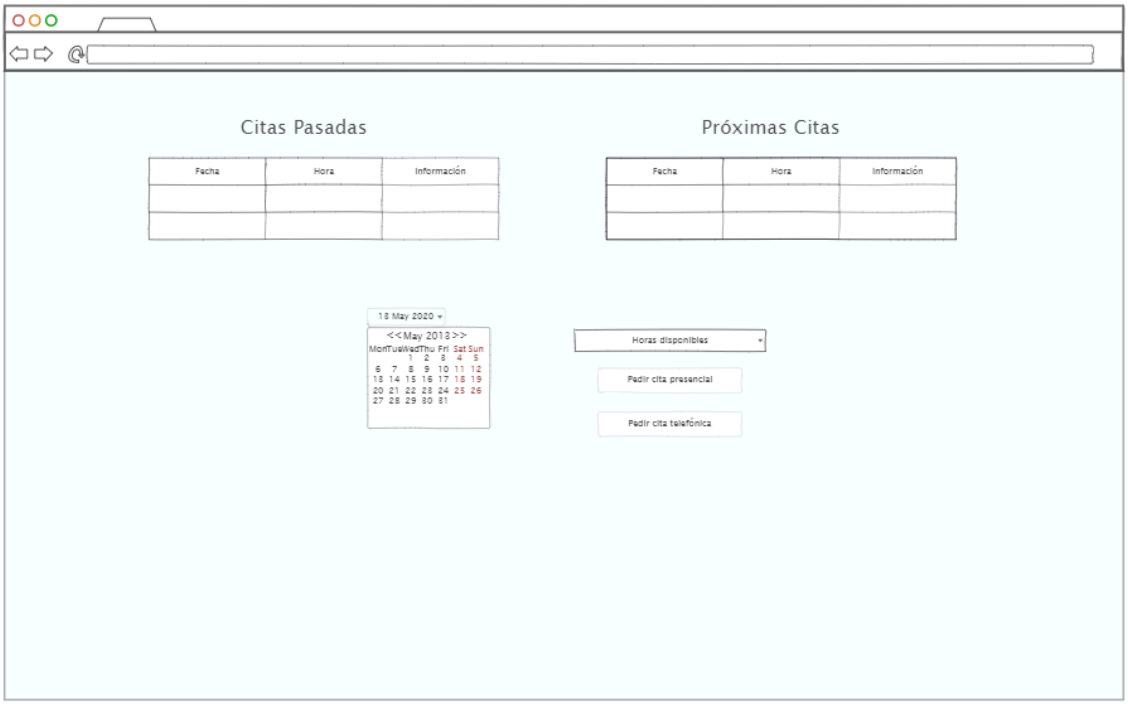

Imagen 12: Mockup de la página de gestión de citas

Imagen 13: Mockup de la gestión de usuarios de la interfaz del administrador

5.6. Modelo de despliegue

En este apartado se describe el despliegue de la aplicación durante el tiempo de desarrollo.

Durante el desarrollo se despliega la aplicación de la siguiente manera:

- La base de datos SQL se mantiene en el equipo local del desarrollador y se despliega en el puerto “localhost:3306”.
- La API REST se despliega en un servidor tomcat proporcionado por Maven y springboot. Se accede a ella a través de la url: “http://localhost:3000/”
- La interfaz web se despliega en el servidor de desarrollo en el puerto 8080, es decir a través de la url: “http://localhost:8080/”.

6. Implementación

En este apartado se describe la implementación del código de la aplicación tanto para el back-end como para el front-end. El código desarrollado se encuentra disponible de forma abierta en https://github.com/jmo985/TrabajafoFinDeGrado.

6.1. Implementación del *back-end*

El back-end cuenta con una API REST desarrollada en Java utilizando los frameworks Spring y JPA usando para ello eclipse como entorno de desarrollo siguiendo el patrón modelo-vista-controlador.

6.1.1. API REST

Los paquetes principales de la API REST son: modelos, repositorios, controladores y servicios. A continuación, se describen utilizando un ejemplo de cada uno de ellos tomado del código implementado.

- Modelos:

```
@Entity
@DiscriminatorValue(value= "USUARIO")
@Table(name = "usuario")
public class UsuarioModel extends GenericUserModel {


    @Column(nullable = false, length = 12)
    private String telefono;

    @Column(unique = true, nullable = false, length = 15, name="numero_ss")
    private String numeroSS;

    @JsonIgnore
    @OneToOne(mappedBy = "usuarioModelHistorial")
    private HistorialMedicoModel historialMedico;

    @ManyToOne
    @JoinColumn(name = "centrosalud_medico_id")
    private CentroSaludMedicoModel centroSaludMedicoModel;

    @JsonIgnore
    @OneToMany(mappedBy="usuarioModel")
    private Set<CitaModel> citaModel = new HashSet<>();

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public String getNumeroSS() {
        return numeroSS;
    }

    public void setNumeroSS(String numeroSS) {
        this.numeroSS = numeroSS;
    }

    public HistorialMedicoModel getHistorialMedico() {
        return historialMedico;
    }
```

Imagen 14: Implementación del modelo usuario

La Imagen 14 muestra el modelo de la clase UsuarioModel, la cual extiende a su vez GenericUserModel. Estos modelos mapean los datos de las entidades de la base de datos gracias a las anotaciones JPA y contiene los getters y setters correspondientes.

Las anotaciones JPA que se observan en la imagen son las siguientes:

- @Entity: Define la clase como una entidad mapeada de la base de datos
- @DiscriminatorValue: Define el valor que debe tener el campo con el que se diferencian los distintos tipos de entidades heredadas.
- @Table: Define el nombre de la tabla de la base de datos.
- @Column: Define el nombre de la columna de la base de datos.
- @JsonIgnore: Indica que cuando se realicen solicitudes HTTP ese valor no se mostrará en el JSON resultante.
- @ManyToOne: Define una relación de N a 1.
- @OneToMany: Define una relación de 1 a N.
- @OneToOne: Define una relación 1 a 1.

- Repositorios

```
package com.sanidadgobcan.tfguc.repositories;

import org.springframework.data.repository.CrudRepository;

@Repository
public interface UsuarioRepository extends CrudRepository<UsuarioModel, Long> {
}
```

Imagen 15: Implementación del repositorio usuario

Los repositorios son interfaces que se encargan de acceder a la capa de persistencia de datos gracias a la etiqueta @Repository del paquete JPA. Estas interfaces son extendidas de CrudRepository lo que nos ayuda a poder realizar métodos CRUD (acrónimo por sus siglas en inglés que representa a los métodos dedicados a la creación, lectura, actualización y eliminación de ítems de la base de datos) de manera sencilla ya que permite hacer búsquedas por cualquiera de los atributos de las entidades.

- Controladores

```java
@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/usuario")
public class UsuarioController {

    @Autowired
    UsuarioRepository usuarioRepository;

    @Autowired
    PasswordEncoder encoder;

    @Autowired
    UsuarioService usuarioService;

    @GetMapping("/listar")
    public ResponseEntity<List<UsuarioModel>> obtenerUsuario(){
        List<UsuarioModel> users= (ArrayList<UsuarioModel>) usuarioRepository.findAll();
        return ResponseEntity.ok(users);
    }



    @SuppressWarnings("unchecked")
    @GetMapping("/buscar/{id}")
    public ResponseEntity<Optional<UsuarioModel>> buscaUsuarioPorID(@PathVariable("id") Long id) {
        Optional<UsuarioModel> user= usuarioRepository.findById(id);
        if(user.isPresent()) {
            return ResponseEntity.ok(user);
        }else {
            return (ResponseEntity<Optional<UsuarioModel>>) ResponseEntity.notFound();
        }
    }
```

Imagen 16: Implementación parcial del controlador usuario

En la Imagen 16 se puede observar algunos de los métodos CRUD del controlador de usuario y su definición con anotaciones JPA. Este controlador se utiliza para recibir las llamadas HTTP procedentes del front-end y realizar la consecuente lógica con los datos recibidos. Las etiquetas más importantes de este tipo de clases son:

- @CrossOrigin: Habilita las solicitudes HTTP de origen cruzado.
- @RestController: Define la clase como un controlador de API REST.
- @RequestMapping: Define la ruta a través de la cual se podrá acceder al controlador.
- @Autowired: Inyecta las dependencias de manera automática
- @GetMapping: Define la ruta para llamar a dicho método GET.
- @PostMapping: Define la ruta para llamar a dicho método POST.
- @PatchMapping: Define la ruta para llamar a dicho método PATCH.
- @PathVariable: Indica el valor que se le debe pasar como parámetro a la llamada en la URL.
- @RequestBody: Indica el conjunto de datos que se debe pasar a la llamada en el body en formato JSON.

- Servicios

```
public ArrayList<CitaModel> verCitasProximasUsuario(Long idUsuario){
    ArrayList<CitaModel> citas= verCitasUsuario(idUsuario);
    ArrayList<CitaModel> citasAux=new ArrayList<CitaModel>();
    Date date=new Date(System.currentTimeMillis());
    for(CitaModel cita:citas) {
        if(cita.getFecha().after(date)) {
            citasAux.add(cita);
        }
    }
    return citasAux;
}
```

Imagen 17: Implementación del servicio verCitasProximasUsuario

Los servicios, marcados con la anotación @Service son una capa intermedia necesaria para aquellas funcionalidades que necesitan una lógica para su implementación de una mayor complejidad. Estos servicios serán llamados posteriormente por el controlador.

En la Imagen 17 podemos ver un método del Servicio correspondiente a las citas donde se devuelven las citas, posteriores a la fecha en la que se haga la llamada, que le corresponden a un usuario pasado por parámetro.

6.1.2. Swagger

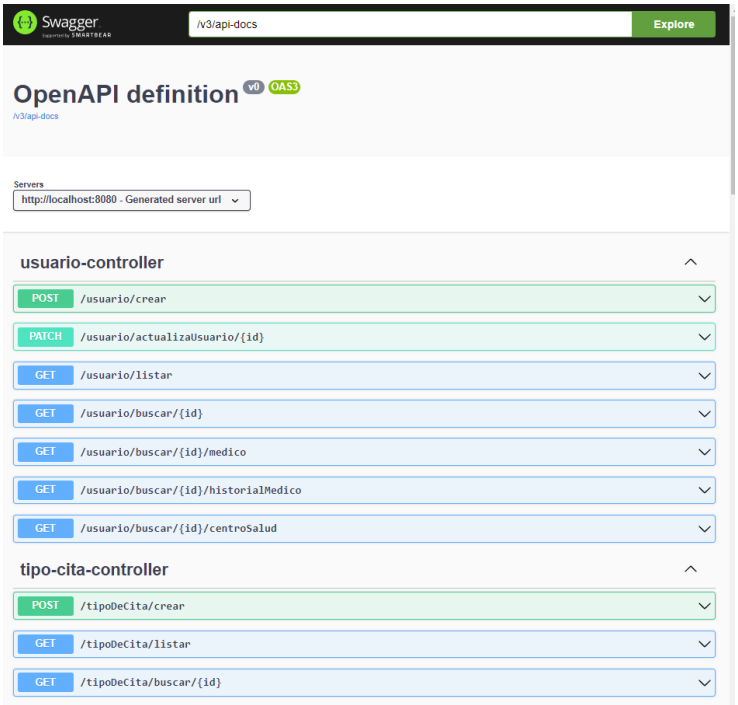

Imagen 18: Swagger de la aplicación

Con la utilización del paquete swagger se habilitó una página donde se muestra tanto la definición de las llamadas que están implementadas en la API REST como los modelos. Este acceso vía Swagger se encuentra configurado en la plataforma de desarrollo y es accesible a través de la url: "http://localhost:8080/swagger-ui/index.html".

6.1.3. Seguridad

Por último, se implementó la seguridad y el acceso restringido a las llamadas de la API REST con JWToken. Para ello se emplean 4 paquetes distintos:

- En el paquete payload se implementan las clases JwtReponse, LoginRequest y MessageResponse las cuales contienen las propiedades necesarias del JWT.
- En el paquete security.jwt se implementan las clases necesarias para la creación de los tokens y su gestión, estas clases son AuthEntryPointJwt, AuthTokenFilter y JwtUtils.
- En el paquete security.services se implementan las clases UserDetailsImpl y UserDetailsServiceImpl usadas para gestionar los usuarios en el momento del inicio de sesión.
- En el paquete security.config se implementa la clase WebSecurityConfig que contiene la configuración de seguridad de la aplicación.

En la Imagen 19 podemos ver el módulo con el cual se permite el acceso a la página de inicio de sesión y al swagger a cualquier usuario, mientras que al resto de peticiones solo se podrá acceder si se tiene un token JWT.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().and().csrf().disable()
        .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
        .authorizeRequests().antMatchers("/auth/login").permitAll().antMatchers( "/swagger-ui/**").permitAll().anyRequest().authenticated();
    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
}
```

Imagen 19: Implementación de la seguridad de la aplicación

6.1.4. Propiedades de la aplicación

En la carpeta src/main/resources del proyecto se encuentra el archivo aplication.properties que contiene las siguientes propiedades de configuración de la aplicación:

- La información para la conexión con la base de datos a través de Hibernate se configura con las siguientes propiedades:
  - spring.datasource.url define la dirección en la que almacena la base de datos.

    - spring.datasource.username define el usuario con el que acceder a la base de datos.
    - spring.datasource.password define la contraseña del usuario con el que acceder a la base de datos.
    - spring.jpa.hibernate.ddl-auto define como se trata la base de datos respecto a los cambios en el modelo. Puede tener el valor create para eliminar el contenido y añadirlo de nuevo al lanzar el servidor, el valor update para que se actualice o el valor none para que no se produzcan cambios en la base de datos. En la versión final a desplegar contendrá el valor none.
- La información para generar los JWToken se configura con:
    - sanidadtfg.app.jwtSecret define el nombre que se le dará al token.
    - sanidadtfg.app.jwtExpirationMs define el tiempo de validez de un token en milisegundos.

6.2. Implementación del front-end

El front-end es una interfaz web basada en React que utiliza JavaScript, HTML y CSS, para ello se ha usado Visual Studio Code como entorno de desarrollo.

El archivo App.js contiene la función principal de la aplicación, con este archivo y gracias a "react-router-dom" se crean todas las rutas accesibles que tiene nuestra interfaz web. La imagen 20 muestra las páginas implementadas en este front-end.

```
function App() {
  return (
    <div className="App">
      <Router>
        <div>
          <Routes>
            <Route exact path="/HomeAdmin/:id" element={<HomeAdmin/>}/>
            <Route exact path="/HomeUsuario/:id" element={<HomeUsuario/>}/>
            <Route exact path="/HomeMedico/:id" element={<HomeMedico/>}/>
            <Route exact path="/Usuario/Datos" element={<UsuarioDatos/>}/>
            <Route exact path="/Usuario/Citas" element={<UsuarioCitas/>}/>s
            <Route exact path="/Usuario/CentroSaludMedico" element={<UsuarioCS/>}/>
            <Route exact path="/" element={<Login/>}/>
            <Route exact path="/Medico/Datos" element={<MedicoDatos/>}/>
            <Route exact path="/Medico/Citas" element={<MedicoCitas/>}/>
            <Route exact path="/Medico/CentroSalud" element={<MedicoCS/>}/>
            <Route exact path="/Admin/Datos" element={<AdminDatos/>}/>
            <Route exact path="/Admin/Usuarios" element={<AdminUsuarios/>}/>
            <Route exact path="/Admin/Medicos" element={<AdminMedicos/>}/>
            <Route exact path="/Admin/CrearCita" element={<AdminCrearCita/>}/>
            <Route exact path="/Admin/CentroSalud" element={<AdminCS/>}/>
          </Routes>
        </div>
      </Router>
    </div>
  );
}
```

Imagen 20: Implementación de App.js

6.2.1. Funciones JavaScript

En este apartado se describe el código JavaScript que da servicio a dos de las páginas de la aplicación. Una de las partes más importantes de React es que permite crear interfaces con HTML mientras se intercala JavaScript en el código para determinadas funciones.

```
export default function UsuarioCS() {
  const [centroSalud, setCentroSalud] = useState();
  const [medico, setMedico] = useState();
  const [CSUrgencias, setCSurgencias] = useState();
  const [user] = useState(JSON.parse(localStorage.getItem("user")));
  const token = useState("Bearer " + user.accessToken);

  useEffect(() => {
   medicoFind();
   CSFind();
   CSurgenciasFind();


  },[]);

  function medicoFind() {
    axios
      .get("http://localhost:8080/usuario/buscar/" + user.id + "/medico", {
        headers: {
          Authorization: token[0],
        },
      })
      .then(function (response) {
        setMedico(response.data);
      })
      .catch(function (error) {
        console.log(error.message);
      });
  }
```

Imagen 21: Implementación de la página UsuarioCS

En la Imagen 21 se puede observar parte de la codificación correspondiente a la página utilizada para ver los datos del centro de salud que tiene asignado un usuario.

- En primer lugar, se definen los hooks, estos son variables de estado que pueden almacenar cualquier tipo de dato de manera que no sea necesario la utilización de componentes para cambiar el estado de un objeto, ya que estos nos proporcionan además un setter para modificar su valor. En el caso de los hooks creados en la imagen tenemos tres de ellos que se inicializan a null. En el caso de user se está cargando en la variable un JSON con ciertos datos del usuario que están almacenados en la cache del navegador como puede ser el id o el token JWT. Por último, el hook llamado token es cargado con el “Bearer” y el JWT que se cargó previamente en user.

- La función useEffect() es una función de react que se ejecuta cuando el componente es renderizado, a esta función se le añade "[]" como segundo argumento, lo que significa que solo se ejecutará una vez por renderizado. En este caso, useEffect() llama a tres funciones que han sido implementadas en esa misma página.
- La función medicoFind() es una función interna que nos permite hacer una llamada GET a la API REST gracias a la librería axios. El primer parámetro es la dirección a la que se realiza la llamada y el segundo el header, este debe contener el token JWT para que la aplicación permita realizar la petición. Por último, se controla la respuesta obtenida de la petición, para esta función si la petición es correcta se cambiará el valor del hook medico por el que nos devuelve la llamada. En caso de algún error en la petición se mostrará un mensaje, por ejemplo, si no se incorporase el token en el header la aplicación devolvería un error 401 Unauthorized.

```
export default function Login() {
  //const [user, setUser] = useState({ username: "", password: "" });
  const [username, setUsername] = useState();
  const [password, setPassword] = useState();
  const [message, setMessage] = useState();

  let navigate = useNavigate();

  function login(e) {
    e.preventDefault();
    axios
      .post("http://localhost:8080/auth/login", {
        username: username,
        password: password,
      })
      .then(function (response) {
        localStorage.setItem("user", JSON.stringify(response.data));
        handleUser(response.data.roles.authority, response.data.id);
      })
      .catch(function (error) {
        setMessage("fallo al iniciar sesion");
      });
  }

  function handleUser(role, id) {
    switch (role) {
      case "ADMIN":
        navigate(`/HomeAdmin/${id}`);
        break;

      case "MEDICO":
        navigate(`/HomeMedico/${id}`);
        break;

      case "USUARIO":
        navigate(`/HomeUsuario/${id}`);
        break;
      default:
        break;
    }
  }
```

Imagen 22: Implementación de la página de inicio de sesión

Para el caso del inicio de sesión tenemos algo similar al resto de la aplicación con algunas peculiaridades.

- La función login() se hace a través de una llamada POST a la URL del controlador de la API REST. Para esta función no necesitamos ningún header ya que, como se explica en el apartado de seguridad del back-end, la llamada login permite peticiones de todos los orígenes debido a que es la que permitirá a la API generar el token si los datos son correctos.
- Como segundo parámetro de la llamada se pasa un body en formato JSON con las credenciales que se introducen en la interfaz.
- En caso de que la respuesta a la llamada POST sea positiva la función login() almacenará en la cache del navegador los datos de usuario devueltos con la función localStorage.setItem(). Además, se invoca a la función handleUser() la cual, usando las propiedades de useNavigate(), redireccionará al usuario a la página inicial correspondiente según su rol.
- En caso de un error de autenticación con la llamada el hook message cambiará su valor.

6.2.2. Interfaz de usuario HTML.

Para la realización de la parte visual se utiliza HTML mientras que para el estilo se utiliza css.

```
return (
  <div>
    <div>
      <div className="container">
        <div className="row">
          {medico && <div className="col-6">
            <label>Datos del médico </label>
            <br />
            <br />
            <label> Nombre: {medico.nombre}</label>
            <br />
            <label> Apellidos: {medico.apellidos}</label>
            <br />
            <label> Correo electrónico: {medico.mail}</label>
          </div>}
          {centroSalud && <div className="col-6">
            <label>Datos del Centro Salud </label>
            <br />
            <br />
            <label> Nombre: {centroSalud.nombre}</label>
            <br />
            <label> Dirección: {centroSalud.direccion}</label>
            <br />
            <label>
              {" "}
              Horario: {centroSalud.horaApertura} - {centroSalud.horaCierre}{" "}
            </label>
            <br />
            <label> Teléfono: {centroSalud.telefono}</label>
          </div>}
        </div>
      </div>
    </div>
  </div>
);
```

Imagen 23: Implementación de la página UsuarioCS

En la Imagen 23 se observa el contenido HTML de la página correspondiente a mostrar los datos del médico y del centro de salud de la interfaz del usuario.

Algunos de los de la página tienen un atributo className, este valor viene dado del repositorio de css de Bootstrap. A la hora de colocar los componentes en la interfaz Bootstrap nos proporciona el ClassName “row” el cual divide la pantalla en doce columnas de igual tamaño. En este caso, lo que se consigue con el valor “col-6” es dividir la pantalla en dos columnas de igual tamaño, de manera que la distribución de información en la pantalla sea simétrica.

```
return (
  <div>
    <form>
      <div className="row mb-3">
        <label className="col-mb-6 col-form-label">Nombre de usuario</label>
        <div className="col-mb-10">
          <input
            className=""
            type="text"
            onChange={(e) => setUsername(e.target.value)}
          ></input>
        </div>
      </div>

      <div className="row mb-3">
        <label className="col-mb-6 col-form-label">Contraseña</label>
        <div className="col-mb-10">
          <input
            className=""
            type="password"
            onChange={(e) => setPassword(e.target.value)}
          ></input>
        </div>
      </div>

      <button type="submit" className="btn btn-primary" onClick={login}>
        Iniciar sesion
      </button>
    </form>
    {message && <div className="alert alert-danger" role="alert">{message}</div>}
  </div>
);
```

Imagen 24: Implementación HTML de la página de inicio de sesión

Para el inicio de sesión se renderiza un formulario para que el usuario introduzca las credenciales. El método onChange() hace que los valores de los hooks cambien con la información que se añade en los inputs del formulario. En este caso, al igual que antes tanto el botón como el que contiene el mensaje llevan un className que les proporciona un formato obtenido de Bootstrap. Además, en caso de que el valor del hook message cambiase se renderizaría de nuevo la página y se mostraría el error.

7. Pruebas y resultados

7.1. Pruebas Unitarias

Las pruebas unitarias comprueban, de manera independiente, el correcto funcionamiento de cada uno de los métodos de las clases que componen la API REST. Para la implementación de dichas pruebas se ha utilizado la librería spring-boot-starter-test perteneciente a Springboot. Para ello se hace uso de la anotación @WebMvcTest que se encarga de preparar el mock para hacer llamadas HTTP al controlador correspondiente.

```
@WebMvcTest(CentroSaludController.class) public class CentroSaludControllerTest {

    @Autowired private MockMvc mock;

    @MockBean private CentroSaludController centroSaludController;

    @Autowired private ObjectMapper mapper;

    @Test public void buscarPorIDTest() throws Exception{ CentroSaludModel cs=new
        CentroSaludModel(); cs.setCentroSaludId((long) 1);
        cs.setDireccion("Direccion test"); cs.setHoraApertura("09:00");
        cs.setHoraCierre("17:00"); cs.setNombre("CS test"); cs.setTelefono("test");

        ResponseEntity<CentroSaludModel> csRE = new ResponseEntity<>(cs,
        HttpStatus.OK);

        when(centroSaludController.buscaCSPorID((long) 1)).thenReturn(csRE);
        var verUrgencias = mock.perform(get("/centroSalud/1").accept(MimeTypeUtils.
        APPLICATION_JSON_VALUE)).andExpect(status().isOk()).andReturn(); var test=
        mapper.readValue(verUrgencias.getResponse().getContentAsString(),
        CentroSaludModel.class);
        assert test.getCentroSaludId().equals(cs.getCentroSaludId());
        assert test.getDireccion().equals(cs.getDireccion());
        assert test.getHoraApertura().equals(cs.getHoraApertura());
        assert test.getHoraCierre().equals(cs.getHoraCierre());
        assert test.getNombre().equals(cs.getNombre());
        assert test.getTelefono().equals(cs.getTelefono());
    }
}
```

Imagen 25: Implementación prueba unitaria

En la Imagen 25 se puede observar una prueba unitaria de una petición GET de la API REST que utiliza buscaCSporID() perteneciente al controlador de la entidad centro de salud. En este caso, el mock realiza la llamada HTTP correspondiente, simulando así, una petición realizada desde el front-end. Para simular el acceso a la capa de persistencia se crea un objeto de la entidad CentroSaludModel que es lo que devuelve el controlador. Posteriormente, se corrobora que los datos obtenidos del controlador son los mismos a los obtenidos desde la capa de persistencia.

7.2. Pruebas de Integración

Las pruebas de integración son aquellas que una vez comprobados los módulos de los controladores de manera independiente nos van a permitir comprobar el correcto funcionamiento de dichos módulos, pero, esta vez, conectados con la capa de persistencia.

Estas pruebas se han realizado de manera manual, utilizando la herramienta Postman. Postman nos permite realizar llamadas HTTP a nuestra API REST sin necesidad de una interfaz web. Las pruebas de integración se realizaron de abajo a arriba (bottom-up) integrando primero la capa de negocio con la capa de persistencia y, posteriormente, el front-end sobre el backend. Para cada servicio implementado en los controladores se han realizado las correspondientes pruebas con el fin de verificar la viabilidad en el funcionamiento de ambas capas juntas. Para ello se realizaba una petición válida a cada servicio y se comprobaba que la respuesta era la esperada, teniendo en cuenta tanto los estatus correctos como los incorrectos.

7.3. Pruebas de aceptación

Las pruebas de aceptación corresponden a la última etapa de pruebas relacionadas con la aplicación. En ellas se comprueba que la aplicación contiene y realiza de manera correcta las funcionalidades que se han propuesto para el desarrollo del proyecto.

En este caso, para realizar dichas pruebas se ha comprobado el correcto funcionamiento de todas las funcionalidades implementadas en el front-end de manera manual en el servidor local. La imagen 26 muestra las pruebas realizadas para la gestión de citas de un usuario por parte de un administrador.

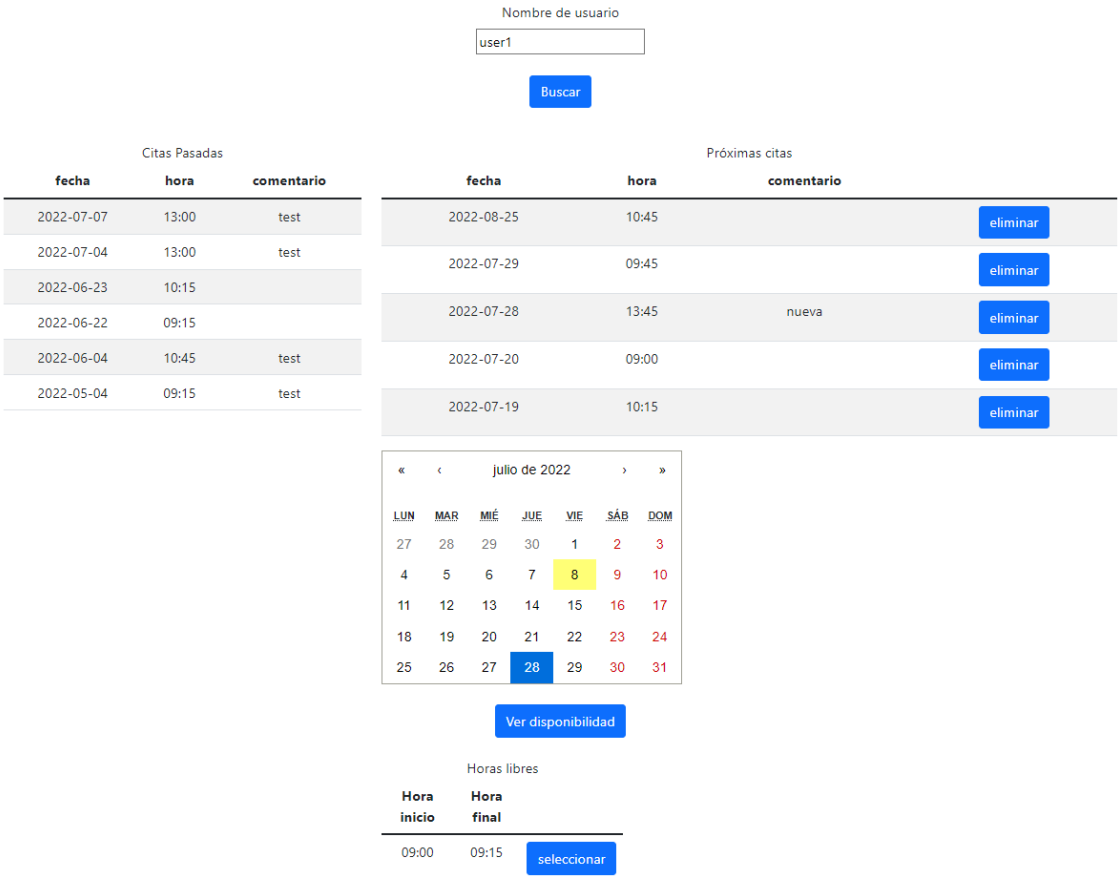

Imagen 25: vista de una de las pruebas de aceptación de la implementación

8. Conclusiones y trabajos futuros

Esta sección esboza algunas conclusiones, conocimientos adquiridos y apunta algunas funcionalidades que podrían enriquecer la aplicación en sus siguientes versiones.

8.1. Conclusiones

En primer lugar, podemos confirmar que se ha cumplido con los objetivos básicos propuestos. La aplicación fullstack se ha desarrollado en un tiempo aproximado de 2 meses, siendo implementadas todas las funcionalidades propuestas para las dos primeras fases del desarrollo.

En segundo lugar, este trabajo ha permitido profundizar en los conocimientos aprendidos en el grado sobre los procesos de ingeniería de software que se siguen habitualmente para el desarrollo profesional de aplicaciones software, desde análisis de requisitos hasta implementación. Por otra parte, ha servido para mejorar mi capacidad de estimación de los tiempos de desarrollo, ya que me propuse un plan riguroso y mis estimaciones mejoraron de una fase a la siguiente. En cualquier caso, aunque algunas funcionalidades han llevado algo más de tiempo del esperado estas han sido compensadas con otras que han llevado menos tiempo del estimado.

Adicionalmente, este proyecto me ha servido para introducirme en entornos y tecnologías de las cuales era conocedor, pero sobre las que no había indagado en profundidad. Respecto al back-end he ampliado los conocimientos que tenía sobre librerías de Java, tales como Springboot, durante el desarrollo de la API REST y la utilización de Spring Security para la realización de aplicaciones seguras. En cuanto a la parte del front-end, este desarrollo me ha permitido familiarizarme con JavaScript, componentes HTML y CSS gracias al uso de React y la librería Bootstrap.

8.2. Trabajos futuros

Una vez implementadas las funcionalidades de la aplicación, se han encontrado otras que podrían ser atractivas para el usuario como pueden ser:

- Integración de SMS con la información de la cita.
- Integración de una cola de email con la información de la cita.
- Mensajes emergentes en respuesta a la interacción del usuario con la aplicación.

Por otra parte, para el correspondiente despliegue del proyecto en producción se han valorado distintas opciones y se piensa que la mejor herramienta para desplegar el sistema es Microsoft Azure [17], ya que nos permite almacenar la base de datos y desplegar nuestro back-end en sus servidores. Además, brinda elementos como las

“aplicaciones lógicas” [18] que permitirían un desarrollo más sencillo de las funcionalidades que se han comentado anteriormente.

9. Bibliografía*

[1] MySQL_Workbench. https://es.wikipedia.org/wiki/MySQL_Workbench

[2] Entorno de desarrollo integrado. Eclipse IDE para Java. http://www.edu4java.com/es/java/eclipse-ide-entorno-de-desarrollo.html

[3] Spring Framework. https://www.techtarget.com/searchapparchitecture/definition/Spring-Framework

[4] ¿Qué es Spring Boot? https://blog.codmind.com/que-es-spring-boot/#:~:text=Spring%20Boot%20permite%20compilar%20nuestras,de%20aplicaciones%20en%20el%20propio%20.

[5] ¿Qué es Java Hibernate? ¿Por qué usarlo? https://ifgeekthen.nttdata.com/es/que-es-java-hibernate-por-que-usarlo

[6] ¿Qué es Maven y para qué se utiliza?. http://panamahitek.com/que-es-maven-y-para-que-se-utiliza/

[7] SWAGGER Y SWAGGER UI: ¿QUÉ ES Y POR QUÉ ES IMPRESCINDIBLE PARA TUS APIS? https://www.chakray.com/es/swagger-y-swagger-ui-por-que-es-imprescindible-para-tus-apis/

[8] Qué es Visual Studio Code. https://openwebinars.net/blog/que-es-visual-studio-code-y-que-ventajas-ofrece/

[9] React. https://es.reactjs.org/

[10] Acerca de Node.js. https://nodejs.org/es/about/

[11] Bootstrap (framework). https://es.wikipedia.org/wiki/Bootstrap_(framework)

[12] Cómo realizar pruebas automatizadas con Postman. https://www.encora.com/es/blog/como-realizar-pruebas-automatizadas-con-postman

[13] GitHub. https://es.wikipedia.org/wiki/GitHub

[14] JavaScript. https://es.wikipedia.org/wiki/JavaScript

[15] CSS, ¿qué es?. http://www.arumeinformatica.es/blog/css/

[16] ¿Qué es JSON Web Token?. https://platzi.com/blog/introduccion-json-web-tokens/

[17] Microsoft Azure. https://es.wikipedia.org/wiki/Microsoft_Azure

[18] ¿Qué es Azure Logic Apps? https://docs.microsoft.com/es-es/azure/logic-apps/logic-apps-overview

* Los enlaces a internet han sido accedidos el día 04/07/2022 por última vez.