Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de una Aplicación Web para la Gestión de Accesos de Seguridad

Autor: Fernando Romero Calvo

Tutor(a): Sergio Paraíso Medina

Madrid, mayo 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Desarrollo de una Aplicación Web para la Gestión de Accesos de Seguridad
Mayo 2024

Autor: Fernando Romero Calvo

Tutor:
Sergio Paraíso Medina
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Este proyecto se enfoca en la creación de una aplicación web diseñada específicamente para el control de accesos de personas en entornos de alta seguridad, tales como edificios gubernamentales o instalaciones militares, donde la vigilancia y la gestión eficiente son de suma importancia.

Esta aplicación, concebida como una solución completa y versátil, tiene como objetivo principal registrar de manera precisa la entrada y salida de personas autorizadas, garantizando un sistema seguro y eficiente. Para lograr este propósito, se ha empleado una arquitectura basada en llamadas API REST, que permite una comunicación fluida entre el frontend y el backend, maximizando así la capacidad de respuesta y escalabilidad del sistema.

En cuanto a la tecnología utilizada, se ha optado por herramientas modernas y robustas. En el frontend, se ha empleado React, un framework de JavaScript ampliamente reconocido por su capacidad para construir interfaces de usuario dinámicas y altamente interactivas. Por otro lado, en el backend se ha hecho uso de Spring Boot junto con Java, una combinación que proporciona un entorno de desarrollo sólido y altamente eficiente para la creación de aplicaciones web de alto rendimiento.

Además, la aplicación cuenta con una base de datos MySQL para el almacenamiento seguro y eficiente de todos los datos relacionados con los accesos y las autorizaciones. Esta base de datos será fundamental para garantizar la integridad y la disponibilidad de la información, así como para permitir su fácil consulta y manipulación a través de la interfaz de usuario.

El sistema cuenta con un robusto sistema de autenticación y registro de usuarios que garantiza la seguridad y la privacidad de la información. Los usuarios autenticados tienen acceso a una variedad de funcionalidades dentro de la aplicación para administrar eficazmente los accesos y las autorizaciones.

La aplicación está organizada en tres módulos principales, cada uno diseñado para cubrir una faceta específica de la gestión de accesos. En primer lugar, el módulo de gestión de visitantes ofrece una interfaz intuitiva para dar de alta nuevos visitantes mediante un formulario de creación o revisar los registros existentes, que se muestran en una tabla interactiva.

El segundo módulo, dedicado a las autorizaciones, permite a los usuarios crear nuevas autorizaciones o consultar las ya existentes. Este aspecto es esencial para planificar y coordinar las visitas, debiendo proporcionar información crucial como el intervalo de fechas en el que el visitante está autorizado a acceder y pasar el control, el propósito de su visita o la asociación de visitantes a los que se les autoriza el acceso.

Por último, el tercer módulo de registros facilita el seguimiento de las entradas de las personas. La aplicación implementa rigurosas verificaciones para asegurar que cada persona registrada este autorizada para acceder en la fecha específica, garantizando así un control de acceso coherente y seguro en todo momento.

Este último módulo se divide en dos partes: la parte de “Registrar entrada”, donde se nos mostrará una tabla con todos los usuarios asignados a una autorización que le permite el paso por el control de seguridad para el día actual, existiendo la opción de registrar su entrada. En el momento en el que se registre su entrada, constará como que el visitante ha pasado el control. La segunda parte del módulo es la de “Registrar salida”, que muestra una tabla con todas

las personas que se encuentran dentro de la infraestructura que esté gestionada por este control de seguridad, teniendo la opción de registrar la salida del visitante.

En resumen, el Trabajo de Fin de Grado se centra en el desarrollo de una aplicación web avanzada que aprovecha al máximo las tecnologías modernas disponibles, con el objetivo de optimizar la gestión de accesos en entornos de alta seguridad, proporcionando una solución integral y efectiva para el control de accesos de personas autorizadas.

Abstract

This project focuses on the creation of a web application designed specifically for the access control of people in high security environments, such as government buildings or military installations, where surveillance and efficient management are of utmost importance.

This application, conceived as a complete and versatile solution, has as its main objective to accurately record the entry and exit of authorized people, guaranteeing a safe and efficient system. To achieve this purpose, an architecture based on REST API calls has been used, which allows fluid communication between the frontend and the backend, thus maximizing the responsiveness and scalability of the system.

Regarding the technology used, modern and robust tools have been chosen. On the frontend, React has been used, a JavaScript framework widely recognized for its ability to build dynamic and highly interactive user interfaces. On the other hand, in the backend, Spring Boot has been used along with Java, a combination that provides a solid and highly efficient development environment for the creation of high-performance web applications.

In addition, the application has a MySQL database for the secure and efficient storage of all data related to access and authorizations. This database will be essential to guarantee the integrity and availability of the information, as well as to allow its easy consultation and manipulation through the user interface.

The system has a robust user authentication and registration system that guarantees the security and privacy of the information. Authenticated users have access to a variety of functionality within the application to effectively manage access and authorizations.

The application is organized into three main modules, each designed to cover a specific facet of access management. First, the visitor management module offers an intuitive interface to register new visitors using a creation form or review existing records, which are displayed in an interactive table.

The second module, dedicated to authorizations, allows users to create new authorizations or consult existing ones. This aspect is essential to plan and coordinate visits, and must provide crucial information such as the date range in which the visitor is authorized to access and pass control, the purpose of their visit or the association of visitors to whom they are authorized. the access.

Finally, the third registration module makes it easy to track people's entries. The application implements rigorous verifications to ensure that each registered person is authorized to access on the specific date, thus guaranteeing consistent and secure access control at all times.

This last module is divided into two parts: the “Register entry” part, where we will be shown a table with all the users assigned to an authorization that allows them to pass through the security control for the current day, with the option to register your entry. At the time the entry is registered, it will appear that the visitor has passed the control. The second part of the module is “Register exit”, which shows a table with all the people who are within the

infrastructure that is managed by this security control, having the option of registering the visitor's exit.

In summary, the Final Degree Project focuses on the development of an advanced web application that makes the most of the modern technologies available, with the aim of optimizing access management in high security environments, providing a comprehensive and effective solution for access control of authorized persons.

Tabla de contenidos

1 Introducción

1.1 Contexto, planteamiento de la necesidad y motivación

En un mundo cada vez más interconectado y tecnológicamente dependiente, el ámbito de la seguridad y el control de accesos también se ha visto afectado, convirtiéndose en un aspecto crítico para una amplia gama de organizaciones, desde instituciones gubernamentales y militares hasta empresas privadas y centros educativos, que no deben perder la oportunidad evolutiva en el sector tecnológico para ofrecer un servicio más seguro y eficiente. La necesidad de proteger activos físicos, datos confidenciales, y lo que es más importante, la seguridad de las personas, ha impulsado la búsqueda de soluciones innovadoras y eficientes en el ámbito de los controles de accesos.

Existe una creciente demanda de sistemas de gestión de accesos que sean no solo efectivos, sino también ágiles, adaptables y altamente seguros. El objetivo principal es abordar los desafíos propios de la gestión de accesos en entornos de seguridad, donde la precisión, la rapidez y la fiabilidad son fundamentales para garantizar un funcionamiento sin contratiempos.

En muchas instituciones y organizaciones, la gestión de accesos se ha enfrentado a diversos obstáculos, como por ejemplo los centenarios cuadernos de registros, herramientas manuales propensas a errores y lentos en su uso. La necesidad de una solución moderna se hace evidente en un mundo donde la agilidad y la seguridad son imperativos ineludibles y de fácil acceso gracias a las nuevas tecnologías que surgen casi a diario.

Es por todo esto por lo que me gustaría destacar el gran impacto que un sistema de control de accesos eficiente puede tener en la productividad de una organización. Al optimizar los procesos de entrada y salida, se pueden reducir los tiempos de esperar y mejorar la experiencia del usuario, igual que el fácil acceso a la información de todos los accesos que se han realizado sobre la institución con gran detalle de información puede elevar el nivel óptimo de seguridad que produce. La capacidad de crear soluciones sencillas y eficientes con el desarrollo de software sumada a la capacidad de mejorar procesos que se llevan años realizando de una forma que seguramente parecía impensable mejorar debido a los recursos que se disponían, pudiendo influir directamente en ese cambio, son las motivaciones que han hecho que el desarrollo de este proyecto me haya resultado fascinante.

1.2 Solución propuesta y objetivos

La solución propuesta para abordar los desafíos en la gestión de accesos en entornos de seguridad se fundamenta en el desarrollo de una aplicación web. Esta solución basará en una arquitectura moderna y escalable, haciendo uso de tecnologías avanzadas para garantizar un rendimiento óptimo y una experiencia de usuario amigable.

Para cumplir con estos objetivos se han asumido las siguientes premisas técnicas:

La aplicación debe proporcionar al usuario una experiencia intuitiva y fluida, para poder cumplir con la finalidad de su uso de la forma más eficiente posible. Para ello la solución consta de un frontend desarrollado en React, un framework de JavaScript ampliamente reconocido por su capacidad para construir interfaces de usuario dinámicas y altamente interactivas.

Para la comunicación entre el frontend y el backend se ha optado por utilizar la tecnología API REST, que cumple con los requisitos de eficiencia y flexibilidad que necesita esta aplicación.

Por otro lado, el backend de la aplicación se ha implementado utilizando Spring Boot con Java, donde Hibernate, una librería de mapeo objeto-relacional que permite el mapeo de atributos entre una base de datos relacional y un modelo de datos de objetos en una aplicación, se ha utilizado para la comunicación con la base de datos, en cuyo caso se ha optado por MySQL. Esta elección tecnológica asegura un manejo eficiente de los datos, permitiendo el almacenamiento, la edición y la recuperación de toda la información necesaria de una manera óptima y segura.

Como objetivo extra para este proyecto, me he propuesto garantizar la calidad del código y mantener altos estándares de limpieza y mantenibilidad, para lo cual he utilizado el plugin de SonarLint, instalado directamente en IDE de Spring Boot. Esta herramienta proporciona un análisis estático del código en tiempo real, identificando posibles problemas de calidad y sugiriendo mejoras para asegurar un desarrollo consistente y libre de errores.

1.3 Planificación y diagrama de Gantt

Este proyecto se ha realizado durante gran parte del primer semestre de 2024, como Trabajo de Fin de Grado para la titulación de Ingeniería Informática. Para la realización de este se ha cumplido una planificación que dividía el proyecto en 8 hitos que se han ido completando con la resolución de las metas descritas en cada uno de ellos.

1. Planificación del proyecto
    - Definir el tema y los objetivos del proyecto.
    - Establecer los objetivos específicos a lograr.
    - Elaborar un plan de trabajo detallado que incluya fechas límite para cada etapa del proyecto.
2. Diseño de la aplicación.
    - Crear bocetos de la interfaz de usuario y diseñar la arquitectura general de la aplicación.
    - Definir los modelos de datos necesarios para la aplicación.

3. Desarrollo del backend con Spring Boot
    - Configurar el entorno de desarrollo para Spring Boot.
    - Implementar las entidades y relaciones de base de datos utilizando JPA/Hibernate.
    - Desarrollar controladores RESTful para manejar las solicitudes HTTP.
4. Desarrollo del frontend con React
    - Configurar el entorno de desarrollo para React.
    - Diseñar y desarrollar los componentes de la interfaz de usuario.
    - Consumir los endpoints API del backend para obtener y enviar datos.
5. Integración y pruebas
    - Integrar el frontend y el backend para asegurar su funcionamiento conjunto.
    - Realizar pruebas de integración para verificar la interoperabilidad entre las partes.
    - Realizar pruebas de usuario para obtener retroalimentación sobre la usabilidad y la experiencia del usuario.
6. Corrección y ajustes
    - Corregir errores identificados durante las pruebas y a justar la funcionalidad según sea necesario.
    - Optimizar el código y la interfaz de usuario para mejorar la experiencia del usuario.
    - Realizar pruebas adicionales para validar las correcciones realizadas.
7. Documentación y presentación
    - Preparar documentación técnica del proyecto, incluyendo manuales de usuario si es necesario.
    - Redacción de una memoria del proceso de desarrollo de la aplicación.
    - Preparar una presentación del proyecto que destaque los objetivos, el proceso de desarrollo y los objetivos obtenidos.
8. Entrega y evaluación.
    - Entregar el trabajo de fin de grado según las especificaciones y plazos establecidos por la ETSIINF.
    - Participar en la defensa oral del proyecto, explicando los detalles técnicos y respondiendo preguntas de los evaluadores.

Para la realización y cumplimiento de cada uno de los hitos descritos me he propuesto las fechas límite indicadas en el siguiente diagrama de Gantt:

Nombre de la tarea	Fecha de inicio	Fecha de finalización	05.02.2024	12.02.2024	19.02.2024	26.02.2024	04.03.2024	11.03.2024	18.03.2024	25.03.2024	01.04.2024	08.04.2024	15.04.2024	22.04.2024	29.04.2024	06.05.2024	13.05.2024	20.05.2024	27.05.2024	03.06.2024	10.06.2024	17.06.2024	24.06.2024
Planificación del proyecto	05.02.2024	19.02.2024	■	■	■																		
Diseño de la aplicación	12.02.2024	26.02.2024		■	■	■																	
Desarrollo del backend	26.02.2024	04.03.2024				■	■																
Desarrollo del frontend	04.03.2024	01.04.2024					■	■	■	■	■												
Integración y pruebas	01.04.2024	15.04.2024									■	■	■										
Corrección y ajustes	15.04.2024	29.04.2024											■	■	■								
Documentación	29.04.2024	17.06.2024													■	■	■	■	■	■	■	■	
Entrega y presentación	17.06.2024	20.06.2024																				■	

Ilustración 1 - Diagrama de Gantt

2 Estado del arte

2.1 API REST

En esta sección, se explicará de forma detallada el funcionamiento de la arquitectura API REST [1] . Una API [2] (Application Programming Interface) es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas. El formato de intercambio de datos normalmente es JSON o XML. REST [3] (REpresentational State Transfer) es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP para solicitar funciones de bases de datos estándar como crear, leer, actualizar y eliminar registros dentro de un recurso. Esta arquitectura hace uso de verbos HTTP [4], que son aquellos verbos propios del protocolo HTTP que fueron tomados para definir operaciones muy puntuales y específicas sobre los recursos de una API. Los más utilizados son:

GET: listado de recursos o detalle de un solo recurso.

POST: creación de un nuevo recurso.

PUT: modificación total de un recurso.

DELETE: eliminación de un recurso.

2.2 Spring Boot

Java Spring Framework [5] (Spring Framework) es un popular marco de trabajo empresarial de código abierto que sirve para crear aplicaciones autónomas de producción que se ejecutan en una máquina virtual Java (JVM). Java Spring Boot es una herramienta que acelera y simplifica el desarrollo de microservicios y aplicaciones web con Spring Framework gracias a su capacidad de configuración automática y de crear aplicaciones autónomas. La popularidad de Spring Boot para el desarrollo de este tipo de aplicaciones se debe a su función de inyección de dependencias, que permite a los desarrolladores crear aplicaciones modulares que constan de componentes sin conexión directa, ideales para microservicios.

2.3 MySQL

MySQL [6] es el sistema de gestión de bases de datos relacional más extendido en la actualidad. Es un sistema de gestión de bases de datos de código abierto que permite a los desarrolladores disponer de una solución fiable y estandarizada para sus aplicaciones. MySQL se basa en un modelo cliente-servidor, es decir, clientes y servidores se comunican entre sí de manera diferenciada para un mejor rendimiento. Cada cliente puede hacer consultas a través del sistema de registro para obtener datos o modificarlos.

2.4 React

ReactJS [7] es una de las librerías más populares de JavaScript para el desarrollo de aplicaciones web. React contiene una colección de fragmentos de código JavaScript reutilizables utilizados para crear interfaces de usuario llamadas componentes. React utiliza JSX, una extensión de la sintaxis de JavaScript que permite incrustar código HTML en objetos JavaScript.

2.5 SonarLint

SonarLint [8] es un complemento IDE de código abierto que resalta, en tiempo real, errores y vulnerabilidades en el código mientras los estas escribiendo. Funciona como un corrector ortográfico y ofrece soluciones rápidas para resolver los problemas

3 Requisitos y casos de uso

En este apartado, se detallan los requisitos y casos de uso de la aplicación desarrollada. Los requisitos describen las funcionalidades y características que el sistema debe cumplir para satisfacer las necesidades de los usuarios y asegurar un funcionamiento eficiente y seguro. Los casos de uso, por su parte, proporcionan una descripción clara y estructurada de cómo los usuarios interactuarán con el sistema para llevar a cabo tareas específicas. Esta sección es fundamental para entender las expectativas y el alcance del proyecto, así como para guiar el proceso de desarrollo y asegurar que todos los componentes necesarios estén correctamente implementados.

3.1 Requisitos funcionales

3.1.1 Gestión de usuarios

- **Iniciar sesión**: El sistema debe permitir el inicio de sesión a los usuarios con las credenciales (nombre de usuario y contraseña) registradas en la aplicación.
- **Cerrar sesión**: El sistema debe permitir el cierre de la sesión del usuario.
- **Registrar usuario**: El sistema debe permitir el registro de un nuevo usuario en la aplicación en caso de no estar dado de alta en la misma, indicando los campos nombre, primer apellido, segundo apellido, correo electrónico, nombre de usuario y contraseña.

3.1.2 Gestión de visitantes

- **Alta de un nuevo visitante**: El sistema debe permitir el alta de un nuevo visitante en la aplicación indicando los campos DNI, nombre, primer apellido, segundo apellido, fecha de nacimiento, nombre del padre y nombre de la madre.
- **Consulta de visitantes**: El sistema debe permitir la búsqueda y consulta de visitantes registrados, pudiendo opcionalmente filtrar la búsqueda por criterios como DNI, nombre, primer apellido, segundo apellido, nombre del padre o nombre de la madre.
- **Edición de un visitante**: El sistema debe permitir la edición de la información de un visitante, actualizando dicha información en la base de datos de la aplicación.
- **Borrado de un visitante**: El sistema debe permitir el borrado de un visitante de la base de datos de la aplicación.

3.1.3 Gestión de autorizaciones

- **Alta de una nueva autorización**: El sistema debe permitir la creación de autorizaciones, especificando el motivo de la autorización, el tipo de autorización y las fechas de inicio y fin del vigor de la autorización.

- **Consulta de autorizaciones**: El sistema debe permitir la búsqueda y consulta de autorizaciones registradas, pudiendo opcionalmente filtrar la búsqueda por criterios como el motivo de la autorización o el tipo de la autorización.
- **Edición de una autorización**: El sistema debe permitir la edición de una autorización para poder asignar o desasignar visitantes de esta.
- **Borrado de una autorización**: El sistema debe permitir el borrado de una autorización de la base de datos, borrando como consecuencia la relación de cualquier visitante a esa autorización.

3.1.4 Gestión de registros

- **Consulta de registros de entrada permitidos**: El sistema debe proporcionar una lista de visitantes cuyas autorizaciones está vigentes en el día actual y no tienen fecha de entrada registrada.
- **Registro de entrada**: El sistema debe permitir registrar la entrada de cualquier visitante mostrado en la consulta anterior.
- **Consulta de registros de salida**: El sistema debe proporcionar una lista de visitantes a los que se les ha registrado la entrada y están disponibles para que su salida sea registrada.
- **Registro de salida**: El sistema debe permitir registrar la salida de cualquier visitante mostrado en la consulta anterior.

3.2 Requisitos no funcionales

3.2.1 Usabilidad

- **Interfaz de usuario intuitiva**: La interfaz de usuario debe ser intuitiva y fácil de usar, permitiendo a los usuarios realizar tareas comunes sin necesidad de formación extensa.

3.2.2 Compatibilidad

- **Compatibilidad con navegadores**: El sistema debe ser compatible con los navegadores web más utilizados, incluyendo Opera, Google Chrome, Mozilla Firefox, Microsoft Edge.

3.3 Requisitos técnicos

3.3.1 Plataforma

- **Lenguaje de programación**: El sistema debe estar desarrollado en Java usando Spring Boot
- **Base de datos**: El sistema debe utilizar una base de datos relacional para el almacenamiento de datos.
- **Servidor de aplicaciones**: El sistema debe ser capaz de ejecutarse en un servidor de aplicaciones como Tomcat.

3.3.2 Herramientas y Frameworks

- **Framework de desarrollo**: Spring Boot para el backend.
- **Framework de persistencia**: Spring Data JPA para la gestión de la persistencia de datos.
- **Frontend**: React para el desarrollo de la interfaz de usuario.

3.4 Casos de uso

3.4.1 Iniciar sesión

Precondición: El usuario debe estar registrado en la aplicación.

Postcondicion: Se inicia la sesión del usuario.

Escenario:

1. El usuario accede a la aplicación.
2. El sistema muestra la pantalla de login.
3. El usuario completa los campos nombre de usuario y contraseña.
4. El usuario envía el formulario.
5. El sistema valida las credenciales y da acceso a la aplicación en caso de ser correctos.

3.4.2 Cerrar sesión

Precondición: El usuario debe estar autenticado.

Postcondición: Se cierra la sesión del usuario.

Escenario:

1. El usuario selecciona la opción “Cerrar sesión”
2. El sistema pide confirmación de la acción.
3. El usuario confirma la acción.
4. El sistema cierra la sesión del usuario y redirige a la pantalla de login.

3.4.3 Registrar nuevo usuario

Precondición: El usuario debe tener acceso a la aplicación.

Postcondición: Se registra un nuevo usuario en la base de datos de la aplicación.

Escenario:

1. El usuario accede a la aplicación
2. El sistema muestra la pantalla de login.
3. El usuario presiona sobre el botón de “Registrarse”
4. El sistema muestra el formulario de registro.

5. El usuario completa los campos de nombre, primer apellido, segundo apellido, correo electrónico, nombre de usuario y contraseña y envía el formulario.
6. El sistema comprueba que no existe un usuario con ese nombre de usuario y lo registra en la aplicación, mostrando la pantalla principal de la aplicación.

3.4.4 Alta de un nuevo visitante

Precondición: El usuario debe estar autenticado.

Postcondición: Se añade un nuevo visitante a la aplicación

Escenario:

1. El usuario selecciona la opción de “Visitantes”.
2. El sistema muestra el módulo de visitantes.
3. El usuario selecciona la opción de “Alta nuevo visitante”.
4. El sistema muestra el formulario de alta de un nuevo visitante.
5. El usuario completa los campos del formulario y lo envía.
6. El sistema comprueba que no exista un visitante con ese DNI y lo registra en la aplicación.

3.4.5 Consulta de visitantes

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema muestra el listado de visitantes.

Escenario:

1. El usuario selecciona la opción de “Visitantes”.
2. El sistema muestra el módulo de visitantes.
3. El usuario pulsa sobre el botón “Buscar”, pudiendo antes completar alguno de los campos de filtrado.
4. El sistema recupera la lista de visitantes que cumplen con los filtros cumplimentados o todos si no se ha aplicado ninguno, y los muestra en la tabla.

3.4.6 Edición de un visitante

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema edita el visitante con los cambios especificados por el usuario.

Escenario:

1. El usuario selecciona la opción de “Visitantes”.
2. El sistema muestra el módulo de visitantes.
3. El usuario realiza una búsqueda de visitantes.
4. El sistema muestra el listado de visitantes.
5. El usuario pulsa sobre el botón “editar” correspondiente a uno de los visitantes.
6. El sistema muestra una pantalla con los campos de los visitantes pudiendo editarlos.

7. El usuario edita los campos requeridos y pulsa sobre “guardar”.
8. El sistema guarda los cambios sobre el visitante.

3.4.7 Borrado de un visitante

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema elimina el registro correspondiente al visitante.

Escenario:

1. El usuario selecciona la opción de “Visitantes”.
2. El sistema muestra el módulo de visitantes.
3. El usuario realiza una búsqueda de visitantes.
4. El sistema muestra el listado de visitantes.
5. El usuario pulsa sobre el botón “borrar” correspondiente a uno de los visitantes.
6. El sistema pide confirmación de la acción.
7. El usuario confirma la acción.
8. El sistema elimina el registro correspondiente al visitante.

3.4.8 Alta de una nueva autorización

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema crea un nuevo registro correspondiente a una autorización

Escenario:

1. El usuario selecciona la opción de “Autorizaciones”
2. El sistema muestra el módulo de autorizaciones.
3. El usuario pulsa sobre el botón “Alta nueva autorización”
4. El sistema muestra el formulario de creación de una autorización
5. El usuario completa el formulario y pulsa sobre “guardar”.
6. El sistema crea el registro correspondiente a la nueva autorización.

3.4.9 Consulta de autorizaciones

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema muestra el listado de autorizaciones

Escenario:

1. El usuario selecciona la opción de “Autorizaciones”.
2. El sistema muestra el módulo de autorizaciones.
3. El usuario pulsa sobre el botón “Buscar”, pudiendo antes completar alguno de los campos de filtrado.
4. El sistema recupera la lista de autorizaciones que cumplen con los filtros cumplimentados o todas si no se ha aplicado ninguno, y los muestra en la tabla.

3.4.10 Edición de una autorización

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema edita la autorización.

Escenario:

1. El usuario selecciona la opción de “Autorizaciones”.
2. El sistema muestra el módulo de autorizaciones.
3. El usuario realiza una búsqueda de autorizaciones.
4. El sistema muestra el listado de autorizaciones.
5. El usuario pulsa sobre el botón “editar” correspondiente a una de las autorizaciones.
6. El sistema muestra una pantalla con los campos de las autorizaciones y una tabla con los visitantes asignados a dicha autorización.
7. El usuario puede seleccionar un visitante en el combo y pulsar sobre “asignar” o pulsar sobre “desasignar” en uno de los visitantes asociados a la autorización.
8. El sistema asigna o desasigna al visitante de la autorización dependiendo de la acción que se haya realizado.

3.4.11 Borrado de una autorización

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema edita la autorización.

Escenario:

1. El usuario selecciona la opción de “Autorizaciones”.
2. El sistema muestra el módulo de autorizaciones.
3. El usuario realiza una búsqueda de autorizaciones.
4. El sistema muestra el listado de autorizaciones.
5. El usuario pulsa sobre el botón “borrar” correspondiente a una de las autorizaciones.
6. El sistema pide confirmación de la acción.
7. El usuario confirma la acción.
8. El sistema elimina el registro correspondiente a la autorización y los registros que relacionan sus visitantes asociados a la misma.

3.4.12 Consulta de registros de entrada permitidos

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema muestra la lista de visitantes autorizados a pasar por el control de acceso para el momento en el que se lanza la petición.

Escenario:

1. El usuario selecciona la opción de “Registrar entrada”
2. El sistema recoge todos los registros de visitantes que pertenecen a una autorización válida para el momento en el que se envía la petición y los muestra en la tabla que aparece al mostrar dicho módulo.

3.4.13 Registro de entrada

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema registra la entrada del visitante seleccionado.

Escenario:

1. El usuario selecciona la opción de “Registrar entrada”.
2. El sistema muestra el módulo de registros de entrada, mostrando los visitantes autorizados a entrar.
3. El usuario pulsa sobre el botón “registrar entrada” correspondiente a uno de los visitantes.
4. El sistema pide confirmación de la acción.
5. El usuario confirma la acción.
6. El sistema registra la entrada del visitante, haciendo que desaparezca de la tabla.

3.4.14 Consulta de registros de salida

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema muestra la lista de visitantes a los que se les ha registrado la entrada.

Escenario:

1. El usuario selecciona la opción de “Registrar entrada”
2. El sistema recoge todos los registros de visitantes a los que se les ha registrado la entrada y los muestra en la tabla que aparece en la pantalla principal al acceder al módulo.

3.4.15 Registro de salida

Precondición: El usuario debe estar autenticado.

Postcondición: El sistema registra la salida del visitante seleccionado.

Escenario:

1. El usuario selecciona la opción de “Registrar salida”.
2. El sistema muestra el módulo de registros de salida, mostrando los visitantes a los que se les ha registrado la entrada.
3. El usuario pulsa sobre el botón “registrar salida” correspondiente a uno de los visitantes.
4. El sistema pide confirmación de la acción.
5. El usuario confirma la acción.
6. El sistema registra la salida del visitante, haciendo que desaparezca de la tabla.

4 Diseño

En este apartado se mostrará en detalle el diseño del sistema, haciendo una primera explicación general de la arquitectura y explicando más tarde cada una de las capas de la aplicación.

4.1 Arquitectura

Para el desarrollo de esta aplicación se ha optado por una arquitectura de 3 capas [9], una de las arquitecturas más comunes para aplicaciones cliente-servidor. Esta arquitectura se organiza en tres niveles y cada capa es un proceso separado y bien definido, lo que permite ventajas como un desarrollo más rápido en caso de tratarse de un desarrollo en equipo o la seguridad que proporciona esa independencia de las capas.

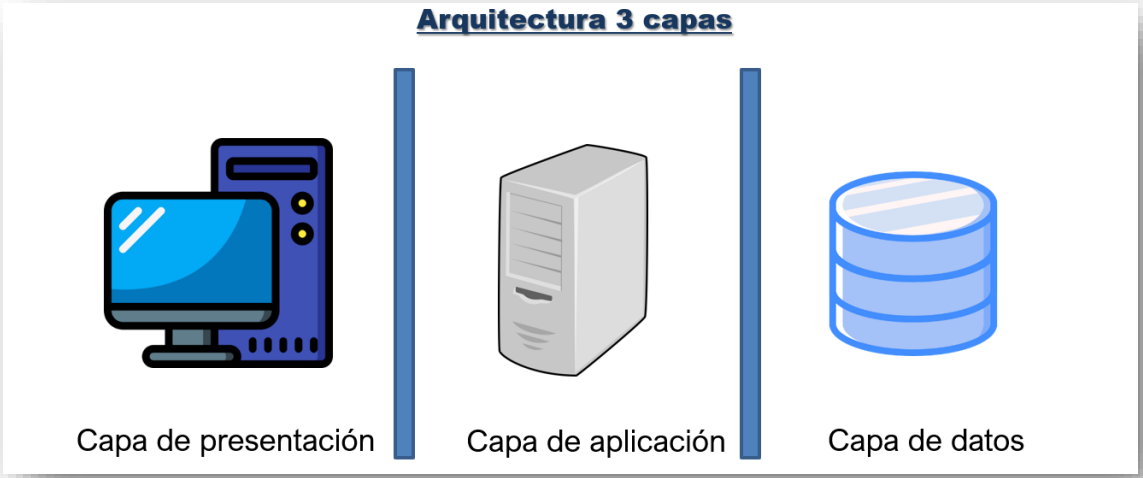


Ilustración 2 - Arquitectura 3 capas

Las capas en las que se divide está aplicación son la capa de presentación, la capa de aplicación y la capa de datos.

- La capa de presentación es el punto de encuentro entre el usuario y la aplicación, permitiendo la interacción directa del usuario con la misma. Su función primordial radica en la exhibición de datos al usuario.
- La capa de aplicación es la capa intermedia que conecta la capa de presentación y la de datos y constituye el núcleo de la aplicación. Aquí es donde se lleva a cabo el procesamiento de la información recopilada en la capa de presentación, a menudo contrastándola con otros datos de la capa de datos. El uso principal de esta capa en esta aplicación es la de añadir, eliminar recoger o modificar datos de la capa de datos.
- La capa de datos es el lugar donde se guarda y administra toda la información procesada por la aplicación.

4.2 Diseño de la capa de presentación

En este apartado se mostrará en detalle el diseño de la capa de presentación o Frontend de la aplicación, que es la interfaz de usuario con la que los usuarios interactuarán directamente.

4.2.1 Estructura del proyecto React

A continuación, se muestra la estructura del proyecto que da forma a la capa de presentación de la aplicación utilizando el framework React.

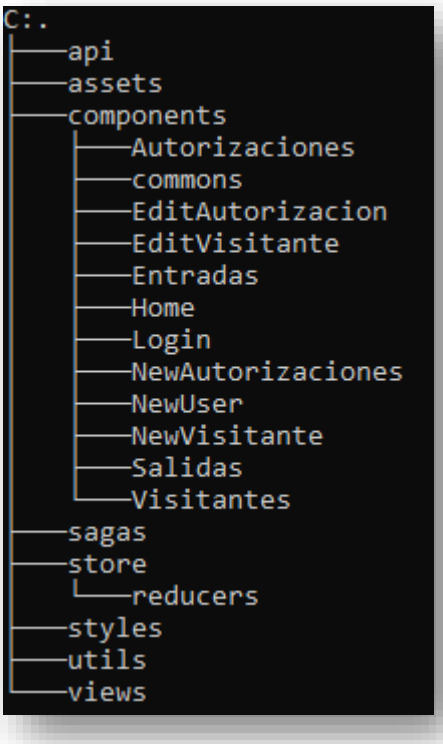


Ilustración 3 - Estructura del proyecto React

4.2.1.1 Archivos raíz

En la raíz del proyecto nos encontramos con los siguientes archivos:

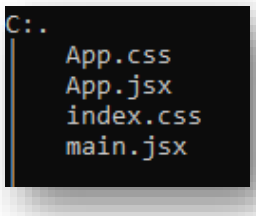


Ilustración 4 - Archivos raíz

- **App.css/index.css**: Archivos de estilo CSS para toda la aplicación.
- **main.jsx:** Este archivo define el punto de entrada principal de la aplicación y se configura el enrutador para manejar la navegación entre diferentes vistas utilizando React Router. Además, se monta el componente App dentro del enrutador para que pueda acceder a las rutas definidas de la aplicación.
- **App.jsx**: Este archivo es el componente principal de la aplicación. Aquí se configuran y se integran aspectos clave como el almacenamiento del estado global con Redux y el enrutamiento de las vistas principales, que son la vista del menú superior que siempre será visible por el usuario y la vista Main a partir de la cual se mostrarán el resto de vistas de la aplicación. También se configura la gestión de notificaciones con la librería Toast.

```
export const store = configureSagaStore();

class App extends Component {
  render() {
    return(
    <Provider store={store}>
      <div className="container">
        <ControlAccesoMenu/>
        <Main/>
        <ToastContainer
          position="top-center"
          autoclose={5000}
          hideProgresBar={false}
          newestOnTop
          closeOnClick
          rtl={false}
          pauseOnVisibilityChange
          draggablepauseOnHover
        />
      </div>
    </Provider>
    )
  }

}

export default App;
```

Ilustración 5 - App.jsx

4.2.1.2 Directorio `api`

Contiene módulos que gestionan las interacciones con los controladores del backend.

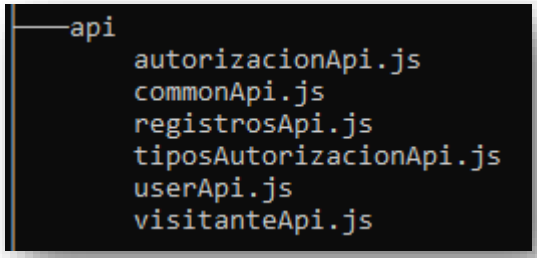


Ilustración 6 - Directorio `api`

- **autorizacionApi.js**: Manejador API para las autorizaciones.
- **commonApi.js**: Funciones comunes para las llamadas.
- **registrosApi.js**: Manejador API para los registros de entrada y salida.
- **tiposAutorizacionApi.js**: Manejador API para los tipos de autorización.
- **userApi.js**: Manejador API para la gestión de usuarios.
- **visitanteApi.js**: Manejador API para la gestión de visitantes.

4.2.1.3 Directorio `assets`

Directorio que tienen todos los proyectos por defecto que almacena recursos estáticos y que no ha sido usado en esta aplicación, aunque podría contener imágenes o documentos con los que el usuario podría interactuar si se requiriese.

4.2.1.4 Directorio `components`

Contiene todos los componentes de la aplicación que se pueden usar en diferentes vistas.

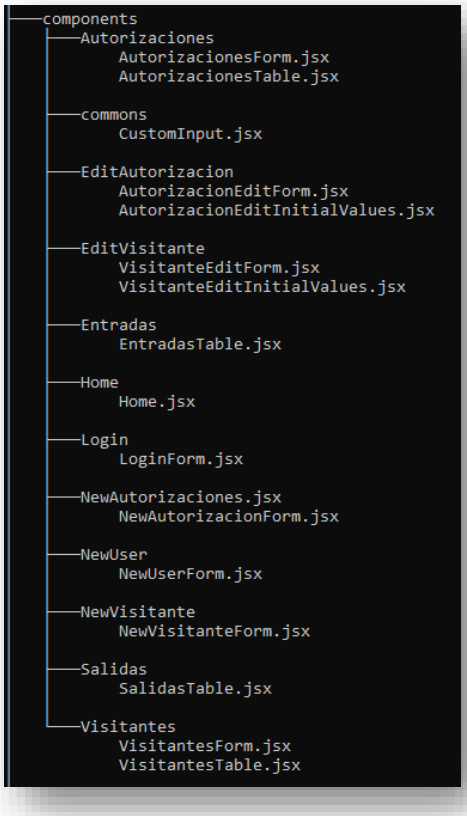


Ilustración 7 - Directorio `components`

La pantalla de inicio de sesión consta de un único componente LoginForm.jsx que muestra el formulario con los campos de nombre de usuario y contraseña. Es la primera vista que se observa al arrancar la aplicación al no estar todavía autenticados. El componente NewUserForm.jsx es un componente muy parecido que se usa para el registro de un nuevo usuario con todos los campos necesarios para cumplimentar.

El componente Home.jsx es un componente sencillo que muestra un mensaje de bienvenido al acceder a la aplicación después del login o del registro de un nuevo usuario.

Como se puede observar, las vistas de visitantes y autorizaciones contienen dos componentes, uno para el diseño del formulario de búsqueda, donde aparecen esos campos opcionales que podemos utilizar para realizar un filtrado a la hora de la búsqueda de esos registros, y otro componente tabla, que diseña la parte de la pantalla donde se visualiza la pantalla donde se vuelcan los datos recuperados de la búsqueda.

Las vistas de edición de visitantes y autorizaciones constan de un componente que diseña el formulario con los campos a editar del recurso y un componente previo que recupera los valores iniciales del recurso que se quiere editar para mostrarlos por defecto en el formulario.

Las vistas de los registros de entrada y salida solo tienen un componente que diseña la tabla donde se vuelcan los datos correspondientes.

4.2.1.5 Directorio `views`

Directorio que contiene los componentes principales de todas las vistas de la aplicación.

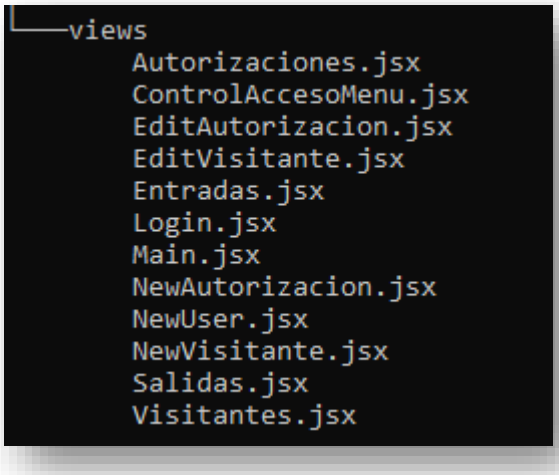
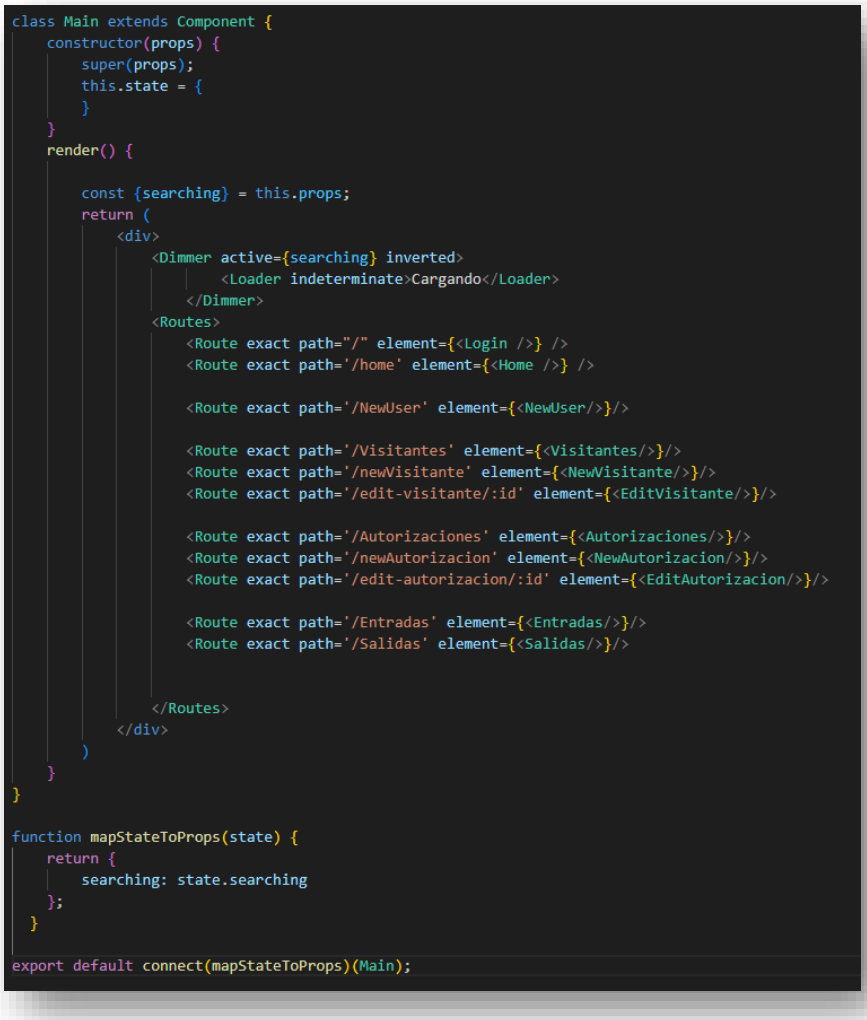


Ilustración 8 - Directorio `views`

- **ControlAccesoMenu.jsx**: este componente actúa de forma independiente al resto de las vistas de la aplicación, tal y como se ha definido en App.jsx, para aparecer siempre en la parte superior independientemente de la vista de la aplicación a la que accedamos, a excepción de cuando no estamos autenticados y nos encontramos en el formulario de inicio de sesión o de registro de un nuevo usuario, donde este componente será ocultado.
- **Main.jsx**: define el componente Main, que actúa como el contenedor principal de las diferentes vistas de la aplicación. Este componente utiliza React Router para definir las rutas y asociarlas con los componentes correspondientes.



```
class Main extends Component {
    constructor(props) {
        super(props);
        this.state = {
        }
    }
    render() {

        const {searching} = this.props;
        return (
            <div>
                <Dimmer active={searching} inverted>
                        <Loader indeterminate>Cargando</Loader>
                    </Dimmer>
                <Routes>
                    <Route exact path="/" element={<Login />} />
                    <Route exact path='/home' element={<Home />} />

                    <Route exact path='/NewUser' element={<NewUser/>}/>

                    <Route exact path='/Visitantes' element={<Visitantes/>}/>
                    <Route exact path='/newVisitante' element={<NewVisitante/>}/>
                    <Route exact path='/edit-visitante/:id' element={<EditVisitante/>}/>

                    <Route exact path='/Autorizaciones' element={<Autorizaciones/>}/>
                    <Route exact path='/newAutorizacion' element={<NewAutorizacion/>}/>
                    <Route exact path='/edit-autorizacion/:id' element={<EditAutorizacion/>}/>

                    <Route exact path='/Entradas' element={<Entradas/>}/>
                    <Route exact path='/Salidas' element={<Salidas/>}/>


                </Routes>
            </div>
        )
    }
}

function mapStateToProps(state) {
    return {
        searching: state.searching
    };
  }

export default connect(mapStateToProps)(Main);
```

Ilustración 9 - Main.jsx

- **Login.jsx**: vista de inicio de sesión, que usa el componente LoginForm.
- **NewUser.jsx**: vista de registro de un nuevo usuario, que usa el componente NewUserForm
- **Visitantes.jsx**: vista del módulo de visitantes, que usa los componentes VisitantesForm y VisitantesTable.
- **NewVisitante.jsx**: vista de la pantalla de creación de visitantes, que usa el componente NewVisitanteForm.
- **EditVisitante.jsx**: vista de la pantalla de edición de un visitante, que usa el componente VisitanteEditInitialValues
- **Autorizaciones.jsx**: vista del módulo de autorizaciones, que usa los componentes de AutorizacionesForm y AutorizacionesTable
- **NewAutorizacion.jsx**: vista de la pantalla de creación de autorizaciones, que usa el componente newAutorizacionForm.
- **EditAutorizacion.jsx**: vista de la pantalla de edición de una autorización, que usa el componente AutorizacionEditInitialValues
- **Entradas.jsx**: vista del módulo de entradas, que usa el componente EntradasTable.
- **Salidas.jsx**: vista del módulo de salidas, que usa el componente SalidasTable

4.2.1.6 Directorio `store`

Directorio en el que se almacena la configuración del estado global de la aplicación.

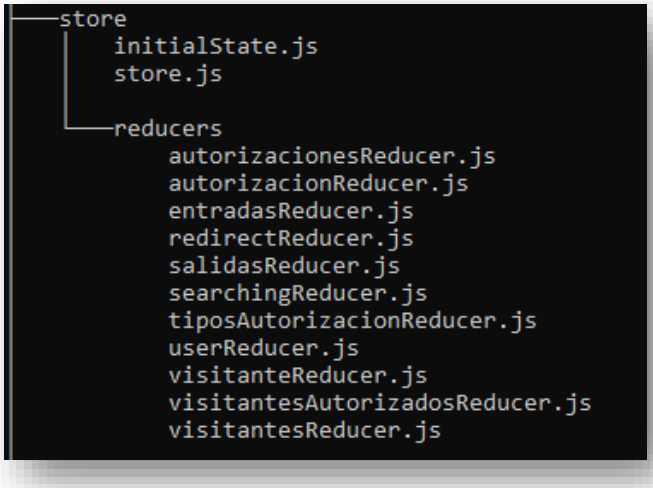


Ilustración 10 - Directorio `store`

- **initialState.js**: Archivo en el que se inicializan todas las variables globales que se van a usar en la aplicación para almacenar los datos que obtenemos de base de datos.
- **store.js**: es el archivo en el que se crea y configura la store de Redux para la aplicación, integrando Redux-Saga para manejar efectos secundarios como la carga de datos desde los reducers en las variables globales.
- La carpeta reducers contiene todas las funcionen reducer de la aplicación, que determinan cómo se actualiza el estado de la aplicación en respuesta a una acción, es decir, toma el estado actual de una variable global y una acción como argumentos y devuelve un nuevo estado, que puede ser, por ejemplo, la población de la variable visitantes con todos los recursos de tipo visitante que han sido devueltos por la capa de aplicación.
    - userReducer: Reducer para la gestión de usuarios.
    - searchingReducer: Reducer para el estado de búsqueda, que maneja la activación o desactivación del elemento “cargando” que aparece mientras la aplicación procesa una petición.
    - visitantesReducer: Reducer para la lista de visitantes.
    - redirectReducer: Reducer para manejar redirecciones.
    - visitanteReducer: Reducer para un visitante específico.
    - tiposAutorizacionReducer: Reducer para los tipos de autorización.
    - autorizacionesReducer: Reducer para la lista de autorizaciones.
    - autorizacionReducer: Reducer para una autorización específica.
    - visitantesAutorizadosReducer: Reducer para los visitantes asociados a una autorización.
    - entradasReducer: Reducer para las entradas.
    - salidasReducer: Reducer para las salidas.

4.2.1.7 Directorio `styles`

Directorio que contiene definiciones de estilos personalizados.

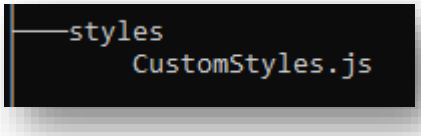


Ilustración 11 - Directorio `styles`

- **CustomStyles.js**: define el estilo de elementos utilizados en los mensajes de confirmación que se usan a la hora de cerrar sesión, borrar autorizaciones y visitantes o confirmar el registro de entrada o salida de un visitante.

4.2.1.8 Directorio `utils`

Contiene funciones utilitarias.

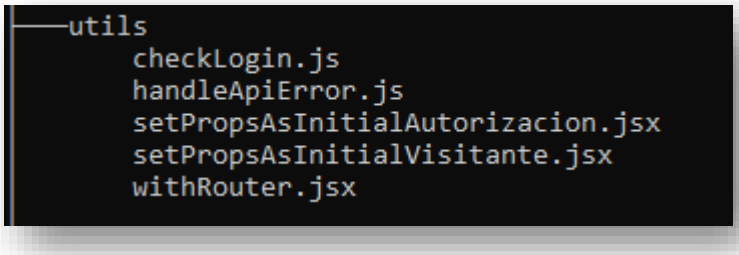


Ilustración 12 - Directorio `utils`

- **checkLogin.js**: Verifica el estado de la sesión del usuario.
- **handleApiError.js**: Maneja errores de las llamadas a la API.
- **setPropsAsInitialAutorizacion.jsx**: Establece propiedades iniciales para el componente de edición de una autorización.
- **setPropsAsInitialVisitante.jsx**: Establece propiedades iniciales para el componente de edición de un visitante.
- **withRouter.jsx**: define una función de orden superior (Higher-Order Component, HOC) que usa para obtener información de las propiedades de enrutamiento proporcionadas la librería `React-router-dom`.

4.2.1.9 Directorio `sagas`

Directorio que contiene las funciones saga, que es una función que escucha las acciones lanzadas por el store de Redux y construye una llamada API REST en función de lo obtenido. Por ejemplo, cuando visitantesSaga escuche el `store.dispatch({type: 'GET_ALL_VISITANTES'})` construira la llamda que obtiene la lista de todos los visitantes de base de datos.

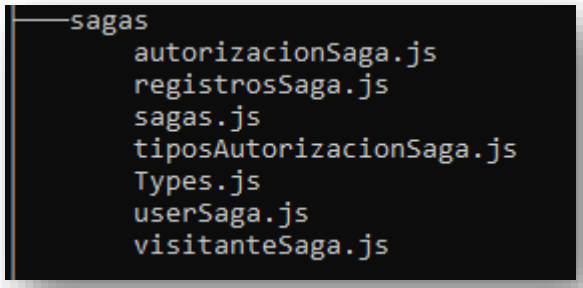


Ilustración 13 - Directorio `sagas`

- **Types.js**: Define los tipos de acciones para Redux.
- **sagas.js**: Punto de entrada principal para exportar todas las sagas.
- **autorizacionSaga.js**: Saga relacionada con la gestión de las autorizaciones.
- **registrosSaga.js**: Saga relacionada con la gestión de los registros.
- **tiposAutorizacionSaga.js**: Saga relacionadas con la gestión de los tipos de autorización.
- **userSaga.js**: Sagas relacionadas con la gestión de usuarios.
- **visitanteSaga.js**: Sagas relacionadas con la gestión de visitantes.

4.3 Diseño de la capa de aplicación

La capa de aplicación es la encargada de recoger las peticiones HTTP lanzadas por la capa de presentación, actuar en consecuencia, comunicándose con la base de datos a través de repositorios JPA y devolver el resultado al Frontend para que pueda mostrárselo al usuario.

4.3.1 Estructura del proyecto Spring

A continuación, se muestra la estructura del proyecto que da forma a la capa de aplicación utilizando el framework Spring.

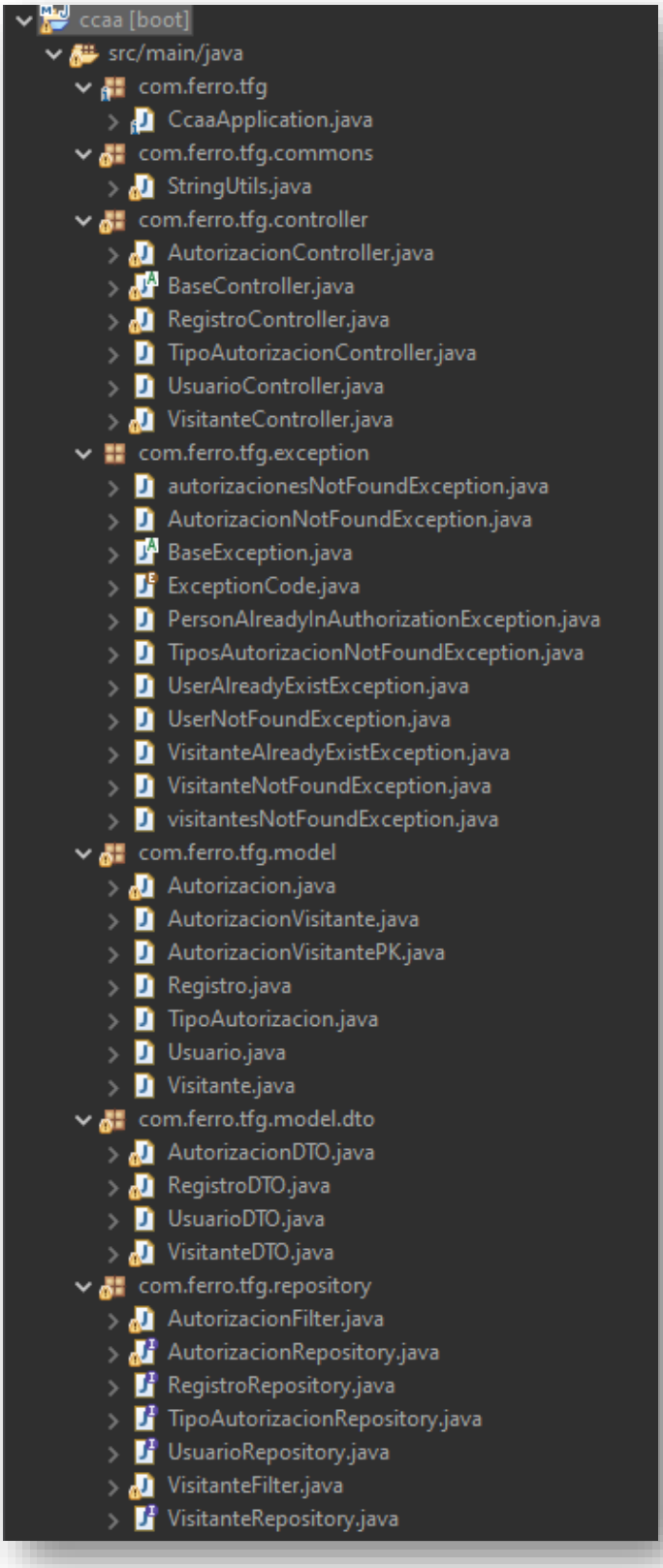


Ilustración 14 - Estructura del proyecto Spring

4.3.1.1 Directorio raíz

En este directorio se encuentra el archivo main (CcaaApplication.java) que arranca la aplicación.

```java
package com.ferro.tfg;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class CcaaApplication {

    public static void main(String[] args) {
        SpringApplication.run(CcaaApplication.class, args);
    }

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                        .allowedOrigins("http://localhost:3000") // Origen permitido
                        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS") // Métodos permitidos
                        .allowedHeaders("*") // Encabezados permitidos
                        .allowCredentials(true); // Permitir credenciales
            }
        };
    }
}
```

Ilustración 15 - CcaaApplication.java

4.3.1.2 Directorio `commons`

En este directorio se alejan las funciones comunes para usar en cualquier parte de la aplicación. En este caso solo existe el StringUtils.java que normaliza variables String para ser usadas en filtrados sobre base de datos.

4.3.1.3 Directorio `controller`

En este directorio se encuentran todos los controladores de la aplicación, que se encargan de manejar las distintas solicitudes HTTP entrantes y definen cómo se deben procesas dichas solicitudes y qué respuestas se deben devolver al cliente.

Existe un BaseController que define una variable por cada repositorio JPA que existe en la aplicación para que todos los controladores hereden de él y puedan hacer uso de estos repositorios. El resto de controladores son los necesarios para manejar solicitudes sobre los diferentes recursos de la aplicación (usuarios, visitantes, autorizaciones, tipos de autorización y registros).

4.3.1.4 Directorio `exception`

Directorio que contiene todas las clases para manejar las excepciones que pueden acontecer durante el uso de la aplicación de manera personalizada, configurando el código de error y mensaje que se devolverá en cada caso.

- **autorizacionesNotFoundException.java**: Excepción que se lanza cuando no se encuentran ninguna autorización que coincida con los criterios de búsqueda solicitados por el usuario.
- **AutorizacionNotFoundException.java**: Excepción lanzada en el caso de no encontrar una autorización que se busca de forma concreta a través de su identificador.
- **PersonAlreadyInAuthorizationException.java**: Excepción lanzada en el caso de intentar asignar un visitante a una autorización en la que ya está asignado.
- **TiposAutorizacionNotFoundException.java**: Excepción lanzada en el caso de no encontrar ningún tipo de autorización definido en la base de datos.
- **UserAlreadyExistException.java**: Excepción lanzada en el caso de intentar registrarnos en la aplicación con un nombre de usuario que ya está en uso.
- **UserNotFoundException.java**: Excepción lanzada en el caso de intentar iniciar sesión con un nombre de usuario no existente.
- **VisitanteAlreadyExistException.java**: Excepción lanzada en el caso de intentar dar de alta un nuevo visitante con un DNI que ya está asociado a otro visitante en la aplicación.
- **VisitanteNotFoundException.java**: Excepción lanzada en el caso de no encontrar un visitante que se busca de forma concreta a través de su identificador o su DNI.
- **visitantesNotFoundException.java**: Excepción lanzada cuando no se encuentra ningún visitante que coincida con los criterios de búsqueda solicitados por el usuario.

4.3.1.5 Directorio `model`

Directorio que contiene un modelo por cada tipo de datos (tabla) almacenado en base de datos.

- **Autorizacion.java**: Objeto para el tratamiento de los recursos autorizaciones con los siguientes atributos de clase correspondientes a sus columnas en base de datos:
    - autorizacionId
    - motivoAutorizacion
    - tipoAutorizacion
    - fechaInicio
    - fechaFin
    - visitantesAutorizados: uso de la etiqueta de persistencia `@JoinTable` para obtener la lista de visitantes con los que se relaciona la correspondiente autorización en la tabla autorización_visitantes.
- **AutorizacionVisitante.java** y **AutorizacionVisitantePK.java**: Objetos para el tratamiento de los recursos referentes a la tabla autorización_visitante, que modela la asignación de una autorización con diferentes visitantes con los atributos:
    - autorizacion
    - visitante

- **Registro.java**: Objeto para el tratamiento de los recursos referentes a la tabla registro con los siguientes atributos de clase correspondientes a sus columnas en base de datos:
    - registroId
    - fechaEntrada
    - fechaSalida
    - visitante
    - autorizacion
- **TipoAutorizacion.java**: Objeto para el tratamiento de los recursos referentes a la tabla tipo_autorizacion con los siguientes atributos de clase correspondientes a sus columnas en base de datos:
    - tipoAutorizacionId
    - nombreTipoAutorizacion
- **Usuario.java**: Objeto para el tratamiento de los recursos referentes a la tabla usuario con los siguientes atributos de clase correspondientes a sus columnas en base de datos:
    - usuarioId
    - nombreUsuario
    - nombre
    - primerApellido
    - segundoApellido
    - password
    - email
- **Visitante.java:** Objeto para el tratamiento de los recursos referentes a la tabla visitante con los siguientes atributos de clase correspondientes a sus columnas en base de datos:
    - visitanteId
    - dni
    - nombre
    - primerApellido
    - segundoApellido
    - fechaNacimiento
    - nombrePadre
    - nombreMadre

4.3.1.6 Directorio `model.DTO`

Directorio en el que se definen los objetos DTO (Data Transfer Object), que son objetos sin lógica similares a los originales que se utilizan para transportar datos. En este desarrollo se ha definido un DTO por cada modelo usado en la transferencia de información entre la capa de presentación y la capa de aplicación

- AutorizacionDTO
- RegistroDTO
- UsuarioDTO
- VisitanteDTO

4.3.1.7 Directorio `repository`

En este directorio se definen las interfaces por cada modelo que extienden de JpaRepository y JpaSpecificationExecutor que permiten realizar diferentes operaciones sobre la base de datos sin necesidad de escribir manualmente las querys.

- **AutorizacionRepository**: repositorio que permite realizar consultas y operaciones sobre la tabla autorización con los siguientes métodos definidos:
  - `List<Autorizacion> findAll(Specification<Autorizacion> specification);`
    - Permite la búsqueda de registros autorización con los filtros definidos en la clase AutorizacionFilter.java con el uso de la librería criteria.
  - `Autorizacion findByAutorizacionId(Integer autorizacionId);`
  - `List<Autorizacion> findByFechaInicioLessThanEqualAndFechaFinGreaterThanEqual( LocalDate hoy, LocalDate hoy2);`
- **RegistroRepository**: repositorio que permite realizar consultas y operaciones sobre la tabla registro con los siguientes métodos definidos:
  - `boolean existsByVisitanteAndAutorizacionAndFechaEntradaIsNotNullAndFechaSalidaIsNull(Visitante visitante,Autorizacion autorizacion);`
  - `List<Registro> findByFechaEntradaIsNotNullAndFechaSalidaIsNull();`
  - `Registro findByRegistroId(Integer id);`
- **TipoAutorizacionRepository**: repositorio que permite realizar consultas y operaciones sobre la tabla tipo_autorizacion con los siguientes métodos definidos:
  - `TipoAutorizacion findByTipoAutorizacionId(Long tipoAutorizacion);`
- **UsuarioRepository**: repositorio que permite realizar consultas y operaciones sobre la tabla usuario con los siguiente métodos definidos:
  - `Usuario findByNombreUsuarioAndPasswordIgnoreCase(String username, String password);`
  - `Usuario findByNombreUsuarioIgnoreCase(String nombreUsuario);`
- **VisitanteRepository**: repositorio que permite realizar consultar y operaciones sobre la tabla usuario con los siguiente métodos definidos:
  - `List<Visitante> findAll(Specification<Visitante> specification);`
    - Permite la búsqueda de registro visitante con los filtros definidos en la clase VisitanteFilter.java con el uso de la librería criteria.

4.3.2 Documentación de la API y endpoints

4.3.2.1 getUser

Endpoint: GET api/user/login

Descripción: permite verificar si la información de autentiación es correcta para dar acceso al sistema.

Parámetros:

Username: nombre de usuario.

Password: contraseña.

Excepciones:

UserNotFoundException

Respuestas:

Retorna un objeto Usuario con la información completa del usuario

200 OK

404 Not Found

4.3.2.2 newUser

Endpoint: POST api/user/new

Descripción: permite registrar un nuevo usuario.

Parámetros:

userDTO: DTO con la información del usuario a registrar.

Excepciones:

UserAlreadyExistsException

Respuestas:

Retorna un objeto Usuario con la información completa del usuario registrado

200 OK

400 Bad Request

4.3.2.3 getVisitante

Endpoint: GET api/visitantes/all

Descripción: devuelve una lista de visitantes con todos los visitantes de la aplicación.

Excepciones:

VisitantesNotFoundException

Respuestas:

Retorna una lista de objetos Visitante

200 OK

404 Not Found

4.3.2.4 buscarVisitantes

Endpoint: GET api/visitantes/filter

Descripción: permite buscar visitantes siguiendo unos criterios específicos.

Parámetros:

dni (opcional): DNI del visitante.

name(opcional): nombre del visitante.

surname(opcional): apellido del visitante.

Surname2(opcional): segundo apellido del visitante.

fatherName(opcional): nombre del padre del visitante.

motherName(opcional): nombre de la madre del visitante

Excepciones:

VisitantesNotFoundException

Respuestas:

Retorna una lista de objetos Visitante

200 OK

404 Not Found

4.3.2.5 newVisitante

Endpoint: POST api/visitantes/new

Descripción: devuelve una lista de visitantes con todos los visitantes de la aplicación.

Parámetros:

VisitanteDTO: DTO con la información del visitante a registrar.

Excepciones:

VisitanteAlreadyExistsException

Respuestas:

Retorna un objeto Visitante con la información del visitante creado

200 OK

400 Bad Request

4.3.2.6 getVisitante

Endpoint: GET api/visitantes/get

Descripción: busca un visitante en concreto.

Parámetros:

id: id del visitante a buscar

Excepciones:

VisitanteNotFoundException

Respuestas:

Retorna un objeto Visitante con la información del visitante que se buscaba.

200 OK

404 Not Found

4.3.2.7 editVisitante

Endpoint: PUT api/visitantes/edit/{visitanteId}

Descripción: edita un visitante.

Parámetros:

id: id del visitante a editar.

VisitanteDTO: DTO con la información del visitante modificada.

Excepciones:

VisitanteNotFoundException

Respuestas:

Retorna un objeto Visitante con la información del visitante actualizada.

200 OK

404 Not Found

4.3.2.8 deleteVisitante

Endpoint: DELET api/visitantes/delete/{Id}

Descripción: elimina un visitante.

Parámetros:

id: id del visitante a eliminar

Excepciones:

VisitanteNotFoundException

Respuestas:

Retorna un objeto Visitante con la información del visitante que se ha borrado.

200 OK

404 Not Found

4.3.2.9 getTiposAutorizacion

Endpoint: GET api/tipoAutorizacion/all

Descripción: busca todos los tipos de autorización.

Excepciones:

TiposAutorizacionNotFoundException

Respuestas:

Retorna una lista de objetos TipoAutorizacion.

200 OK

404 Not Found

4.3.2.10 getAutorizaciones

Endpoint: GET api/autorizaciones/all

Descripción: obtiene todas las autorizaciones.

Excepciones:

AutorizacionesNotFoundException

Respuestas:

Retorna una lista de objetos Autorizacion.

200 OK

404 Not Found

4.3.2.11 buscarAutorizaciones

Endpoint: GET api/autorizaciones/filter

Descripción: busca autorizaciones según los criterios de búsqueda especificados.

Parámetros:

motivoAutorizacion: motivo de la autorización

tipoAutorizacion: id del tipo de autorización

Excepciones:

AutorizacionesNotFoundException

Respuestas:

Retorna una lista de objetos Autorizacion.

200 OK

404 Not Found

4.3.2.12 newAutorizacion

Endpoint: POST api/autorizaciones/new

Descripción: registra una nueva autorización.

Parámetros:

autorizacionDTO: DTO de la autorización a crear.

Respuestas:

Retorna un objeto Autorizacion con la información de la autorización creada.

200 OK

4.3.2.13 editAutorizacion

Endpoint: PUT api/autorizaciones/edit/{autorizacionId}

Descripción: edita una autorización

Parámetros:

autorizacionId: id de la autorización a editar

autorizacionDTO: DTO con la información de la autorización editada.

Excepciones:

AutorizacionNotFoundException

Respuestas:

Retorna un objeto Autorizacion con la información de la autorización actualizada.

200 OK

404 Not Found

4.3.2.14 deleteAutorizacion

Endpoint: DELETE api/autorizaciones/delete/{Id}

Descripción: elimina una autorización.

Parámetros:

Id: id de la autorización a eliminar

Excepciones:

AutorizacionNotFoundException

Respuestas:

Retorna objeto Autorizacion con la información de la autorización eliminada.

200 OK

404 Not Found

4.3.2.15 asignarVisitante

Endpoint:

PUT api/autorizaciones/{autorizacionId}/visitantesAutorizados/{visitanteDNI}

Descripción: asigna un visitante a una autorización.

Parámetros:

autorizacionId: id de la autorización a la que se va a asignar.

visitanteDNI: DNI del visitante a asignar.

Excepciones:

PersonAlreadyInAuthorizationException

Respuestas:

Retorna un objeto Visitante con la información del visitante asignado.

200 OK

400 Bad Request

4.3.2.16 visitantesAutorizados

Endpoint: GET api/autorizaciones/{autorizacionId}/visitantesAutorizados

Descripción: devuelve los visitantes asignados a una autorización.

Parámetros:

autorizacionId: id de la autorización de la que se quiere obtener sus visitantes asignados.

Excepciones:

AutorizacionNotFoundException

Respuestas:

Retorna una lista de objetos Visitante.

200 OK

404 Not Found

4.3.2.17 desasignarVisitante

Endpoint: DELETE api/autorizaciones/{autorizacionId}/visitantesAutorizados/{visitanteId}

Descripción: desasigna un visitante de una autorización.

Parámetros:

autorizacionId: id de la autorización de que se va a desasignar.

visitanteId: id del visitante a desasignar.

Respuestas:

Retorna un objeto Visitante.

200 OK

4.3.2.18 obtenerRegistrosActivosHoy

Endpoint: GET api/registros/activosHoy

Descripción: devuelve los visitantes asignados a una autorización valida en el día de hoy y que no hayan pasado dentro del control.

Respuestas:

Retorna una lista de objetos Registro.

200 OK

4.3.2.19 registrarEntrada

Endpoint: POST api/registros/new

Descripción: registra la entrada de un visitante.

Parámetros:

registroDTO: DTO de tipo registro con la información del visitante y la autorización a la que pertenece.

Respuestas:

Retorna un objeto de tipo registro con la fecha de entrada cumplimentada.

200 OK

4.3.2.20 obtenerRegistrosSalida

Endpoint: GET api/registros/out

Descripción: obtiene los visitantes que están dentro del control esperando a que se registre su salida.

Respuestas:

Retorna una lista de objetos Registro que puede estar vacía.

200 OK

4.3.2.21

Endpoint: POST api/registros/out

Descripción: registra la salida de un visitante.

Parámetros:

id: id del registro del que se quiere registrar su salida.

Respuestas:

Retorna un objeto de tipo registro con la fecha de salida cumplimentada.

200 OK

4.4 Diseño de la capa de datos

Para el diseño de la base de datos se ha creado un esquema `ccaa` con diferentes tablas que forman el modelo de datos.

4.4.1 Tabla usuario

Tabla que almacena los registros de los diferentes usuarios dados de alta en la aplicación. La tabla consta de las columnas ID_USUARIO, USERNAME, CONTRASEÑA, NOMBRE, APELLIDO1, APELLIDO2 y CORREO.

DDL for ccaa.usuario

```
1  CREATE TABLE `usuario` (
2    `ID_USUARIO` int NOT NULL AUTO_INCREMENT,
3    `USERNAME` varchar(255) NOT NULL,
4    `CONTRASEÑA` varchar(255) NOT NULL,
5    `NOMBRE` varchar(255) DEFAULT NULL,
6    `APELLIDO1` varchar(255) DEFAULT NULL,
7    `APELLIDO2` varchar(255) DEFAULT NULL,
8    `CORREO` varchar(255) DEFAULT NULL,
9    PRIMARY KEY (`ID_USUARIO`)
10 ) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Ilustración 16 - DDL tabla `usuario`

4.4.2 Tabla visitante

Tabla que almacena los registros de los diferentes visitantes de la aplicación. La tabla consta de las columnas ID_VISITANTE, NOMBRE, APELLIDO1, APELLIDO2, DNI, FECHA_NACIMIENTO, NOMBRE_PADRE y NOMBRE_MADRE

DDL for ccaa.visitante

```
1  CREATE TABLE `visitante` (
2    `ID_VISITANTE` int NOT NULL AUTO_INCREMENT,
3    `NOMBRE` varchar(255) NOT NULL,
4    `APELLIDO1` varchar(255) DEFAULT NULL,
5    `APELLIDO2` varchar(255) DEFAULT NULL,
6    `DNI` varchar(255) NOT NULL,
7    `FECHA_NACIMIENTO` date DEFAULT NULL,
8    `NOMBRE_PADRE` varchar(255) DEFAULT NULL,
9    `NOMBRE_MADRE` varchar(255) DEFAULT NULL,
10   PRIMARY KEY (`ID_VISITANTE`)
11 ) ENGINE=InnoDB AUTO_INCREMENT=111 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Ilustración 17 - DDL tabla `visitante`

4.4.3 Tabla tipo_autorizacion

Tabla que almacena los tipos de autorización con los que se pueden clasificar las autorizaciones. En nuestro caso se han añadido unos tipos de autorización predefinidas ya que la aplicación aun no cuenta con una funcionalidad que permita actuar sobre estos registros. La tabla consta de las columnas ID_TIPO_AUTORIZACION y TIPO_AUTORIZACION.

DDL for ccaa.tipo_autorizacion

```
1  CREATE TABLE `tipo_autorizacion` (
2    `ID_TIPO_AUTORIZACION` int NOT NULL,
3    `TIPO_AUTORIZACION` varchar(255) NOT NULL,
4    PRIMARY KEY (`ID_TIPO_AUTORIZACION`)
5  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Ilustración 18 - DDL tabla `tipo_autorizacion`

```
19 • INSERT INTO TIPO_AUTORIZACION (ID_TIPO_AUTORIZACION, TIPO_AUTORIZACION) VALUES (0, 'INTERIOR');
20 • INSERT INTO TIPO_AUTORIZACION (ID_TIPO_AUTORIZACION, TIPO_AUTORIZACION) VALUES (1, 'MANTENIMIENTO');
21 • INSERT INTO TIPO_AUTORIZACION (ID_TIPO_AUTORIZACION, TIPO_AUTORIZACION) VALUES (2, 'OFICINAS');
22 • INSERT INTO TIPO_AUTORIZACION (ID_TIPO_AUTORIZACION, TIPO_AUTORIZACION) VALUES (3, 'SEGURIDAD');
23 • INSERT INTO TIPO_AUTORIZACION (ID_TIPO_AUTORIZACION, TIPO_AUTORIZACION) VALUES (4, 'OTRO');
```

Ilustración 19 - Registros tabla `tipo_autorizacion`

4.4.4 Tabla autorizacion

Tabla que almacena las autorizaciones dadas de alta en la aplicación. La tabla consta de las columnas ID_AUTORIZACION, MOTIVO_AUTORIZACION, ID_TIPO_AUTORIZACION, FECHA_INICIO y FECHA_FIN.

DDL for ccaa.autorizacion

```
CREATE TABLE `autorizacion` (
  `ID_AUTORIZACION` int NOT NULL AUTO_INCREMENT,
  `MOTIVO_AUTORIZACION` varchar(255) NOT NULL,
  `ID_TIPO_AUTORIZACION` int DEFAULT NULL,
  `FECHA_INICIO` date NOT NULL,
  `FECHA_FIN` date NOT NULL,
  PRIMARY KEY (`ID_AUTORIZACION`),
  KEY `ID_TIPO_AUTORIZACION` (`ID_TIPO_AUTORIZACION`),
  CONSTRAINT `autorizacion_ibfk_1` FOREIGN KEY (`ID_TIPO_AUTORIZACION`) REFERENCES `tipo_autorizacion` (`ID_TIPO_AUTORIZACION`)
) ENGINE=InnoDB AUTO_INCREMENT=140 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Ilustración 20 - DDL tabla `autorizacion`

4.4.5 Tabla autorizacion_visitante

Tabla que almacena las relaciones de los visitantes con las autorizaciones a las que están asignados. La tabla consta de las columnas ID_AUTORIZACION y ID_VISITANTE.

DDL for ccaa.autorizacion_visitante

```
CREATE TABLE `autorizacion_visitante` (
  `ID_AUTORIZACION` int NOT NULL,
  `ID_VISITANTE` int NOT NULL,
  PRIMARY KEY (`ID_AUTORIZACION`,`ID_VISITANTE`),
  KEY `ID_VISITANTE` (`ID_VISITANTE`),
  CONSTRAINT `FK_AUT_AUT_VISITANTE` FOREIGN KEY (`ID_AUTORIZACION`) REFERENCES `autorizacion` (`ID_AUTORIZACION`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Ilustración 21 - DDL tabla `autorizacion_visitante`

4.4.6 Tabla registro

Tabla que almacena los registros que se han realizado a través de la aplicación, indicando la fecha de entrada con la hora a la que el visitante ha entrado y la fecha de salida con la hora a la que el visitante ha salido, en caso de haberlo hecho. La tabla consta de las columnas ID_REGISTRO, FECHA_ENTRADA, FECHA_SALIDA, ID_VISITANTE, ID_AUTORIZACION.

DDL for ccaa.registro

```
1  CREATE TABLE `registro` (
2    `ID_REGISTRO` int NOT NULL AUTO_INCREMENT,
3    `FECHA_ENTRADA` timestamp NULL DEFAULT NULL,
4    `FECHA_SALIDA` timestamp NULL DEFAULT NULL,
5    `ID_VISITANTE` int NOT NULL,
6    `ID_AUTORIZACION` int NOT NULL,
7    PRIMARY KEY (`ID_REGISTRO`),
8    KEY `FK_REGISTRO_AUTORIZACION` (`ID_AUTORIZACION`),
9    KEY `FK_REGISTRO_VISITANTE` (`ID_VISITANTE`),
10   CONSTRAINT `FK_REGISTRO_AUTORIZACION` FOREIGN KEY (`ID_AUTORIZACION`) REFERENCES `autorizacion` (`ID_AUTORIZACION`),
11   CONSTRAINT `FK_REGISTRO_VISITANTE` FOREIGN KEY (`ID_VISITANTE`) REFERENCES `visitante` (`ID_VISITANTE`)
12 ) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Ilustración 22 - DDL tabla `registro`

4.4.7 Diagrama Entidad-Relación

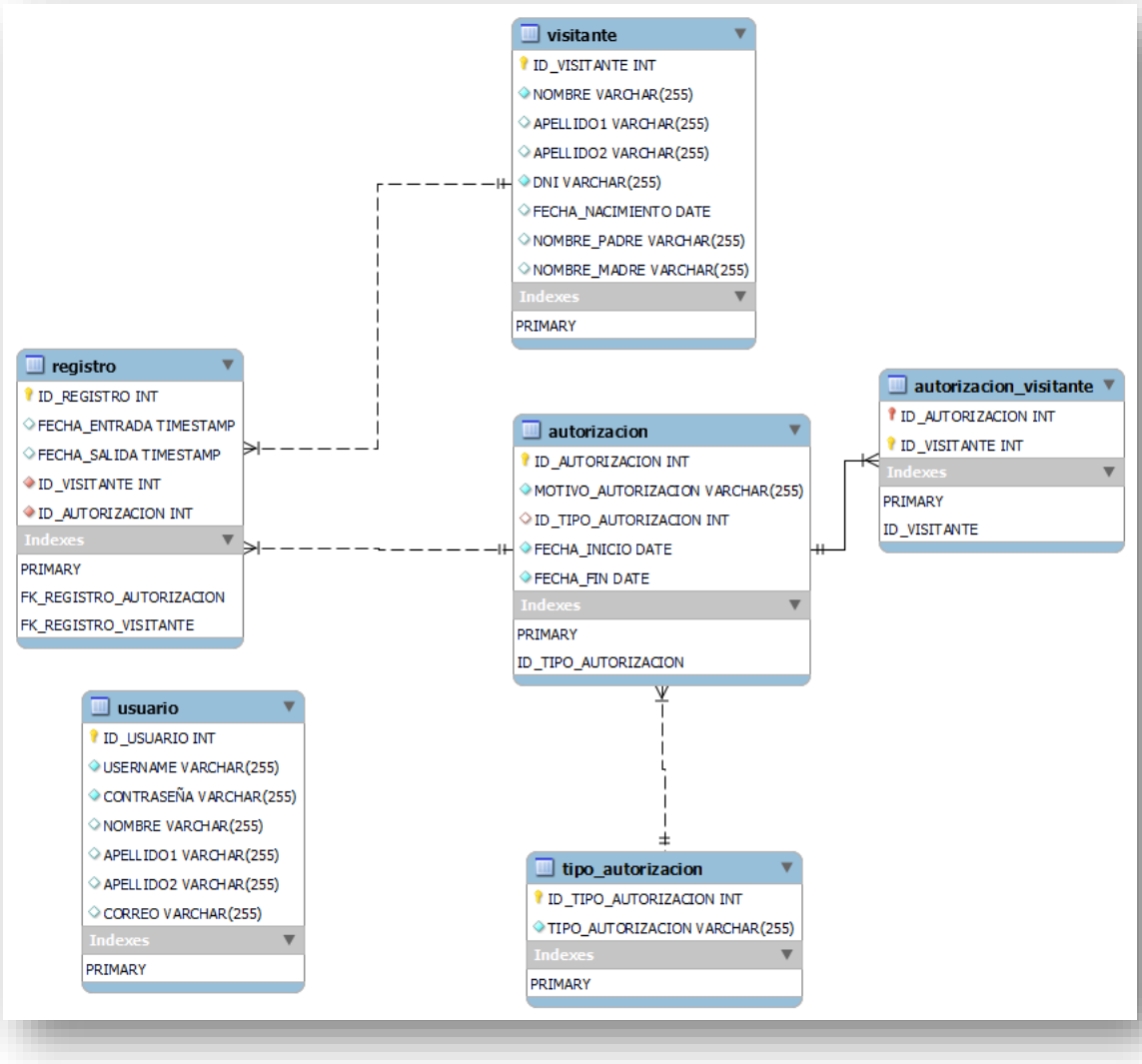


Ilustración 23 - Diagrama EER

5 Implementación

En esta sección se expone la forma en la que la aplicación se ha implementado, prestando especial atención en el desarrollo de la capa de aplicación, que hace de núcleo de la aplicación, desarrollado en Java con Spring Boot y la implementación del Frontend con React.

5.1 Desarrollo de la capa de aplicación

La capa de aplicación se ha desarrollado con Java y Spring Boot, haciendo uso de diferentes librerías para cumplir con los requisitos predefinidos.

5.1.1 Configuración inicial del proyecto

En primer lugar, al montar el entorno en el que se iba a desarrollar la aplicación, se ha decidido de qué librerías se iba a hacer uso para la implementación y se ha optado por el uso de Maven para importar las librerías desde un archivo pom.xml, que tiene el siguiente contenido:

Ilustración 24 - pom.xml

Como se puede observar hay información básica sobre la configuración de nuestro proyecto, como el proyecto del que hereda configuraciones, que en este caso es por defecto el `spring-boot-starter-parent`, que proporciona configuraciones y dependencias comunas para aplicaciones Spring Boot. Tambien muestra información como el identificador del proyecto, estructurado como un nombre de dominio invertido (com.ferro), el nombre del proyecto (ccaa) y la versión de java que va a utilizar (java 17).

Respecto a las dependencias usadas en el proyecto:

- **spring-boot-starter-web**: proporciona dependencias necesarias para construir aplicaciones web utilizando Spring Boot.
- **spring-web**: incluye funcionalidades del módulo Spring Web.
- **spring-boot-starter-data-jpa**: dependencia que permite trabajar con JPA (Java Persistence API).
- **mssql-jdbc y mysql-connector-j**: dependencias para conectar con bases de datos MySQL
- **lombok**: biblioteca utilizada para reducir el código boilerplate con anotaciones y que nos ayuda a resolver operaciones como getters, setters o toString().
- **ojdcb8**: driver JDBC para conectar con bases de datos Oracle (no se ha usado en este caso).

Despues de las dependencias también se define en el pom el plugin spring-boot-maven-plugin, que es una herramienta especifica de Maven proporcionada por Spring Boot para construir y empaquetar aplicaciones Spring Boot

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <image>
                    <builder>paketobuildpacks/builder-jammy-base:latest</builder>
                </image>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Ilustración 25 - Plugin maven para Spring Boot

Una vez definida la configuración y dependencias del proyecto en el pom.xml se configura las propiedades principales del proyecto, sobre todo de cara a la conexión con la base de datos. Esto se realiza en el archivo application.properties.

```
spring.datasource.url=jdbc:mysql://localhost:3306/ccaa
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=validate
```

Ilustración 26 - application.properties

En este archivo se definen parámetros como la URL de conexión JDBC a la base de datos MySQL, las credenciales para autenticarse en la base de datos y el controlador JDBC que se utilizará para conectarse a la base de datos.

También se definen parámetros correspondientes al uso de JPA como el dialecto de Hibernate para generar consultas SQL o la estrategia de generación de DDL, que en este ejemplo usa “validate”, lo que significa que Hibernate validará el esquema de la base de datos existente con los modelos de entidades JPA cada vez que se arranca la aplicación.

Por último, se ha realizado una configuración personalizada de CORS, que es un mecanismo de seguridad implementado por los navegadores para restringir solicitudes HTTP en diferentes casos.

```
package com.ferro.tfg;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class CcaaApplication {

    public static void main(String[] args) {
        SpringApplication.run(CcaaApplication.class, args);
    }

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                        .allowedOrigins("http://localhost:3000") // Origen permitido
                        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS") // Métodos permitidos
                        .allowedHeaders("*") // Encabezados permitidos
                        .allowCredentials(true); // Permitir credenciales
            }
        };
    }
}
```

Ilustración 27 - corsConfigurer

En este caso, al ser un entorno controlado, se han habilitado las peticiones con origen en el puerto 3000 de localhost, que es donde se aloja el frontend, y se han habilitado las peticiones HTTP necesarias para el funcionamiento de esta aplicación.

5.1.2 Implementación del modelo de las entidades

Para la implementación de los modelos correspondientes a las diferentes entidades se ha realizado un desarrollo muy simple que relaciona cada modelo con la entidad correspondiente en base de datos y los atributos correspondientes a cada columna.

5.1.2.1 Usuario

```java
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@Entity(name = "usuario")
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID_USUARIO", nullable = false, unique = true)
    private Integer usuarioId;

    @Column(name = "USERNAME", nullable = false)
    private String nombreUsuario;

    @Column(name = "NOMBRE", nullable = false)
    private String nombre;

    @Column(name = "APELLIDO1", nullable = false)
    private String primerApellido;

    @Column(name = "APELLIDO2", nullable = false)
    private String segundoApellido;

    @Column(name = "CONTRASEÑA", nullable = false)
    private String password;

    @Column(name = "CORREO", nullable = false)
    private String email;
```

Ilustración 28 - Modelo Usuario.

Para el modelo Usuario se ha utilizado la etiqueta `@Entity` que relaciona el tipo de datos con la tabla de base de datos con el mismo nombre. Para los atributos de clase se ha utilizado las etiquetas `@Column` para relacionarlos con las columnas correspondientes del modelo de datos y la etiqueta `@Id` para el caso del atributo usuarioId para especificar que se trata del atributo identificativo.

El proyecto incluye la especificación de un tipo de datos de transferencia para estas entidades llamado UsuarioDTO, que consta de los mismos atributos y un método que crea un objeto Usuario a partir de este objeto de transferencia.

```java
@Data
@NoArgsConstructor
public class UsuarioDTO {

    private Integer usuarioId;
    private @NonNull String nombreUsuario;
    private @NonNull String nombre;
    private @NonNull String primerApellido;
    private @NonNull String segundoApellido;
    private @NonNull String password;
    private @NonNull String email;


    public Usuario toUsuario() {
        Usuario usuario = new Usuario();
        usuario.setUsuarioId(this.usuarioId);
        usuario.setNombreUsuario(this.nombreUsuario);
        usuario.setNombre(this.nombre);
        usuario.setPrimerApellido(this.primerApellido);
        usuario.setSegundoApellido(this.segundoApellido);
        usuario.setPassword(this.password);
        usuario.setEmail(this.email);
        return usuario;
    }
}
```

Ilustración 29 - UsuarioDTO

5.1.2.2 Visitante

```java
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@Entity(name = "visitante")
public class Visitante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID_VISITANTE", nullable = false, unique = true)
    private Integer visitanteId;

    @Column(name = "DNI", nullable = false)
    private String dni;

    @Column(name = "NOMBRE", nullable = false)
    private String nombre;

    @Column(name = "APELLIDO1", nullable = false)
    private String primerApellido;

    @Column(name = "APELLIDO2", nullable = false)
    private String segundoApellido;

    @Column(name = "FECHA_NACIMIENTO")
    private LocalDate fechaNacimiento;

    @Column(name = "NOMBRE_PADRE", nullable = false)
    private String nombrePadre;

    @Column(name = "NOMBRE_MADRE", nullable = false)
    private String nombreMadre;

    public static Visitante mergeVisitante(Visitante target, Visitante source) {
        target.setDni(source.getDni());
        target.setNombre(source.getNombre());
        target.setPrimerApellido(source.getPrimerApellido());
        target.setSegundoApellido(source.getSegundoApellido());
        target.setNombreMadre(source.getNombreMadre());
        target.setNombrePadre(source.getNombrePadre());
        return target;
    }


}
```

Ilustración 30 - Modelo Visitante

Para el modelo Visitante se ha utilizado la etiqueta `@Entity` que relaciona el tipo de datos con la tabla de base de datos con el mismo nombre. Para los atributos de clase se ha utilizado las etiquetas `@Column` para relacionarlos con las columnas correspondientes del modelo de datos y la etiqueta `@Id` para el caso del atributo visitanteId para especificar que se trata del atributo identificativo.

La clase también consta de un método mergeVisitantes que dados 2 objetos de tipo Visitante los unifica en uno solo, el cual se usa a la hora de realizar una edición sobre un registro de esta entidad para guardar los nuevos valores sin perder los originales en caso de que no se haya solicitado editarlos.

El proyecto incluye la especificación de un tipo de datos de transferencia para estas entidades llamado VisitanteDTO, que consta de los mismos atributos y un método que crea un objeto Visitante a partir de este objeto de transferencia.

```java
@Data
@NoArgsConstructor
public class VisitanteDTO {

    private  Integer visitanteId;
    private  String dni;
    private  String nombre;
    private  String primerApellido;
    private  String segundoApellido;
    private  LocalDate fechaNacimiento;
    private  String nombrePadre;
    private  String nombreMadre;


    public Visitante toVisitante() {
        Visitante visitante = new Visitante();
        visitante.setDni(this.dni);
        visitante.setNombre(this.nombre);
        visitante.setPrimerApellido(this.primerApellido);
        visitante.setSegundoApellido(this.segundoApellido);
        visitante.setFechaNacimiento(this.fechaNacimiento);
        visitante.setNombrePadre(this.nombrePadre);
        visitante.setNombreMadre(this.nombreMadre);
        return visitante;
    }
}
```

Ilustración 31 - VisitanteDTO

5.1.2.3 TipoAutorizacion

```java
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@Entity(name = "tipo_autorizacion")
public class TipoAutorizacion {

    @Id
    @Column(name = "ID_TIPO_AUTORIZACION", nullable = false, unique = true)
    private Integer tipoAutorizacionId;

    @Column(name = "TIPO_AUTORIZACION", nullable = false)
    private String nombreTipoAutorizacion;

}
```

Ilustración 32 - Modelo TipoAutorizacion

Para el modelo TipoAutorizacion se ha utilizado la etiqueta `@Entity` que relaciona el tipo de datos con la tabla de base de datos con el mismo nombre. Para los atributos de clase se ha utilizado las etiquetas `@Column` para relacionarlos con las columnas correspondientes del modelo de datos y la etiqueta `@Id` para el caso del atributo visitanteId para especificar que se trata del atributo identificativo.

5.1.2.4 Autorizacion

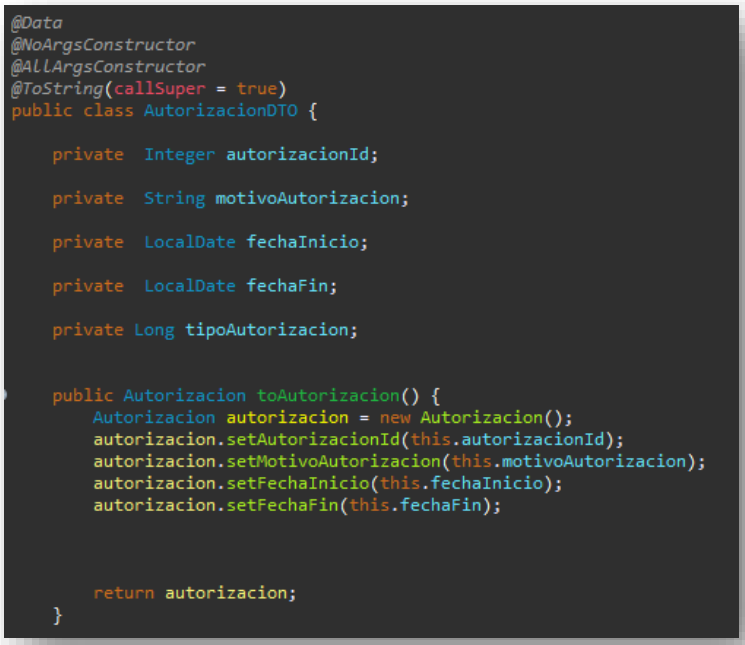
```java
@Entity
@Data
@Table(name = "autorizacion")
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class Autorizacion {


    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID_AUTORIZACION", nullable = false, unique = true)
    private Integer autorizacionId;

    @Column(name = "MOTIVO_AUTORIZACION", nullable = false)
    private String motivoAutorizacion;

    @ManyToOne
    @JoinColumn(name = "ID_TIPO_AUTORIZACION", nullable = false)
    private TipoAutorizacion tipoAutorizacion;

    @Column(name = "FECHA_INICIO", nullable = false)
    private LocalDate fechaInicio;

    @Column(name = "FECHA_FIN", nullable = false)
    private LocalDate fechaFin;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "autorizacion_visitante", joinColumns = @JoinColumn(name = "ID_AUTORIZACION"), inverseJoinColumns = @JoinColumn(name = "ID_VISITANTE"))
    private Set<Visitante> visitantesAutorizados;


    public static Autorizacion mergeAutorizacion(Autorizacion target, Autorizacion source) {
        target.setMotivoAutorizacion(source.getMotivoAutorizacion());
        target.setTipoAutorizacion(source.getTipoAutorizacion());
        target.setFechaInicio(source.getFechaInicio());
        target.setFechaFin(source.getFechaFin());

        return target;
    }
```

Ilustración 33 - Modelo Autorizacion

Para el modelo Autorizacion se ha utilizado la etiqueta `@Entity` que relaciona el tipo de datos con la tabla de base de datos con el mismo nombre. Para los atributos de clase se ha utilizado las etiquetas `@Column` para relacionarlos con las columnas correspondientes del modelo de datos y la etiqueta `@Id` para el caso del atributo visitanteId para especificar que se trata del atributo identificativo. En este caso hay un atributo que no corresponde con ninguna columna y es el atributo visitantesAutorizados, que gracias a la etiqueta `@JoinTable` nos permite obtener la relación de visitantes relacionados con la autorizacion que se hace en la tabla autorización_visitante.

La clase también consta de un método mergeAutorizacion que dados 2 objetos de tipo Autorizacion los unifica en uno solo, el cual se usa a la hora de realizar una edición sobre un registro de esta entidad para guardar los nuevos valores sin perder los originales en caso de que no se haya solicitado editarlos.

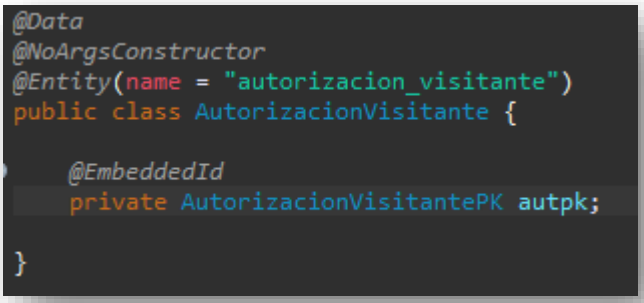
El proyecto incluye la especificación de un tipo de datos de transferencia para estas entidades llamado AutorizacionDTO, que consta de los mismos atributos y un método que crea un objeto Autorizacion a partir de este objeto de transferencia.



```java
@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString(callSuper = true)
public class AutorizacionDTO {

    private  Integer autorizacionId;

    private  String motivoAutorizacion;

    private  LocalDate fechaInicio;

    private  LocalDate fechaFin;

    private Long tipoAutorizacion;


    public Autorizacion toAutorizacion() {
        Autorizacion autorizacion = new Autorizacion();
        autorizacion.setAutorizacionId(this.autorizacionId);
        autorizacion.setMotivoAutorizacion(this.motivoAutorizacion);
        autorizacion.setFechaInicio(this.fechaInicio);
        autorizacion.setFechaFin(this.fechaFin);



        return autorizacion;
    }
```

Ilustración 34 – AutorizacionDTO

5.1.2.5 AutorizacionVisitante



```java
@Data
@NoArgsConstructor
@Entity(name = "autorizacion_visitante")
public class AutorizacionVisitante {

    @EmbeddedId
    private AutorizacionVisitantePK autpk;

}
```

Ilustración 35 - Modelo AutorizacionVisitante

Para el modelo AutorizacionVisitante se ha realizado una implementación algo diferente ya que esta entidad consta de 2 claves primarias (autorizacionId y visitanteId), por lo que se utiliza la etiqueta `@EmbeddedId` y un nuevo tipo de datos llamado AutorizacionVisitantePK que ya contiene la referencia a las dos columnas de base de datos.

```java
@Embeddable
@Data
public class AutorizacionVisitantePK implements Serializable{

    private static final long serialVersionUID = 1L;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "ID_AUTORIZACION", insertable = false, updatable = false, referencedColumnName = "ID_AUTORIZACION")
    private Autorizacion autorizacion;

    @ManyToOne(fetch = FetchType.EAGER)
    @MapsId
    @JoinColumn(name = "ID_VISITANTE", updatable = false)
    private Visitante visitante;


}
```

Ilustración 36 - AutorizacionVisitantePK

5.1.2.6 Registro

```java
@Entity
@Table(name = "registro")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class Registro {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID_REGISTRO", nullable = false, unique = true)
    private Integer registroId;

    @Column(name = "FECHA_ENTRADA")
    private LocalDateTime fechaEntrada;

    @Column(name = "FECHA_SALIDA")
    private LocalDateTime fechaSalida;

    @ManyToOne
    @JoinColumn(name = "ID_VISITANTE", nullable = false)
    private Visitante visitante;

    @ManyToOne
    @JoinColumn(name = "ID_AUTORIZACION", nullable = false)
    private Autorizacion autorizacion;
}
```

Ilustración 37 - Modelo Registro

Para el modelo Registro se ha utilizado la etiqueta `@Entity` que relaciona el tipo de datos con la tabla de base de datos con el mismo nombre. Para los atributos de clase se ha utilizado las etiquetas `@Column` para relacionarlos con las columnas correspondientes del modelo de datos y la etiqueta `@Id` para el caso del atributo visitanteId para especificar que se trata del atributo identificativo.

El proyecto incluye la especificación de un tipo de datos de transferencia para estas entidades llamado RegistroDTO, que consta de los mismos atributos y un método que crea un objeto Registro a partir de este objeto de transferencia.

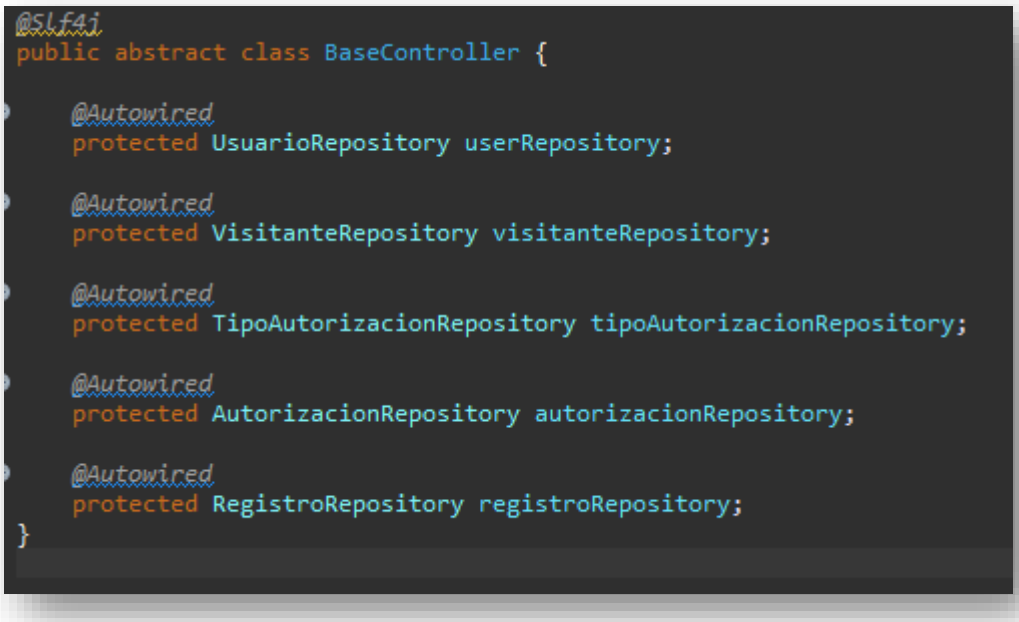
```java
@Data
@NoArgsConstructor
public class RegistroDTO {

    private Integer registroId;
    private LocalDateTime fechaEntrada;
    private LocalDateTime fechaSalida;
    private Visitante visitante;
    private Autorizacion autorizacion;

    public Registro toRegistro() {
        Registro registro = new Registro();
        registro.setFechaEntrada(fechaEntrada);
        registro.setFechaSalida(fechaSalida);
        registro.setVisitante(visitante);
        registro.setAutorizacion(autorizacion);
        return registro;
    }

}
```

Ilustración 38 - RegistroDTO

5.1.3 Implementación de los controladores

Se ha desarrollado un controlador común del que heredarán todos los controladores para poder hacer uso de todos los repositorios disponibles en la aplicación y un controlador por cada modelo existente que reciba llamadas API REST y actúe en consecuencia de estas.

5.1.3.1 BaseController



```java
@Slf4j
public abstract class BaseController {

    @Autowired
    protected UsuarioRepository userRepository;

    @Autowired
    protected VisitanteRepository visitanteRepository;

    @Autowired
    protected TipoAutorizacionRepository tipoAutorizacionRepository;

    @Autowired
    protected AutorizacionRepository autorizacionRepository;

    @Autowired
    protected RegistroRepository registroRepository;
}
```

Ilustración 39 - BaseController

Este controlador es una clase abstracte que sirve como clase base para otros controladores. Al heredar de esta clase base, los demás controladores pueden acceder a los repositorios comunes que están inyectados en el BaseController. Esto ayuda a evitar la repetición de código y a centralizar la gestión de los componentes compartidos.

5.1.3.2 UsuarioController

```java
@RestController
@CrossOrigin
@RequestMapping(value = "api/user")
public class UsuarioController extends BaseController{

    @GetMapping(value = "/login")
    public Usuario getUser(
            @RequestParam(value = "username") String username,
            @RequestParam(value = "password") String password) throws UserNotFoundException {

        Usuario user = userRepository.findByNombreUsuarioAndPasswordIgnoreCase(username, password);

        if(user == null) {
            throw new UserNotFoundException();
        }

        return user;

    }

    @PostMapping(value = "/new")
    public Usuario newUser(
            @RequestBody UsuarioDTO userDTO) throws UserAlreadyExistException {

        Usuario user = userRepository.findByNombreUsuarioIgnoreCase(userDTO.getNombreUsuario());

        if(user != null) {
            throw new UserAlreadyExistException();
        }

        return userRepository.save(userDTO.toUsuario());
    }

}
```

Ilustración 40 - UsuarioController

El UsuarioController es un controlador REST que maneja las solicitudes HTTP relacionadas con la entidad Usuario. Hereda del BaseController, lo que le permite acceder a los repositorios inyectados en la clase base.

La etiqueta `@RestController` indica que esta clase es un controlador donde cada método retorna un objeto de dominio. Spring convierte automáticamente las respuestas en JSON. La etiqueta `@CrossOrigin` permite que las solicitudes de origen cruzado sean manejadas por este controlador, etiqueta necesaria para permitir que el Frontend haga solicitudes a este backend. La etiqueta `@RequestMapping` define la ruta base para todas las solicitudes manejadas por este controlador, en este caso las rutas comenzaran con /api/user.

getUser

Mediante la etiqueta `@GetMapping(value = "/login")` se permite manejar las solicitudes GET a /api/user/login. El método busca un usuario en el repositorio que coincida con el nombre de usuario y la contraseña proporcionados, si no se encuentra un usuario lanza la excepción UserNotFoundException y si lo encuentra lo devuelve.

newUser

Mediante la etiqueta `@PostMapping(value = "/new")` se permite manejar las solicitudes POST a /api/user/new. El método verifica si un usuario con el nombre de usuario proporcionado ya existe en el repositorio, si existe lanza la excepción UserAlreadyExistsException y si no, convierte el UsuarioDTO a una entidad Usuario y la guarda en el repositorio, retornando el usuario recién creado.

5.1.3.3 VisitanteController

```java
@RestController
@CrossOrigin
@RequestMapping(value = "api/visitantes")
public class VisitanteController extends BaseController{

    @GetMapping(value = "/all")
    public List<Visitante> getVisitante() throws visitantesNotFoundException {

        List<Visitante> lista = visitanteRepository.findAll();

        if(lista.isEmpty()) {
            throw new visitantesNotFoundException();
        }

        return lista;

    }

    @GetMapping(value = "/filter")
    public List<Visitante> buscarVisitantes (
            @RequestParam(value = "dni", required = false) String dni,
            @RequestParam(value = "name", required = false) String name,
            @RequestParam(value = "surname", required = false) String surname,
            @RequestParam(value = "surname2", required = false) String surname2,
            @RequestParam(value = "fatherName", required = false) String fatherName,
            @RequestParam(value = "motherName", required = false) String motherName) throws visitantesNotFoundException{

        VisitanteFilter.VisitanteFilterBuilder builderVisitante = VisitanteFilter.builder();
        builderVisitante = builderVisitante.dni(dni).name(name).surname(surname).surname2(surname2).fatherName(fatherName).motherName(motherName);

        List<Visitante> visitantes = visitanteRepository.findAll(builderVisitante.build());


        if(visitantes == null || visitantes.isEmpty()) {
            throw new visitantesNotFoundException();
        }

        return visitantes;

    }

    @PostMapping(value = "/new")
    public Visitante newVisitante(
            @RequestBody VisitanteDTO visitanteDTO) throws  VisitanteAlreadyExistException {

        Visitante visitante = visitanteRepository.findByDniIgnoreCase(visitanteDTO.getDni());

        if(visitante != null) {
            throw new VisitanteAlreadyExistException();
        }

        return visitanteRepository.save(visitanteDTO.toVisitante());
    }
```

Ilustración 41 - VisitanteController

El VisitanteController es un controlador REST que maneja las solicitudes HTTP relacionadas con la entidad Visitante. Hereda del BaseController, lo que le permite acceder a los repositorios inyectados en la clase base.

La etiqueta `@RestController` indica que esta clase es un controlador donde cada método retorna un objeto de dominio. Spring convierte automáticamente las respuestas en JSON. La etiqueta `@CrossOrigin` permite que las solicitudes de origen cruzado sean manejadas por este controlador, etiqueta necesaria para permitir que el Frontend haga solicitudes a este backend. La etiqueta `@RequestMapping` define la ruta base para todas las solicitudes manejadas por este controlador, en este caso las rutas comenzaran con /api/visitantes.

getVisitante

Mediante la etiqueta `@GetMapping(value = "/all")` se permite manejar las solicitudes GET a /api/visitantes/all. El método busca todos los visitantes en el repositorio. Si no encuentra visitantes, lanza la excepción VisitantesNotFoundException, y si encuentra al menos uno lo devuelve en una lista de visitantes.

buscarVisitantes

Mediante la etiqueta `@GetMapping(value = "/filter")` se permite manejar las solicitudes GET a /api/visitantes/filter. El método construye un filtro VisitanteFilter basado en los parámetros proporcionados.

```java
@Builder
public class VisitanteFilter implements Specification<Visitante>{
    private String dni;
    private String name;
    private String surname;
    private String surname2;
    private String fatherName;
    private String motherName;

    @Override
    public Predicate toPredicate(Root<Visitante> root, CriteriaQuery<?> query, CriteriaBuilder cb) {
        ArrayList<Predicate> predicates = new ArrayList<>();

        if(dni != null) {
            predicates.add(cb.like(cb.upper(root.get("dni")),
                    "%" + StringUtils.normalize(dni).toUpperCase().trim() + "%"));
        }

        if(name != null) {
            predicates.add(cb.like(cb.upper(root.get("nombre")),
                    "%" + StringUtils.normalize(name).toUpperCase().trim() + "%"));
        }

        if(surname != null) {
            predicates.add(cb.like(cb.upper(root.get("primerApellido")),
                    "%" + StringUtils.normalize(surname).toUpperCase().trim() + "%"));
        }

        if(surname2 != null) {
            predicates.add(cb.like(cb.upper(root.get("segundoApellido")),
                    "%" + StringUtils.normalize(surname2).toUpperCase().trim() + "%"));
        }

        if(fatherName != null) {
            predicates.add(cb.like(cb.upper(root.get("nombrePadre")),
                    "%" + StringUtils.normalize(fatherName).toUpperCase().trim() + "%"));
        }

        if(motherName != null) {
            predicates.add(cb.like(cb.upper(root.get("nombreMadre")),
                    "%" + StringUtils.normalize(motherName).toUpperCase().trim() + "%"));
        }


        return predicates.isEmpty() ? null : cb.and(predicates.toArray(new Predicate[predicates.size()]));
    }

}
```

Ilustración 42 - VisitanteFilter

Una vez construido el filtro, busca visitantes que coincidan con el filtro en el repositorio. Si no se encuentran visitantes, lanza la excepción VisitantesNotFoundException, y si los encuentra devuelve la lista de visitantes que coinciden con el filtro.

newVisitante

Mediante la etiqueta `@PostMapping(value = "/new")` se permite manejar las solicitudes POST a /api/visitantes/new. El método verifica si un visitante con el DNI proporcionado ya existe en el repositorio. Si existe, lanza una excepción VisitanteAlreadyExistsException y si no, convierte el VisitanteDTO en una entidad Visitante y lo guarda en el repositorio, retornando el visitante recién creado.

```
@GetMapping(value = "/get")
public Visitante getVisitante(
        @RequestParam(value = "id") Integer id) throws VisitanteNotFoundException  {

    Visitante visitante = visitanteRepository.findByVisitanteId(id);

    if(visitante == null) {
        throw new VisitanteNotFoundException();
    }

    return visitante;

}

@PutMapping(value = "/edit/{visitanteId}")
public Visitante editVisitante(
        @PathVariable(value = "visitanteId") Integer id,
        @RequestBody VisitanteDTO visitanteDTO) throws VisitanteNotFoundException   {

    Visitante visitanteOriginal = visitanteRepository.findByVisitanteId(id);

    if(visitanteOriginal == null) {
        throw new VisitanteNotFoundException();
    }

    Visitante mergedVisitante = Visitante.mergeVisitante(visitanteOriginal, visitanteDTO.toVisitante());

    return  visitanteRepository.save(mergedVisitante);
}

@DeleteMapping(value = "/delete/{Id}")
public Visitante deleteVisitante(
        @PathVariable(value = "Id") Integer id
        ) throws VisitanteNotFoundException   {

    Visitante visitante = visitanteRepository.findByVisitanteId(id);

    if(visitante== null) {
        throw new VisitanteNotFoundException();
    }

    visitanteRepository.delete(visitante);

    return  visitante;
}
```

Ilustración 43 - VisitanteController II

getVisitante

Mediante la etiqueta `@GetMapping(value = "/get")` se permite manejar las solicitudes GET a /api/visitantes/get. El método busca un visitante por su ID en el repositorio. Si no se encuentra un visitante con el ID proporcionado, lanza una excepción VisitanteNotFoundException y si no, retorna el visitante encontrado.

editVisitante

Mediante la etiqueta `@PutMapping(value = "/edit/{visitanteId}")` se permite manejar las solicitudes PUT a /api/visitantes/edit/{visitanteId}. El método busca un visitante por su ID en el repositorio. Si no se encuentra un visitante con el ID proporcionado, lanza una excepción VisitanteNotFoundException. Si lo encuentra fusiona los datos del VisitanteDTO con el visitante original encontrado, los guarda y devuelve el visitante actualizado.

deleteVisitante

Mediante la etiqueta `@DeleteMapping(value = "/delete/{Id}")` se permite manejar las solicitudes DELETE a /api/visitantes/delete/{Id}. El método busca un visitante por su ID en el repositorio. Si no se encuentra un visitante con el ID proporcionado, lanza una excepción VisitanteNotFoundException. Si lo encuentra lo elimina del repositorio y devuelve el visitante eliminado.

5.1.3.4 TipoAutorizacionController

```java
@RestController
@CrossOrigin
@RequestMapping(value = "api/tipoAutorizacion")
public class TipoAutorizacionController extends BaseController{

    @GetMapping(value = "/all")
    public List<TipoAutorizacion> getTiposAutorizacion() throws TiposAutorizacionNotFoundException {

        List<TipoAutorizacion> lista = tipoAutorizacionRepository.findAll();

        if(lista.isEmpty()) {
            throw new TiposAutorizacionNotFoundException();
        }

        return lista;

    }}
```

Ilustración 44 - TipoAutorizacionController

El TipoAutorizacionController es un controlador REST que maneja las solicitudes HTTP relacionadas con la entidad TipoAutorizacion. Hereda del BaseController, lo que le permite acceder a los repositorios inyectados en la clase base.

La etiqueta `@RestController` indica que esta clase es un controlador donde cada método retorna un objeto de dominio. Spring convierte automáticamente las respuestas en JSON. La etiqueta `@CrossOrigin` permite que las solicitudes de origen cruzado sean manejadas por este controlador, etiqueta necesaria para permitir que el Frontend haga solicitudes a este backend. La etiqueta `@RequestMapping` define la ruta base para todas las solicitudes manejadas por este controlador, en este caso las rutas comenzaran con /api/tipoAutorizacion.

getTiposAutorizacion

Mediante la etiqueta `@GetMapping(value = "/all")` se permite manejar las solicitudes GET a /api/tipoAutorizacion/all. El método busca todos los tipos de autorización en el repositorio. Si no encuentra tipos de autorizacion, lanza una excepción TiposAutorizacionNotFoundException. Si los encuentra devuelve la lista de tipos de autorizacion.

5.1.3.5 AutorizacionController

```java
@RestController
@CrossOrigin
@RequestMapping(value = "api/autorizaciones")
public class AutorizacionController extends BaseController{

    @GetMapping(value = "/all")
    public List<Autorizacion> getAutorizaciones() throws autorizacionesNotFoundException {

        List<Autorizacion> lista = autorizacionRepository.findAll();

        if(lista.isEmpty()) {
            throw new autorizacionesNotFoundException();
        }

        return lista;

    }

    @GetMapping(value = "/filter")
    public List<Autorizacion> buscarAutorizaciones (
            @RequestParam(value = "motivoAutorizacion", required = false) String motivoAutorizacion,
            @RequestParam(value = "tipoAutorizacion", required = false) Long tipoAutorizacion) throws autorizacionesNotFoundException{

        AutorizacionFilter.AutorizacionFilterBuilder builderAutorizacion = AutorizacionFilter.builder();
        builderAutorizacion = builderAutorizacion.motivoAutorizacion(motivoAutorizacion).tipoAutorizacion(tipoAutorizacion);

        List<Autorizacion> autorizaciones = autorizacionRepository.findAll(builderAutorizacion.build());


        if(autorizaciones == null || autorizaciones.isEmpty()) {
            throw new autorizacionesNotFoundException();
        }

        return autorizaciones;

    }

    @PostMapping(value = "/new")
    public Autorizacion newAutorizacion(
            @RequestBody AutorizacionDTO autorizacionDTO) {
        Autorizacion autorizacion = autorizacionDTO.toAutorizacion();

        autorizacion.setTipoAutorizacion(tipoAutorizacionRepository.findByTipoAutorizacionId(autorizacionDTO.getTipoAutorizacion()));

        return autorizacionRepository.save(autorizacion);
    }
```

Ilustración 45 - AutorizacionController

El AutorizacionController es un controlador REST que maneja las solicitudes HTTP relacionadas con la entidad Autorizacion. Hereda del BaseController, lo que le permite acceder a los repositorios inyectados en la clase base.

La etiqueta `@RestController` indica que esta clase es un controlador donde cada método retorna un objeto de dominio. Spring convierte automáticamente las respuestas en JSON. La etiqueta `@CrossOrigin` permite que las solicitudes de origen cruzado sean manejadas por este controlador, etiqueta necesaria para permitir que el Frontend haga solicitudes a este backend. La etiqueta `@RequestMapping` define la ruta base para todas las solicitudes manejadas por este controlador, en este caso las rutas comenzaran con /api/autorizaciones.

getAutorizaciones

Mediante la etiqueta `@GetMapping(value = "/all")` se permite manejar las solicitudes GET a /api/autorizaciones/all. El método busca todas las autorizaciones en el repositorio. Si no se encuentran autorizacion en el repositorio, se lanza una excepción AutorizacionesNotFoundException. Si las encuentra devuelve una lista con las autorizaciones.

buscarAutorizaciones

Mediante la etiqueta `@GetMapping(value = "/all")` se permite manejar las solicitudes GET a /api/autorizaciones/all. El método busca autorizaciones en el repositorio que coincidan con los filtros proporcionados (motivoAutorizacion y tipoAutorizacion). Si no se encuentran autorizaciones que coincidan con los criterios de búsqueda, se lanza una excepción AutorizacionesNotFoundException. Si las encuentra, devuelve una lista con las autorizaciones filtradas.

```java
@Builder
public class AutorizacionFilter  implements Specification<Autorizacion>{
    private String motivoAutorizacion;
    private Long tipoAutorizacion;


    @Override
    public Predicate toPredicate(Root<Autorizacion> root, CriteriaQuery<?> query, CriteriaBuilder cb) {
        ArrayList<Predicate> predicates = new ArrayList<>();

        if(motivoAutorizacion != null) {
            predicates.add(cb.like(cb.upper(root.get("motivoAutorizacion")),
                    "%" + StringUtils.normalize(motivoAutorizacion).toUpperCase().trim() + "%"));
        }

        if(tipoAutorizacion != null) {
            predicates.add(cb.equal(root.get("tipoAutorizacion").get("tipoAutorizacionId"), tipoAutorizacion));
        }

        return predicates.isEmpty() ? null : cb.and(predicates.toArray(new Predicate[predicates.size()]));
    }


}
```

Ilustración 46 - AutorizacionFilter

newAutorizacion

Mediante la etiqueta `@PostMapping(value = "/new")` se permite manejar las solicitudes POST a /api/autorizaciones/new. El método crea una nueva autorización a partir del DTO proporcionado en el cuerpo de la solicitud. La autorización se asocia al tipo de autorización correspondiente y se guarda en el repositorio. Devuelve la nueva autorización creada.

```java
@GetMapping(value = "/get")
public Autorizacion getAutorizacion(
        @RequestParam(value = "id") Integer id) throws AutorizacionNotFoundException  {

    Autorizacion autorizacion = autorizacionRepository.findByAutorizacionId(id);

    if(autorizacion == null) {
        throw new AutorizacionNotFoundException();
    }

    return autorizacion;

}

@PutMapping(value = "/edit/{autorizacionId}")
public Autorizacion editAutorizacion(
        @PathVariable(value = "autorizacionId") Integer id,
        @RequestBody AutorizacionDTO autorizacionDTO) throws AutorizacionNotFoundException   {

    Autorizacion autorizacionOriginal = autorizacionRepository.findByAutorizacionId(id);

    if(autorizacionOriginal == null) {
        throw new AutorizacionNotFoundException();
    }

    Autorizacion mergedAutorizacion = Autorizacion.mergeAutorizacion(autorizacionOriginal, autorizacionDTO.toAutorizacion());

    return  autorizacionRepository.save(mergedAutorizacion);
}

@DeleteMapping(value = "/delete/{Id}")
public Autorizacion deleteAutorizacion(
        @PathVariable(value = "Id") Integer id
        ) throws AutorizacionNotFoundException   {

    Autorizacion autorizacion = autorizacionRepository.findByAutorizacionId(id);

    if(autorizacion== null) {
        throw new AutorizacionNotFoundException();
    }

    autorizacionRepository.delete(autorizacion);

    return  autorizacion;
}

@PutMapping(value = "/{autorizacionId}/visitantesAutorizados/{visitanteDNI}")
public Visitante asignarVisitante(
        @PathVariable(value = "autorizacionId") Integer autorizacionId,
        @PathVariable(value = "visitanteDNI") String visitanteDNI) throws PersonAlreadyInAuthorizationException {
    Autorizacion autorizacion = autorizacionRepository.findByAutorizacionId(autorizacionId);
    Visitante visitante = visitanteRepository.findByDniIgnoreCase(visitanteDNI);

    if(!autorizacion.getVisitantesAutorizados().add(visitante)) {
        throw new PersonAlreadyInAuthorizationException();
    }

    autorizacionRepository.save(autorizacion);
```

Ilustración 47 - AutorizacionController II

getAutorizacion

Mediante la etiqueta `@GetMapping(value = "/get")` se permite manejar las solicitudes GET a /api/autorizaciones/get. El método busca una autorización en el repositorio por su ID. Si no se encuentra la autorización, se lanza una excepción AutorizacionNotFoundException. Si la encuentra, devuelve la autorización.

editAutorizacion

Mediante la etiqueta `@PutMapping(value = "/edit/{autorizacionId}")` se permite manejar las solicitudes PUT a /api/autorizaciones/edit/{autorizacionId}. El método actualiza una autorización existente con los datos proporcionados en el DTO en el cuerpo de la solicitud. Si no se encuentra la autorización, se lanza una excepción AutorizacionNotFoundException. Si la encuentra, la autorización se actualiza y se guarda en el repositorio, devolviendo la autorización actualizada.

deleteAutorizacion

Mediante la etiqueta `@DeleteMapping(value = "/delete/{Id}")` se permite manejar las solicitudes DELETE a /api/autorizaciones/delete/{Id}. El método elimina una autorización del repositorio por su ID. Si no se encuentra la autorización, se lanza una excepción AutorizacionNotFoundException. Si la encuentra, se elimina la autorización del repositorio y se devuelve la autorización eliminada.

asignarVisitante

Mediante la etiqueta `@PutMapping(value = "/{autorizacionId}/visitantesAutorizados/{visitanteDNI}")` se permite manejar las solicitudes PUT a /api/autorizaciones/{autorizacionId}/visitantesAutorizados/{visitanteDNI}. El método asigna un visitante a una autorización específica. Si el visitante ya está asignado a la autorización, se lanza una excepción PersonAlreadyInAuthorizationException. Si no, el visitante se asigna a la autorización, se guarda la autorización actualizada en el repositorio y se devuelve el visitante asignado.

```java
@GetMapping(value = "/{autorizacionId}/visitantesAutorizados")
public Set<Visitante> visitantesAutorizados(
        @PathVariable(value = "autorizacionId") Integer autorizacionId) throws AutorizacionNotFoundException {

    Autorizacion autorizacion = autorizacionRepository.findByAutorizacionId(autorizacionId);

    if(autorizacion == null) {
        throw new AutorizacionNotFoundException();
    }

    Set<Visitante> visitantesAutorizados = autorizacion.getVisitantesAutorizados();

    return visitantesAutorizados;
}

@DeleteMapping(value = "/{autorizacionId}/visitantesAutorizados/{visitanteId}")
public Visitante desasignarVisitante(
        @PathVariable(value = "autorizacionId")  Integer autorizacionId,
        @PathVariable(value = "visitanteId")  Integer visitanteId) {
    Autorizacion autorizacion = autorizacionRepository.findByAutorizacionId(autorizacionId);
    Visitante visitante = visitanteRepository.findByVisitanteId(visitanteId);



    autorizacion.getVisitantesAutorizados().remove(visitante);

    autorizacionRepository.save(autorizacion);

    return visitante;
}
```

Ilustración 48 - AutorizacionController III

visitantesAutorizados

Mediante la etiqueta `@GetMapping(value = "/{autorizacionId}/visitantesAutorizados")` se permite manejar las solicitudes GET a /api/autorizaciones/{autorizacionId}/visitantesAutorizados. El método obtiene el conjunto de visitantes autorizados para una autorización específica. Si no se encuentra la autorización, se lanza una excepción AutorizacionNotFoundException. Si la encuentra, devuelve el conjunto de visitantes autorizados, que puede ser vacío.

desasignarVisitante

Mediante la etiqueta `@DeleteMapping(value = "/{autorizacionId}/visitantesAutorizados/{visitanteId}")` se permite manejar las solicitudes DELETE a /api/autorizaciones/{autorizacionId}/visitantesAutorizados/{visitanteId}. El método desasigna un visitante de una autorización específica. Se guarda la autorización actualizada en el repositorio y se devuelve el visitante desasignado.

5.1.3.6 RegistroController

```java
@RestController
@RequestMapping("/api/registros")
public class RegistroController extends BaseController {

    @GetMapping("/activosHoy")
    public List<Registro> obtenerRegistrosActivosHoy() {
        LocalDate hoy = LocalDate.now();
        List<Registro> registrosActivosHoy = new ArrayList<>();

        // Obtener todas las autorizaciones activas para hoy
        List<Autorizacion> autorizacionesActivasHoy = autorizacionRepository
                .findByFechaInicioLessThanEqualAndFechaFinGreaterThanEqual(hoy, hoy);

        // Iterar sobre las autorizaciones activas
        for (Autorizacion autorizacion : autorizacionesActivasHoy) {
            Set<Visitante> visitantesAutorizados = autorizacion.getVisitantesAutorizados();


                // Crear un registro para cada visitante autorizado
                for (Visitante visitante : visitantesAutorizados) {
                    if (!registroRepository.existsByVisitanteAndAutorizacionAndFechaEntradaIsNotNullAndFechaSalidaIsNull(
                            visitante, autorizacion)) {
                        Registro registro = new Registro();
                        registro.setVisitante(visitante);
                        registro.setAutorizacion(autorizacion);
                        registrosActivosHoy.add(registro);
                    }
                }
            }

        return registrosActivosHoy;
    }

    @PostMapping("/new")
    public Registro registrarEntrada(@RequestBody RegistroDTO registroDTO) {

        Registro registro = registroDTO.toRegistro();

        registro.setFechaEntrada(LocalDateTime.now());

        return registroRepository.save(registro);
    }

    @GetMapping("/out")
    public List<Registro> obtenerRegistrosSalida() {

        return registroRepository.findByFechaEntradaIsNotNullAndFechaSalidaIsNull();
    }

    @PostMapping("/out")
    public Registro registrarSalida(@RequestParam(value = "id", required = true) Integer id) {

        Registro registro = registroRepository.findByRegistroId(id);

        registro.setFechaSalida(LocalDateTime.now());

        return registroRepository.save(registro);
    }
```

Ilustración 49 - RegistroController

El RegistroController es un controlador REST que maneja las solicitudes HTTP relacionadas con la entidad Registro. Hereda del BaseController, lo que le permite acceder a los repositorios inyectados en la clase base.

La etiqueta `@RestController` indica que esta clase es un controlador donde cada método retorna un objeto de dominio. Spring convierte automáticamente las respuestas en JSON. La etiqueta `@CrossOrigin` permite que las solicitudes de origen cruzado sean manejadas por este controlador, etiqueta necesaria para permitir que el Frontend haga solicitudes a este backend. La etiqueta `@RequestMapping` define la ruta base para todas las solicitudes manejadas por este controlador, en este caso las rutas comenzaran con /api/registros.

obtenerRegistrosActivosHoy

Mediante la etiqueta `@GetMapping("/activosHoy")` se permite manejar las solicitudes GET a /api/registros/activosHoy. El método obtiene la fecha actual y busca todas las autorizaciones activas para el día de hoy, es decir, aquellas cuya fecha de inicio sea anterior o igual a la fecha actual y cuya fecha de fin sea posterior o igual a la fecha actual. Para cada autorización activa, el método itera sobre los visitantes autorizados y, si no existe un registro de entrada sin salida para ese visitante y autorización, crea un nuevo registro y lo añade a la lista registrosActivosHoy. Finalmente, devuelve la lista de registros activos para hoy.

registrarEntrada

Mediante la etiqueta `@PostMapping("/new")` se permite manejar las solicitudes POST a /api/registros/new. El método crea un nuevo registro a partir del DTO proporcionado en el cuerpo de la solicitud. La fecha de entrada se establece como la fecha y hora actual. Luego, el registro se guarda en el repositorio y se devuelve el registro creado.

obtenerRegistrosSalida

Mediante la etiqueta `@GetMapping("/out")` se permite manejar las solicitudes GET a /api/registros/out. El método busca y devuelve todos los registros en los que la fecha de entrada no es nula y la fecha de salida es nula, es decir, registros de visitantes que han entrado, pero no han salido aún.

registrarSalida

Mediante la etiqueta `@PostMapping("/out")` se permite manejar las solicitudes POST a /api/registros/out. El método busca un registro por su ID proporcionado en la solicitud. Si encuentra el registro, establece la fecha de salida como la fecha y hora actual, guarda el registro actualizado en el repositorio y devuelve el registro actualizado.

5.1.4 Implementación de los repositorios

Para interactuar con la base de datos se han implementado diferentes repositorios usando la librería JpaRepository, que permiten realizar operaciones directamente sobre la base de datos sin necesidad de crear querys manualmente.

5.1.4.1 UsuarioRepository

```java
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    Usuario findByNombreUsuarioAndPasswordIgnoreCase(String username, String password);

    Usuario findByNombreUsuarioIgnoreCase(String nombreUsuario);
}
```

Ilustración 50 - UsuarioRepository

- `Usuario findByNombreUsuarioAndPasswordIgnoreCase(String username, String password)`:

  Este método busca un usuario en la base de datos que tenga el nombre de usuario (nombreUsuario) y la contraseña (password) especificados. La búsqueda es insensible a mayúsculas y minúsculas (IgnoreCase). Se utiliza para autenticar a los usuarios durante el inicio de sesión.

- `Usuario findByNombreUsuarioIgnoreCase(String nombreUsuario)`:

  Este método busca un usuario en la base de datos por su nombre de usuario (nombreUsuario). La búsqueda es insensible a mayúsculas y minúsculas (IgnoreCase). Se utiliza para verificar si un nombre de usuario ya existe antes de registrar un nuevo usuario.

5.1.4.2 VisitanteRepository

```java
public interface VisitanteRepository  extends JpaRepository<Visitante, Integer>, JpaSpecificationExecutor<Visitante>{

    List<Visitante> findAll(Specification<Visitante> specification);

    Visitante findByDniIgnoreCase(String dni);

    Visitante findByVisitanteId(Integer id);


}
```

Ilustración 51 - VisitanteRepository

- `List<Visitante> findAll(Specification<Visitante> specification)`:

  Este método encuentra todos los visitantes que cumplen con los criterios especificados por la Specification<Visitante>, que es el filtro personalizado construido en VisitanteFilter.

- `Visitante findByDniIgnoreCase(String dni)`:

  Este método busca un visitante en la base de datos por su DNI. La búsqueda es insensible a mayúsculas y minúsculas (IgnoreCase), lo que significa que no importa si el DNI está en mayúsculas o minúsculas.

- `Visitante findByVisitanteId(Integer id)`:

  Este método busca un visitante en la base de datos por su ID (visitanteId). Devuelve el visitante que tiene el ID especificado.

5.1.4.3 TipoAutorizacionRepository

```java
@Repository
public interface TipoAutorizacionRepository  extends JpaRepository<TipoAutorizacion, Long>{

    TipoAutorizacion findByTipoAutorizacionId(Long tipoAutorizacion);


}
```

Ilustración 52 - TipoAutorizacionRepository

- `TipoAutorizacion findByTipoAutorizacionId(Long tipoAutorizacion)`:

  Este método busca un tipo de autorización en la base de datos por su ID (tipoAutorizacion). Devuelve el tipo de autorización que tiene el ID especificado.

5.1.4.4 AutorizacionRepository

```java
@Repository
public interface AutorizacionRepository extends JpaRepository<Autorizacion, Integer>, JpaSpecificationExecutor<Autorizacion> {

    List<Autorizacion> findAll(Specification<Autorizacion> specification);

    Autorizacion findByAutorizacionId(Integer autorizacionId);

    List<Autorizacion> findByFechaInicioLessThanEqualAndFechaFinGreaterThanEqual(LocalDate hoy, LocalDate hoy2);
}
```

Ilustración 53 - AutorizacionRepository

- `List<Autorizacion> findAll(Specification<Autorizacion> specification)`:

  Este método busca todas las autorizaciones que cumplen con la especificación proporcionada, que es el filtro personalizado construido en AutorizacionFilter.

- `Autorizacion findByAutorizacionId(Integer autorizacionId)`:

  Este método busca una autorización en la base de datos por su ID (autorizacionId). Devuelve la autorización que tiene el ID especificado.

- `List<Autorizacion> findByFechaInicioLessThanEqualAndFechaFinGreaterThanEqual(LocalDate hoy, LocalDate hoy2)`:

  Este método busca todas las autorizaciones cuya fecha de inicio sea menor o igual a la fecha actual (hoy) y cuya fecha de fin sea mayor que la fecha actual (hoy2). Devuelve una lista de autorizaciones que cumplen con estos criterios.

5.1.4.5 RegistroRepository

```java
@Repository
public interface RegistroRepository extends JpaRepository<Registro, Integer> {

    boolean existsByVisitanteAndAutorizacionAndFechaEntradaIsNotNullAndFechaSalidaIsNull(Visitante visitante,
            Autorizacion autorizacion);

    List<Registro> findByFechaEntradaIsNotNullAndFechaSalidaIsNull();

    Registro findByRegistroId(Integer id);

}
```

Ilustración 54 - RegistroRepository

- `boolean existsByVisitanteAndAutorizacionAndFechaEntradaIsNotNullAndFechaSalidaIsNull(Visitante visitante, Autorizacion autorizacion)`:

  Este método verifica si existe algún registro en la base de datos que cumpla con las siguientes condiciones:

  - El registro pertenece al visitante proporcionado.
  - El registro está asociado con la autorización proporcionada.
  - El registro tiene una fecha de entrada no nula.
  - El registro no tiene una fecha de salida (es decir, la fecha de salida es nula).

  Devuelve true si el registro sobre el que se llama a este método cumple con estas condiciones, de lo contrario, devuelve false.

5.1.5 Implementación de las excepciones

Las excepciones se han implementado de una forma muy simple, con una etiqueta @ResponseStatus que indica el valor de la respuesta HTTP que generará dicha excepción y una razón, para que el Frontend pueda tratarlo correctamente. A continuación, se muestra como ejemplo la implementación de VisitanteAlreadyExistsException:

```java
@ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Ya existe un visitante con ese DNI")
public class VisitanteAlreadyExistException extends BaseException {

    /**
     *
     */
    private static final long serialVersionUID = -1593533401530036963L;

    @Override
    public ExceptionCode getCode() {
        return ExceptionCode.VISITANTE_ALREADY_EXISTS;
    }

}
```

Ilustración 55 - VisitanteAlreadyExistsExcepction

En la clase enum llamada ExceptionCode se han definido todas las excepciones que se pueden dar en la aplicación.

```java
@Getter
public enum ExceptionCode {
    USER_NOT_FOUND("Usuario no encontrado"),
    USER_ALREADY_EXISTS("Usuario ya existente"),
    NO_VISITANTES("No hay visitantes en la base de datos"),
    VISITANTE_ALREADY_EXISTS("Ya existe un visitante con ese DNI"),
    NO_VISITANTE("No existe el visitante con ese DNI"),
    TIPOS_AUTORIZACION_NOT_FOUND("No se han encontrado tipos de autorizacion"),
    NO_AUTORIZACIONES("No hay autorizaciones en la base de datos"),
    NO_AUTORIZACION("No existe la autorizacion"),
    VISITANTE_ALREADY_IN_AUTHORIZATION("Este visitante ya pertenece a la autorizacion");

    private final String description;

    public String getCode() {
        return name();
    }

    ExceptionCode(String description){
        this.description = description;
    }

}
```

Ilustración 56 - ExceptionCode

5.2 Desarrollo de la capa de presentación

La capa de presentación se ha desarrollado con React, haciendo uso de diferentes librerías para cumplir con los requisitos predefinidos.

5.2.1 Configuración inicial

Para el desarrollo del Frontend se ha hecho uso de las siguientes librerías definidas en el archivo package.json

```json
{
  "name": "ccaa-frontend",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview",
    "start": "run dev start"
  },
  "dependencies": {
    "es6-promise-debounce": "^1.0.1",
    "nodejs-base64-encode": "^1.1.0",
    "react": "^18.2.0",
    "react-confirm-alert": "^3.0.6",
    "react-datepicker": "^1.5.0",
    "react-day-picker": "^7.1.9",
    "react-dom": "^18.2.0",
    "react-icons": "^5.0.1",
    "react-redux": "^9.1.0",
    "react-router-dom": "^6.23.1",
    "react-router-redux": "^4.0.8",
    "react-select": "^2.3.0",
    "react-semantic-redux-form": "^1.2.7",
    "react-table": "^6.8.6",
    "react-toastify": "^10.0.4",
    "redux": "^5.0.1",
    "redux-form": "^8.3.10",
    "redux-form-validators": "^3.3.2",
    "redux-saga": "^1.3.0",
    "semantic-ui-css": "^2.3.1",
    "semantic-ui-react": "^0.81.1",
    "styled-components": "^6.1.11"
  },
  "devDependencies": {
    "@types/react": "^18.2.55",
    "@types/react-dom": "^18.2.19",
    "@vitejs/plugin-react": "^4.2.1",
    "eslint": "^8.56.0",
    "eslint-plugin-react": "^7.33.2",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.5",
    "vite": "^5.1.0"
  }
}
```

Ilustración 57 - Package.json

- **es6-promise-debounce (^1.0.1)**: Una biblioteca que proporciona una función de debounce.
- **nodejs-base64-encode (^1.1.0)**: Una utilidad para codificar y decodificar datos en formato base64 en Node.js.
- **react (^18.2.0)**: versión de React utilizada.
- **react-confirm-alert (^3.0.6)**: Un componente de diálogo de confirmación personalizable.
- **react-datepicker (^1.5.0)**: Un componente de selector de fechas.
- **react-day-picker (^7.1.9)**: Un componente de selector de fecha.
- **react-dom (^18.2.0)**: Paquete para manipular el DOM.
- **react-icons (^5.0.1)**: Una biblioteca de iconos.
- **react-redux (^9.1.0)**: La integración oficial de React para Redux, una biblioteca para manejar el estado de la aplicación.
- **react-router-dom (^6.23.1)**: Una biblioteca que proporciona enrutamiento.
- **react-router-redux (^4.0.8)**: Integración de React Router con Redux.
- **react-select (^2.3.0)**: Un componente de selección.
- **react-semantic-redux-form (^1.2.7)**: Integración de Redux Form con Semantic UI.
- **react-table (^6.8.6)**: Una biblioteca para crear tablas.
- **react-toastify (^10.0.4)**: Una biblioteca para mostrar notificaciones toast.
- **redux (^5.0.1)**: Una biblioteca para manejar el estado global de la aplicación en JavaScript.
- **redux-form (^8.3.10)**: Una biblioteca para gestionar formularios.
- **redux-form-validators (^3.3.2)**: Un conjunto de validadores para Redux Form.
- **redux-saga (^1.3.0)**: Una biblioteca para manejar efectos secundarios asincrónicos en aplicaciones Redux.
- **semantic-ui-css (^2.3.1)**: Los estilos CSS de Semantic UI.
- **semantic-ui-react (^0.81.1)**: Componentes de Semantic UI.
- **styled-components (^6.1.11)**: Una biblioteca para escribir estilos CSS dentro de componentes.

5.2.2 Desarrollo de las vistas y componentes

5.2.2.1 App.jsx

Es el componente principal de la aplicación

```
export const store = configureSagaStore();

class App extends Component {
  render() {
    return(
    <Provider store={store}>
      <div className="container">
        <ControlAccesoMenu/>
        <Main/>
        <ToastContainer
          position="top-center"
          autoclose={5000}
          hideProgresBar={false}
          newestOnTop
          closeOnClick
          rtl={false}
          pauseOnVisibilityChange
          draggablepauseOnHover
        />
      </div>
    </Provider>
    )
  }

}

export default App;
```

Ilustración 58 - App.jsx

En él se define el almacenamiento global de la aplicación (store) creando una instancia global de Redux Saga mediante la función configureSagaStore y se asigna la constante store. Se cargan los 2 componentes principales de la aplicación:

- ControlAccesoMenu: menú superior de la aplicación que aparecerá en todo momento salvo cuando no estemos logueados.
- Main: componente desde el que se cargan el resto de vistas de la aplicación.

```
render() {
    const { activeItem, redirectLogin } = this.state;
    const { user } = this.props;

    if (redirectLogin) {
        return <Navigate to="/" />;
    }
    return (
        <Fragment>
            {isLoggedUser() && <Menu compact fluid className={'menu'} icon={'labeled'} size={'small'}>

                <Menu.Item name="visitantes" active={activeItem.indexOf('visitantes') !== -1} as={Link} to={'/visitantes'} onClick={this.handleItemClick}>
                    <Icon size="big" name='user circle' />Visitantes<br />
                </Menu.Item>

                <Menu.Item name="autorizaciones" active={activeItem.indexOf('autorizaciones') !== -1}  as={Link} to={'/Autorizaciones'} onClick={this.handleItemClick}>
                    <Icon size='big' name='address card' style={{ paddingTop: '0px', paddingRight: '0px', paddingLeft: '0px', paddingBottom: '0px' }} /> Autorizaciones
                </Menu.Item>


                <Menu.Item style={{ marginLeft: '639px' }}></Menu.Item>
                <Menu.Item name="registroEntrada" position='center' active={activeItem.indexOf('registroEntrada') !== -1} as={Link} to={'/Entradas'} onClick={this.handleItemClick}>
                    <ImEnter size={25} style={{ marginBottom: '7px' }} /> Registrar <br />entrada
                </Menu.Item>
                <Menu.Item name="registroSalida" position='center' active={activeItem.indexOf('registroSalida') !== -1} as={Link} to={'/Salidas'} onClick={this.handleItemClick}>
                    <ImExit size={25} style={{ marginBottom: '7px' }} /> Registrar <br />salida
                </Menu.Item>
                <Menu.Item position='right' style={{ paddingRight: '0px', paddingLeft: '0px' }}></Menu.Item>

                <Menu.Item style={{ marginLeft: '0px' }} onClick={() => this.signOut()}>
                    <Icon size='big' name='sign out' /> Cerrar <br /> sesión
                </Menu.Item>
            </Menu>}
        </Fragment>

    )
}
```

Ilustración 59 - ControlAccesoMenu.jsx

El resultado visual de este componente es el siguiente:

Ilustración 60 - Menú superior de la aplicación

En el componente Main.jsx se crean los diferentes botones de la aplicación desde los que podremos navegar a las diferentes pantallas.

```
render() {

    const {searching} = this.props;
    return (
        <div>
            <Dimmer active={searching} inverted>
                    <Loader indeterminate>Cargando</Loader>
                </Dimmer>
            <Routes>
                <Route exact path="/" element={<Login />} />
                <Route exact path='/home' element={<Home />} />

                <Route exact path='/NewUser' element={<NewUser/>}/>

                <Route exact path='/Visitantes' element={<Visitantes/>}/>
                <Route exact path='/newVisitante' element={<NewVisitante/>}/>
                <Route exact path='/edit-visitante/:id' element={<EditVisitante/>}/>

                <Route exact path='/Autorizaciones' element={<Autorizaciones/>}/>
                <Route exact path='/newAutorizacion' element={<NewAutorizacion/>}/>
                <Route exact path='/edit-autorizacion/:id' element={<EditAutorizacion/>}/>

                <Route exact path='/Entradas' element={<Entradas/>}/>
                <Route exact path='/Salidas' element={<Salidas/>}/>


            </Routes>
        </div>
    )
}
```

Ilustración 61 - Main.jsx

En este componente se configuran las diferentes rutas de la aplicación, que cargarán una vista diferente dependiendo de la ruta a la que nos movamos. Tambien se carga el elemento Dimmer, que será la ruleta con el texto “cargando” que aparecerá mientras se realiza cualquier petición, bloqueando la aplicación hasta la resolución de la petición.

5.2.2.2 Login.jsx

Esta vista cargará el componente LoginForm.jsx que mostrará el formulario de inicio de sesión con los campos “Nombre de usuario” y “contraseña”.

```
return (
    <div className="login">
        <h3 className="centered">Iniciar sesión en Control de Acceso</h3>
        <Segment>
            <Form
                name="LoginForm"
                onSubmit={handleSubmit(onSubmit)}
            >
                <Form.Field centered required>
                    <label>Usuario</label>
                    <Field
                        type="text"
                        component={InputField}
                        name="user"
                        placeholder="Introduce tu nombre de usuario"
                        validate={required({ msg: "El campo USUARIO es obligatorio" })}
                    />
                </Form.Field>
                <Form.Field centered required>
                    <label >Contraseña</label>
                    <Field
                        type="password"
                        component={InputField}
                        name="password"
                        placeholder="Introduce tu contraseña"
                        validate={required({ msg: "El campo CONTRASEÑA es obligatorio" })}
                    />
                </Form.Field>
                <Form.Field className="centered" >
                    <Button primary type="submit">
                        <Icon name="signup" />Iniciar Sesión
                    </Button>

                </Form.Field>

            </Form>

        </Segment>
    </div>
)
```

Ilustración 62 - LoginForm.jsx

5.2.2.3 NewUser.jsx

La vista NewUser cargará el componente NewUserForm.jsx, que mostrará el formulario de registro de un nuevo usuario con los diferentes campos obligatorios a cumplimentar.

```jsx
return (
    <div className="login">
        <h3 className="centered">Registrar nuevo usuario</h3>
        <Segment>
            <Form
                name="newUserForm"
                onSubmit={handleSubmit(onSubmit)}
            >
                <Form.Field required>
                    <label>Nombre</label>
                    <Field
                        type="text"
                        component={InputField}
                        name="name"
                        placeholder="Introduce tu nombre"
                        validate={required({ msg: "El campo USUARIO es obligatorio" })}
                    />
                </Form.Field>
                <Form.Field required>
                    <label>Primer Apellido</label>
                    <Field
                        type="text"
                        component={InputField}
                        name="surname1"
                        placeholder="Introduce tu primer apellido"
                        validate={required({ msg: "El campo PRIMER APELLIDO es obligatorio" })}
                    />
                </Form.Field>
                <Form.Field required>
                    <label>Segundo Apellido</label>
                    <Field
                        type="text"
                        component={InputField}
                        name="surname2"
                        placeholder="Introduce tu segundo apellido"
                        validate={required({ msg: "El campo SEGUNDO APELLIDO es obligatorio" })}
                    />
                </Form.Field>
```

Ilustración 63 - NewUserForm.jsx I

```
                <Form.Field required>
                    <label>Correo Electrónico</label>
                    <Field
                        type="text"
                        component={InputField}
                        name="email"
                        placeholder="Introduce tu correo electrónico"
                        validate={required({ msg: "El campo CORREO ELECTRONICO es obligatorio" })}
                    />
                </Form.Field>
                <Form.Field required>
                    <label>Nombre de Usuario</label>
                    <Field
                        type="text"
                        component={InputField}
                        name="username"
                        placeholder="Introduce tu nombre de usuario"
                        validate={required({ msg: "El campo NOMBRE DE USUARIO es obligatorio" })}
                    />
                </Form.Field>
                <Form.Field required>
                    <label>Contraseña</label>
                    <Field
                        type="password"
                        component={InputField}
                        name="password"
                        placeholder="Introduce tu contraseña"
                        validate={required({ msg: "El campo CONTRASEÑA es obligatorio" })}
                    />
                </Form.Field>
                <Grid>
                    <Grid.Row>
                        <Grid.Column width={6} textAlign="left">
                            <Button negative as={Link} to={'/'}>
                                <Icon name="left arrow" />Volver
                            </Button>
                        </Grid.Column>
                        <Grid.Column width={10} textAlign="left">
                            <Button primary type="submit">
                                <Icon name="signup" />Registrarse
                            </Button>
                        </Grid.Column>
                    </Grid.Row>
                </Grid>
            </Form>
```

Ilustración 64 - NewUserForm.jsx II

5.2.2.4 Home.jsx

Una vez iniciada la sesión ya sea por el formulario de inicio de sesión corriente o tras haber creado una nueva cuenta se nos redirigirá a la pantalla principal de la aplicación, donde ya podremos seleccionar una opción del menú superior para dirigirnos a una de las diferentes vistas de la aplicación.

```
class Home extends Component {
    constructor(props){
        super(props);
        this.state = {

        }
    }

    render() {



        if (!isLoggedUser()) {
            return <Navigate to='/' />
        }



        return (
            <div><Segment>
                    <h2 className="centered">Bienvenido a la aplicacion de Control de Seguridad de Accesos</h2>
                    <h4 className="centered">Por favor, selecciona una opción del menu superior</h4>
                </Segment>
            </div>
        )
    }
}
```

Ilustración 65 - Home.jsx

A partir de este punto la implementación de todas las vistas es similar. Al pulsar sobre una opción del menú superior se nos redirigirá a la vista asociada a la ruta que nos transporta la selección de dicha opción. Las vistas de Visitantes y Autorizaciones constan de dos componentes cada una. Un componente formulario, que muestra los campos por los que se puede filtrar la búsqueda de visitantes y autorizaciones, y el otro componente que será la tabla donde se volcarán los datos buscados gracias al formulario anteriormente descrito. Las vistas de entradas y salidas son similares salvo que no hay ningún formulario, al entrar en cada una de ellas la store lanza una petición que recupera los visitantes pendientes de entrar o los pendientes de salir y los plasma en un único componente en forma de tabla. A continuación, se detalla a fondo la implementación de la vista de visitantes, que es extrapolable a las otras tres vistas de la aplicación.

5.2.2.5 Visitantes.jsx

Como se ha indicado anteriormente, esta vista carga tanto el componente VisitantesForm como el componente VisitantesTable.

```
class Visitantes extends Component {
    render() {

        if (!isLoggedUser()) {
            return <Navigate to='/' />
        }
        return(
            <div>
                <VisitantesForm/>
                <VisitantesTable/>
            </div>
        )
    }
}
export default Visitantes;
```

Ilustración 66 - Visitantes.jsx

5.2.2.6 VisitantesForm.jsx

Este componente muestra los diferentes campos que se pueden cumplimentar para filtrar la búsqueda de visitantes, que se lanzará al pulsar sobre el botón "Buscar".

```jsx
return (
    <Segment color="black">
        <Form
            name="VisitantesForm"
            onSubmit={handleSubmit(onSubmit)}
        >
            <Grid>
                <Grid.Row>
                    <Grid.Column>
                        <Form.Group widths="equal">
                            <Form.Field>
                                <Field
                                    type="text"
                                    label="DNI"
                                    component={InputField}
                                    name="dni"
                                    placeholder="DNI"
                                />
                            </Form.Field>
                            <Form.Field>
                                <Field
                                    type="text"
                                    label="Nombre"
                                    component={InputField}
                                    name="name"
                                    placeholder="NOMBRE"
                                />
                            </Form.Field>
                            <Form.Field>
                                <Field
                                    type="text"
                                    label="Primer Apellido"
                                    component={InputField}
                                    name="surname1"
                                    placeholder="PRIMER APELLIDO"
                                />
                            </Form.Field>
                            <Form.Field>
                                <Field
                                    type="text"
                                    label="Segundo Apellido"
                                    component={InputField}
                                    name="surname2"
                                    placeholder="SEGUNDO APELLIDO"
                                />
                            </Form.Field>
```

Ilustración 67 - VisitantesForm.jsx I

```
                    <Form.Field>
                        <Field
                            type="text"
                            label="Nombre Padre"
                            component={InputField}
                            name="fatherName"
                            placeholder="NOMBRE PADRE"
                        />
                    </Form.Field>
                    <Form.Field>
                        <Field
                            type="text"
                            label="Nombre Madre"
                            component={InputField}
                            name="MotherName"
                            placeholder="NOMBRE MADRE"
                        />
                    </Form.Field>
                </Form.Group>
            </Grid.Column>
        </Grid.Row>
        <Grid.Row className="centered">
            <Form.Field>
                <Button positive type="submit">
                    <Icon name="search" /> Buscar
                </Button>
            </Form.Field>
            <Form.Field>
                <Button negative type="button" onClick={this.resetForm}>
                    <Icon name="erase" /> Limpiar
                </Button>
            </Form.Field>
        </Grid.Row>
        <Grid.Row className="botonDerecha">
            <Form.Field >
                <Button primary as={Link} to={'/newVisitante'} >
                    <Icon name='user'/>Alta Nuevo Visitante
                </Button>
            </Form.Field>
        </Grid.Row>
```

Ilustración 68 - VisitantesForm.jsx II

Al pulsar sobre el botón guardar se lanzará el método submit del formulario de Redux, que recoge todos los valores de los campos del formulario `<Form.Field>` y los podremos tratar para comenzar el proceso de creación de una llamada API REST a la capa de aplicación.

```
const onSubmit = (values) => {
    store.dispatch({ type: types.SET_SEARCHING_TRUE});
    store.dispatch({ type: types.CLEAR_VISITANTES});
    if((values.dni === undefined || values.dni === "") && (values.name === undefined || values.name === "") && (values.surname1 === undefined || values.surname1 === "") &&
    (values.surname2 === undefined || values.surname2 === "") && (values.fatherName === undefined || values.fatherName === "") && (values.motherName === undefined || values.motherName === "")){
        store.dispatch({ type: types.GET_ALL_VISITANTES, model: values});
    }else{
        store.dispatch({ type: types.GET_VISITANTES_FILTER, model: values});
    }
}
```

Ilustración 69 - VisitantesForm - onSubmit

Como se puede observar, el método submit lo primero que hacer es lanzar la petición que activa el Dimmer de carga mientras se realiza la petición y acto seguido vacía la variable global que contiene la lista de visitantes, ya que va a ser rellenada con la información que nos devuelva el backend. El método hace una comprobación para saber si se ha cumplimentado algún campo del formulario para lanzar la petición que recupera todos los visitantes en caso de no haber rellenado ninguno o lanzar la petición de búsqueda con filtros, pasando como parámetro los valores recogidos por el formulario.

El botón limpiar lanza el método resetForm(), que llama a un action de Redux que vacía los campos del formulario que se les pase como parámetro, en este caso VisitantesForm. Este método también lanza la petición de CLEAR_VISITANTES, que vacía la variable global visitantes para vaciar la tabla en caso de estar mostrando algún dato.

```
resetForm() {
    store.dispatch(reset('VisitantesForm'));
    store.dispatch({type: types.CLEAR_VISITANTES});
}
```

Ilustración 70 - resetForm()

El resultado visual de este componente es el siguiente:

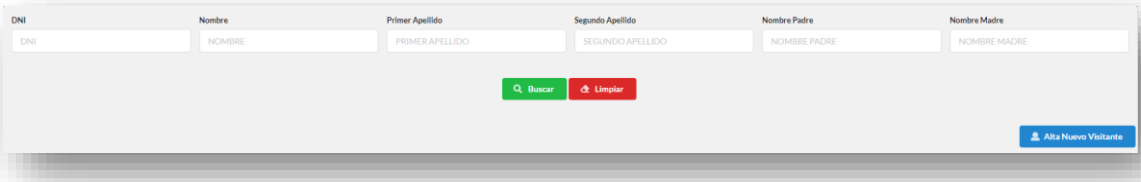


Ilustración 71 - Formulario de búsqueda de visitantes

El botón “Alta nuevo visitante” cambia la ruta de la aplicación a /newVisitante, que cargará la vista NewVisitante. Esta vista carga el componente NewVisitanteForm.

```
    render() {
        if (!isLoggedUser()) {
            return <Navigate to='/' />
        }
        return (
            <div>
                <NewVisitanteForm />
            </div>
        );
    }
}
```

Ilustración 72 - NewVisitante.jsx

En este componente nos encontramos con un formulario similar al de VisitantesForm, con los campos necesarios para dar de alta un nuevo visitante y un botón guardar, que lanza la petición al backend con la información del visitante para que se cree el nuevo registro. El resultado visual de esta vista es el siguiente:

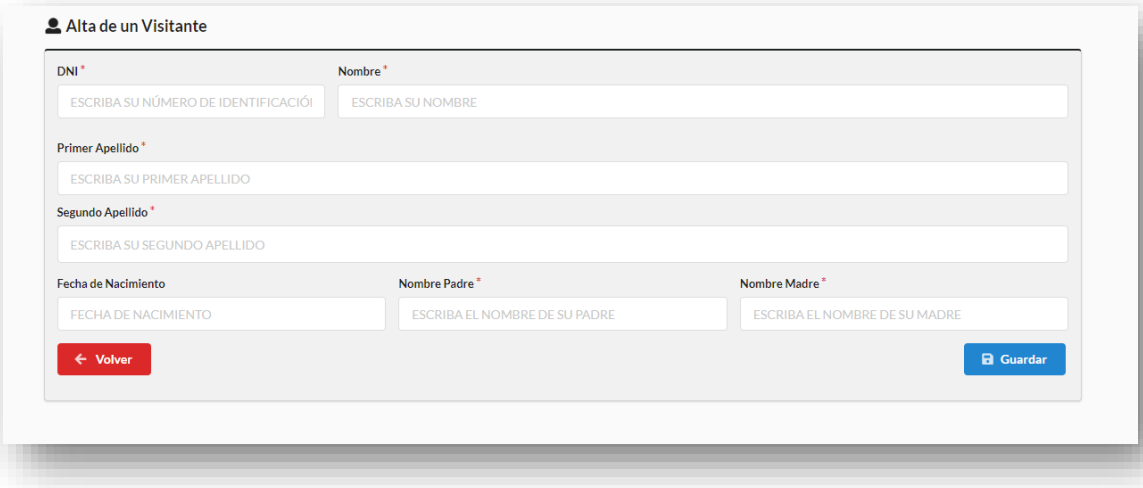


Ilustración 73 - Pantalla alta de un visitante

5.2.2.7 VisitantesTable.jsx

La otra parte que se carga en la vista de Visitantes es el componente VisitantesTable, que implementará la tabla en la que se mostrarán los registros de visitantes recuperados de base de datos.

```jsx
return (
    <div>
        <ReactTable
            className='-striped -highlight'
            data={visitantes}
            columns={columns}
            defaultSorted={sort}
            defaultPageSize={defaultPageSize}
            onPageSizeChange={this.onPageSizeChange}
            style={{ height: '550px' }}
            pageSize={pageSize}
            showPaginationTop
            showPaginationBottom
            previousText='<< Anterior'
            nextText='Siguiente >>'
            loadingText='Cargando...'
            noDataText='No se ha encontrado ningún visitante que coincida con los criterios de búsqueda'
            pageText='Pagina'
            ofText='de'
            rowsText='Visitantes'
            defaultFilterMethod={(filter, row) =>
                String(row[filter.id]).normalize('NFD').replace(/[\u0300-\u036f]/g, "")
                    .search(new RegExp(filter.value.normalize('NFD').replace(/[\u0300-\u036f]/g, ""), "i")) !== -1}
        />
    </div>
)
```

Ilustración 74 - VisitantesTable.jsx

Tal y como está diseñada la tabla, mostrará los datos almacenados en la variable global “visitantes”, que estará vacía en caso de no haber realizado ninguna búsqueda o en caso de haber pulsado sobre el botón “limpiar”. Las columnas que mostrará la tabla son las definidas en la variable columns, que serán diferentes dependiendo de la tabla que se esté mostrando.

```jsx
const columns = [
    {
        Header: row => (<strong>DNI</strong>),
        accessor: 'dni',
        filterable: true,
        width: 200,
        Cell: row => (<label title={row.value.toUpperCase()}><strong>{row.value.toUpperCase()}</strong></label>),
        Filter: ({ filter, onChange }) =>
            <Input style={{ width: '100%' }} onChange={(event) => onChange(event.target.value)} placeholder='DNI...' />
    }, {
        Header: row => (<strong>Nombre</strong>),
        accessor: 'nombre',
        filterable: true,
        width: 240,
        Cell: row => (
            <label title={row.value.toUpperCase()}><strong>{row.value.toUpperCase()}</strong></label>
        ),
        Filter: ({ filter, onChange }) =>
            <Input style={{ width: '100%' }} onChange={(event) => onChange(event.target.value)} placeholder='Nombre...' />
    }, {
        Header: row => (<strong>Primer Apellido</strong>),
        accessor: 'primerApellido',
        filterable: true,
        width: 240,
        Cell: row => (
            <label title={row.value.toUpperCase()}><strong>{row.value.toUpperCase()}</strong></label>
        ),
        Filter: ({ filter, onChange }) =>
            <Input style={{ width: '100%' }} onChange={(event) => onChange(event.target.value)} placeholder='Primer Apellido...' />
    }, {
        Header: row => (<strong>Segundo Apellido</strong>),
        accessor: 'segundoApellido',
        filterable: true,
        width: 240,
        Cell: row => (
            <label title={row.value.toUpperCase()}><strong>{row.value.toUpperCase()}</strong></label>
        ),
        Filter: ({ filter, onChange }) =>
            <Input style={{ width: '100%' }} onChange={(event) => onChange(event.target.value)} placeholder='Segundo Apellido...' />
    }, {
        Header: row => (<strong>Nombre Padre</strong>),
        accessor: 'nombrePadre',
        filterable: true,
        width: 240,
        Cell: row => (
            <label title={row.value.toUpperCase()}><strong>{row.value.toUpperCase()}</strong></label>
        ),
```

Ilustración 75 - VisitantesTable - Columns I

```
}, {
    Header: row => (<strong>Nombre Madre</strong>),
    accessor: 'nombreMadre',
    filterable: true,
    width: 240,
    Cell: row => (
        <label title={row.value.toUpperCase()}><strong>{row.value.toUpperCase()}</strong></label>
    ),
    Filter: ({ filter, onChange }) =>
        <Input style={{ width: '100%' }} onChange={(event) => onChange(event.target.value)} placeholder='Nombre Madre...' />
}, {
    Header: row => (<strong>Acciones</strong>),
    id: 'buttons',
    width: 450,
    filterable: false,
    sortable: false,
    accessor: (row) => (row),
    Cell: row => (
        <div className="centered">
            <Button compact size='tiny' primary as={Link} to={'/edit-visitante/' + row.value.visitanteId}><Icon name='write' />Editar</Button>
            <Button compact size='tiny' negative onClick={() => this.submit(row.value.visitanteId)}><Icon name='trash' />Borrar</Button>
        </div>
    )
}
```

Ilustración 76 - VisitantesTable - Columns II

Al pulsar sobre el botón “Editar” se nos redirigirá a la ruta /edit-visitante/id correspondiente a la vista EditVisitante y donde el id será el identificador del usuario sobre el que vamos a realizar cambios. Al pulsar sobre el botón Borrar accederemos al método que manejará está acción, pasando como parámetro el id del visitante a eliminar.

```
confirmDelete = (id) => {
    store.dispatch({type: types.SET_SEARCHING_TRUE});
    store.dispatch({type: types.DELETE_VISITANTE, id: id});
}


submit = (id) => {
    confirmAlert({
        customUI: ({ onClose }) => (
            <CustomOverlay onClick={onClose}>
                <CustomConfirmAlert>
                    <h1>Borrar visitante</h1>
                    <p>¿Estás seguro de que deseas borrar este visitante?</p>
                    <ButtonGroup>
                        <Button onClick={() => { this.confirmDelete(id); onClose(); }}>Sí</Button>
                        <Button onClick={onClose}>No</Button>
                    </ButtonGroup>
                </CustomConfirmAlert>
            </CustomOverlay>
        )
    });
}
```

Ilustración 77 - VisitantesTable.jsx - submit

El método submit lanzará un componente confirmAlert que nos solicitará confirmación sobre la acción del borrado del visitante, y en caso de realizar dicha confirmación, el método confirmDelete lanzará la llamada de la store que lanza la petición de borrado de un visitante, pasando como parámetro el id de dicho visitante.

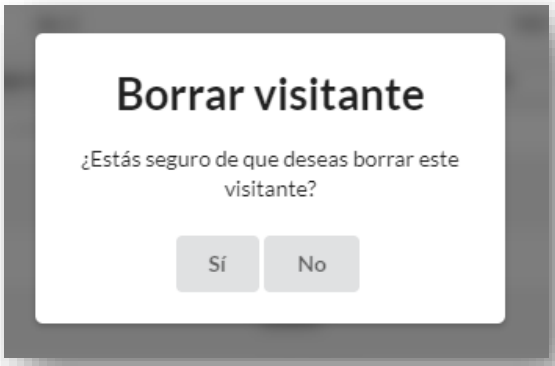


Ilustración 78 - Solicitud de confirmación de borrado

El resultado visual de la tabla mostrando datos de visitantes es la siguiente:

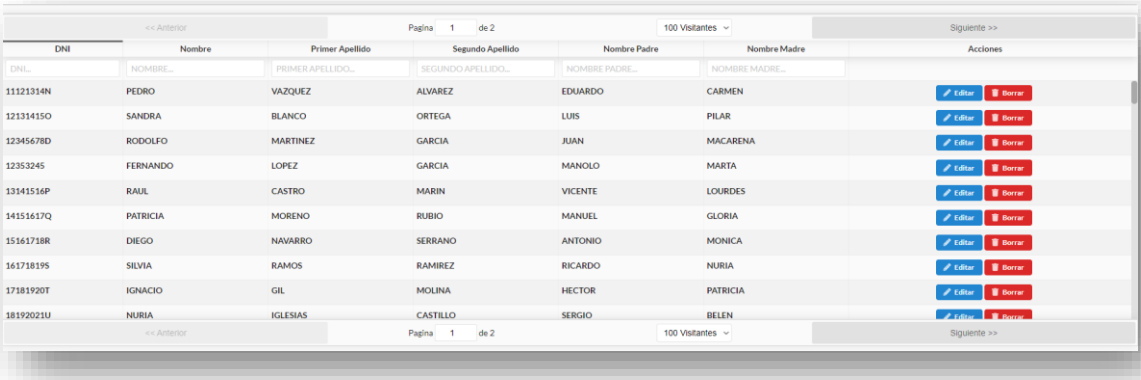


Ilustración 79 - Tabla de visitantes

Al pulsar sobre editar un visitante se nos redirigirá a la ruta que carga la vista de edición de un visitante (EditVisitante). Esta vista carga el componente intermedio VisitanteEditInitialValues, al que se le pasa el identificador del visitante a editar.

```
render(){

    if (!isLoggedUser()) {
        return <Navigate to='/' />
    }

    const { id } = this.props.router.params;
    return(
        <div>
            <VisitanteEditInitialValues id={id}/>
        </div>
    )
}
```

Ilustración 80 - EditVisitante.jsx

El componente intermedio VisitanteEditInitialValues sirve para recoger los datos del visitante a editar y plasmarlo sobre el siguiente componente, que es el componente que muestra el formulario de edición de visitante (VisitanteEditForm).

```
class VisitanteEditInitialValues extends Component {
    render(){
        return(
            <div>
                <VisitanteEditForm {...this.props}/>
            </div>
        )
    }
}

function mapStateToProps(state){
    return{
        visitante: state.visitante
    };
}

export default connect(mapStateToProps)(VisitanteEditInitialValues);
```

Ilustración 81 - VisitanteEditInitialValues.jsx

El componente VisitanteEditForm contiene un formulario idéntico al de creación de un nuevo visitante, con la diferencia que los valores iniciales que se muestran en los campos editables son los del visitante recuperado en el componente anterior. El resultado visual de este componente es el siguiente:

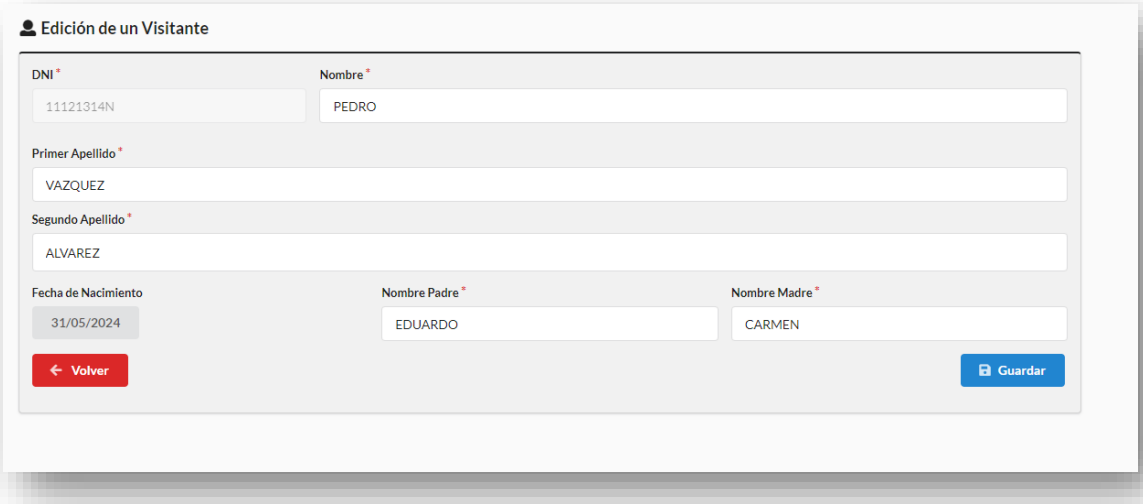


Ilustración 82 - Edición de un visitante

Esta es la implementación del módulo de visitantes, aunque como ya se ha indicado anteriormente, es totalmente extrapolable al resto de módulos ya que el funcionamiento es similar.

5.2.3 Desarrollos de las sagas y las apis

Las sagas son funciones que escuchan las acciones lanzadas por el store de Redux y construye una llamada API REST en función de lo obtenido. A continuación, se explica el funcionamiento y la implementación de una saga tomando como ejemplo la saga de visitantes.

```
function* watchGetAllVisitantes() {
    yield takeEvery(types.GET_ALL_VISITANTES, getAllVisitantes);
}


function* watchGetVisitantesFilter() {
    yield takeEvery(types.GET_VISITANTES_FILTER, getVisitantesFilter);
}

function* watchPostVisitante() {
    yield takeEvery(types.POST_VISITANTE, postVisitante);
}

function* watchGetVisitante() {
    yield takeEvery(types.GET_VISITANTE, getVisitante);
}

function* watchPutVisitante() {
    yield takeEvery(types.PUT_VISITANTE, putVisitante);
}

function* watchDeleteVisitante() {
    yield takeEvery(types.DELETE_VISITANTE, deleteVisitante);
}
```

Ilustración 83 - watchers de visitanteSaga.js

Como se puede observar, el visitanteSaga está continuamente escuchando hasta recibir una de las acciones lanzadas por la store relacionada con visitantes. Como por ejemplo GET_ALL_VISITANTES, esta acción indicada que se quiere recuperar todos los visitantes, por lo que la saga redirige a la función getAllVisitantes.

```
function*  getAllVisitantes(action){
    const { visitantes, error } = yield call(visitanteApi.getAllVisitantes);
    if(visitantes){
        yield put({type: types.GET_ALL_VISITANTES_SUCCESS, visitantes});
        yield put({type: types.SET_SEARCHING_FALSE});
    }else{
        if(error.status === 404){
            toast.error("No se ha obtenido ningún visitante");
        }else{
            toast("Ha ocurrido un error inesperado")
        }
        yield put({type: types.SET_SEARCHING_FALSE})
    }
}
```

Ilustración 84 - getAllVisitantes

Esta función llama a la api de visitantes (visitanteApi) para que lance la petición HTTP que solicite recuperar la información de todos los visitantes de la base de datos. En caso de que la api retorne una variable visitantes indicará que la petición ha ido bien, por lo que se desactiva el Dimmer de "cargando" y se lanza la acción que recogerá el reducer para poblar la variable global visitantes con la información devuelta por la solicitud HTTP. En caso de que la api retorne un error se tratará dependiendo del código del error y se informará al usuario a través de mensajes Toast.

La api se encarga de construir la llamada HTTP, construyendo la URL correspondiente a la llamada que se quiere hacer, indicando el tipo de operación y pasando unas cabeceras predefinidas y un cuerpo de petición en caso de ser necesario por ejemplo en una petición POST o PUT.

```
/**
 * Metodo:      getAllVisitantes
 * Funcion:     Obtener todos los visitantes de la aplicacion
 * Parametros:
 * Return:      Lista visitantes
 */
static getAllVisitantes() {
    return fetch(Window.URI + 'visitantes/all', {
        method: 'get',
        headers: JSON.parse(commonApi.getHeaders())
    }).then(response => handleApiError(response, bodyType.JSON))
      .then(visitantes => ({visitantes}))
      .catch(error => (commonApi.parseError(error))
    );
}
```

Ilustración 85 - Función getAllVisitantes de la api de visitantes

Dependiendo de la respuesta recibida la api la tratará para devolvérsela a la saga y que está actúe en consecuencia.

5.2.4 Desarrollo de los reducers

Los reducers son funciones de la store que determinan cómo se actualiza el estado de la aplicación en respuesta a una acción, es decir, toma el estado actual de una variable global y una acción como argumentos y devuelve un nuevo estado, que puede ser, por ejemplo, la población de la variable visitantes con todos los recursos de tipo visitante que han sido devueltos por la capa de aplicación.

```
export default function visitantesReducer(state = initialState.visitantes, action) {
    switch(action.type) {
        case types.GET_ALL_VISITANTES_SUCCESS:
            return action.visitantes
        case types.CLEAR_VISITANTES:
            return initialState.visitantes
        case types.GET_VISITANTES_FILTER_SUCCESS:
            return action.visitantes
        default:
            return state;
    }
}
```

Ilustración 86 - visitantesReducer

En el caso mostrado en la ilustración anterior se observa como dependiendo de la acción que recibe el reducer de visitantes va a tratar el estado global de la variable visitantes de una forma o de otra. En caso de recibir la acción GET_ALL_VISITANTES_SUCCESS o GET_VISITANTES_FILTER_SUCCESS, que indican que la llamada HTTP que solicita registros visitantes ha ido correctamente, puebla la variable visitantes con la información que haya recibido en el action. En caso de recibir la acción CLEAR_VISITANTES, lanzada por ejemplo al pulsar sobre el botón limpiar de la pantalla de visitantes, devolverá el valor inicial a la variable visitantes, que será vacío.

5.3 Caso integración entre capas

En este apartado se va a mostrar un flujo completo explicando el proceso que sufren las tres capas durante el mismo. Como ejemplo vamos a poner la situación en la que un usuario pulsa sobre el botón “Buscar” de la pantalla de Visitantes sin haber realizado ningún filtrado.

En el instante inicial el usuario se encuentra en la pantalla de visitantes y pulsa sobre el botón “Guardar” sin haber cumplimentado ninguno de los campos de filtrado de búsqueda.

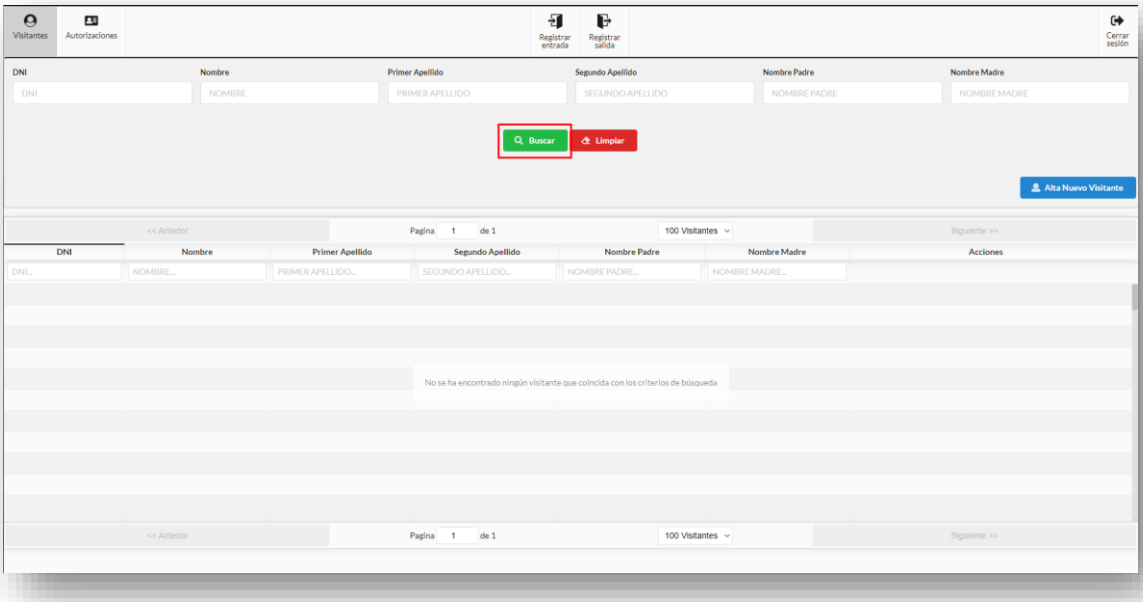


Ilustración 87 - Pulsa botón "Buscar"

Al pulsar sobre ese botón, en el componente VisitantesForm.jsx se ejecuta el método onSubmit, lanzado por el handler que maneja lo que ocurre en el formulario cuando se hace submit del mismo.

```
const onSubmit = (values) => {
    store.dispatch({ type: types.SET_SEARCHING_TRUE});
    store.dispatch({ type: types.CLEAR_VISITANTES});
    if((values.dni === undefined || values.dni === "") && (values.name === undefined || values.name === "") && (values.surname1 === undefined || values.surname1 === "") &&
    (values.surname2 === undefined || values.surname2 === "") && (values.fatherName === undefined || values.fatherName === "") && (values.motherName === undefined || values.motherName === "")){
        store.dispatch({ type: types.GET_ALL_VISITANTES, model: values});
    }else{
        store.dispatch({ type: types.GET_VISITANTES_FILTER, model: values});
    }
}

return (
    <Segment color="black">
        <Form
            name="VisitantesForm"
            onSubmit={handleSubmit(onSubmit)}
        >
```

Ilustración 88 - VisitantesForm.jsx - onSubmit

En el método onSubmit la store lanza la petición que activa el Dimmer de “Cargando” con la sentencia `store.dispatch({ type: types.SET_SEARCHING_TRUE});`

La store también lanza la acción de limpieza de la variable global visitantes con la sentencia `store.dispatch({ type: types.CLEAR_VISITANTES});` Hecho esto y una vez comprobado que ningún filtro ha sido cumplimentado, la store lanza la acción que solicita la recogida de todos los visitantes de la aplicación mediante la sentencia `store.dispatch({ type: types.GET_ALL_VISITANTES, model: values});`

Esta acción es recogida por la visitanteSaga, que una de las acciones que está escuchando es GET_ALL_VISITANTES.

```
function* watchGetAllVisitantes() {
    yield takeEvery(types.GET_ALL_VISITANTES, getAllVisitantes);
}
```

Ilustración 89 - Escucha GET_ALL_VISITANTES

Al escuchar esta acción lanza la función getAllVisitantes:

```
function*  getAllVisitantes(action){
    const { visitantes, error } = yield call(visitanteApi.getAllVisitantes);
    if(visitantes){
        yield put({type: types.GET_ALL_VISITANTES_SUCCESS, visitantes});
        yield put({type: types.SET_SEARCHING_FALSE});
    }else{
        if(error.status === 404){
            toast.error("No se ha obtenido ningún visitante");
        }else{
            toast("Ha ocurrido un error inesperado")
        }
        yield put({type: types.SET_SEARCHING_FALSE})
    }
}
```

Ilustración 90 - visitantesSaga - getAllVisitantes

Esta función llama al método getAllVisitantes de la api de visitantes (visitantesApi), que construira la llamada HTTP.

```
/**
 * Metodo:      getAllVisitantes
 * Funcion:     Obtener todos los visitantes de la aplicacion
 * Parametros:
 * Return:      Lista visitantes
 */
static getAllVisitantes() {
    return fetch(Window.URI + 'visitantes/all', {
        method: 'get',
        headers: JSON.parse(commonApi.getHeaders())
    }).then(response => handleApiError(response, bodyType.JSON))
      .then(visitantes => ({visitantes}))
      .catch(error => (commonApi.parseError(error))
    );
}
```

Ilustración 91 - visitanteApi - getAllVisitantes

Una vez construida la petición la lanza. En este caso la llamada será http://localhost:8080/api/visitantes/all, que será recogida por el controller de visitantes de la capa de aplicación.

```
@RestController
@CrossOrigin
@RequestMapping(value = "api/visitantes")
public class VisitanteController extends BaseController{

    @GetMapping(value = "/all")
    public List<Visitante> getVisitante() throws visitantesNotFoundException {

        List<Visitante> lista = visitanteRepository.findAll();

        if(lista.isEmpty()) {
            throw new visitantesNotFoundException();
        }

        return lista;

    }
```

Ilustración 92 - VisitanteController - getVisitante()

El controlador de visitantes recogerá la petición, contactará con la capa de datos solicitándole todos los recursos de tipo visitantes, y lo devolverá en forma de lista como respuesta a la petición HTTP.

```
function*  getAllVisitantes(action){
    const { visitantes, error } = yield call(visitanteApi.getAllVisitantes);
    if(visitantes){
        yield put({type: types.GET_ALL_VISITANTES_SUCCESS, visitantes});
        yield put({type: types.SET_SEARCHING_FALSE});
    }else{
        if(error.status === 404){
            toast.error("No se ha obtenido ningún visitante");
        }else{
            toast("Ha ocurrido un error inesperado")
        }
        yield put({type: types.SET_SEARCHING_FALSE})
    }
}
```

Ilustración 93 - saga recibe respuesta

Una vez la api recibe la respuesta correcta por parte de la capa de aplicación, esta le pasa los valores devueltos a la saga, que al recibirlos lanza las acciones que desactivan el Dimmer y la que recibe el reducer para poblar la variable global.

```
export default function visitanteReducer(state = initialState.visitante, action) {
    switch(action.type) {
        case types.GET_VISITANTE_SUCCESS:
            return action.visitante
        case types.PUT_VISITANTE_SUCCESS:
            return action.visitante
        default:
            return state;
    }
}
```

Ilustración 94 - visitanteReducer recibe SUCCESS

El reducer al recibir la el SUCCESS actualiza el estado de la variable global, asignándole los datos recibidos de la saga.

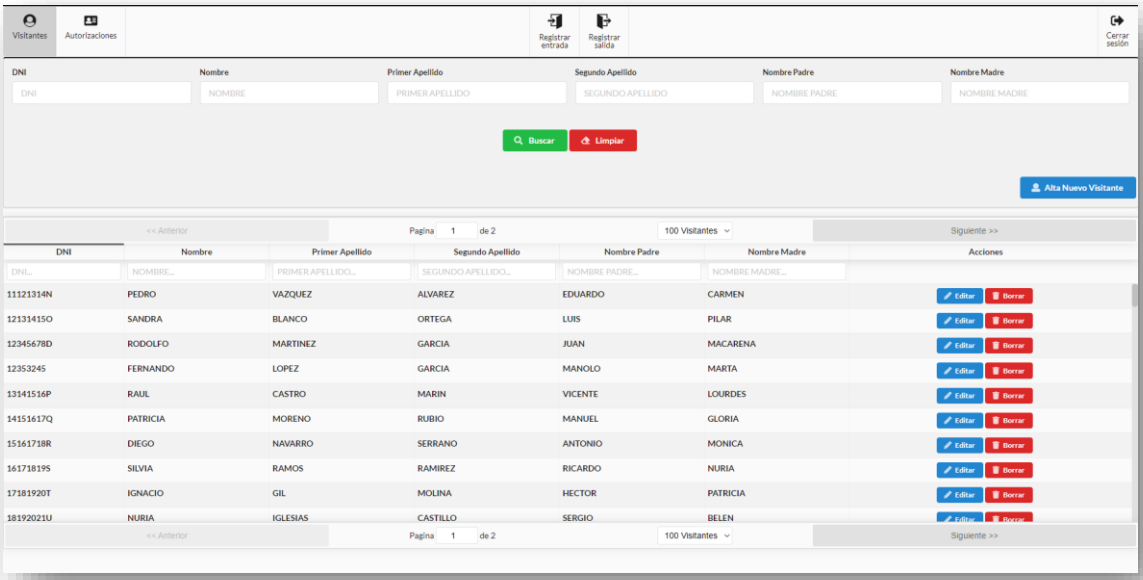


Ilustración 95 - Tabla visitantes con datos.

En ese momento, que la tabla de visitantes está diseñada para mostrar los datos de la variable visitantes, que ya está poblada, muestra la información de todos los visitantes.

5.4 Caso de uso

A continuación, se muestra el caso de uso que muestra el flujo esperado de la aplicación.

Ilustración 96 - Pantalla Login

Al arrancar la aplicación el usuario se encuentra con la pantalla de login. Al no tener cuenta pulsa sobre “Registrarse” que le redirigirá al formulario de creación de un nuevo usuario.

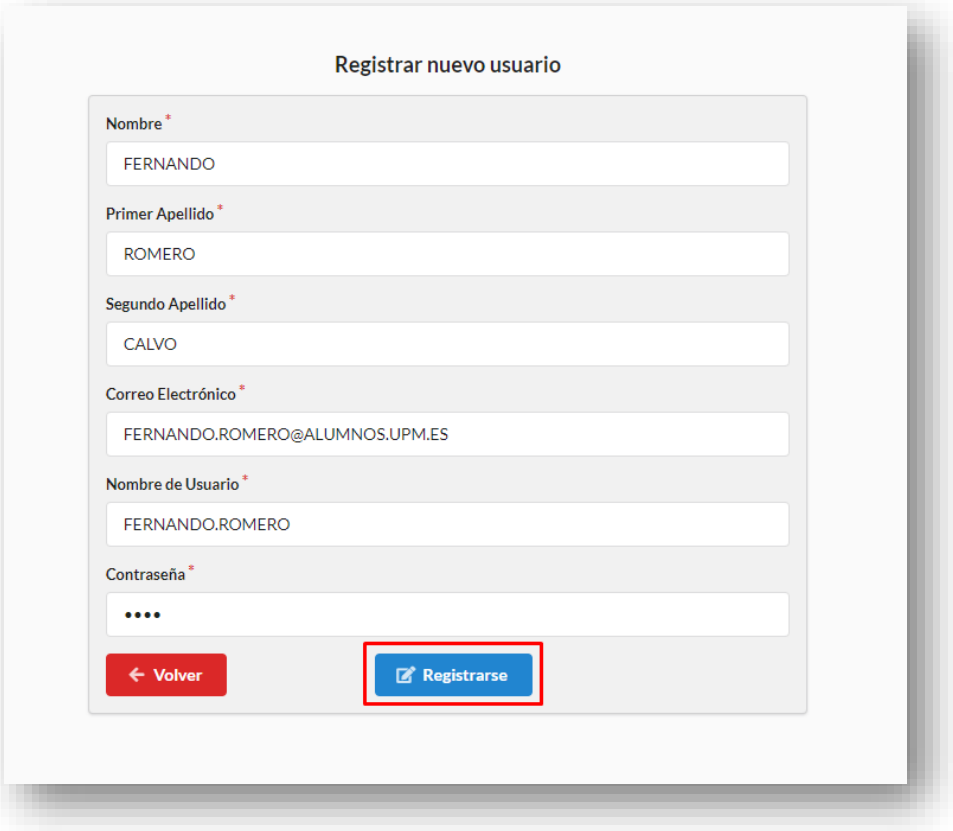


Ilustración 97 - Pantalla registro nuevo usuario

El usuario cumplimenta los datos de registro y pulsa sobre el botón "Registrarse", completándose así el registro y siendo redirigido automáticamente a la pantalla de inicio.

Ilustración 98 - Pantalla de inicio

El usuario pulsa sobre la opción "Visitantes" del menú superior, siendo redirigido a la pantalla de visitantes.

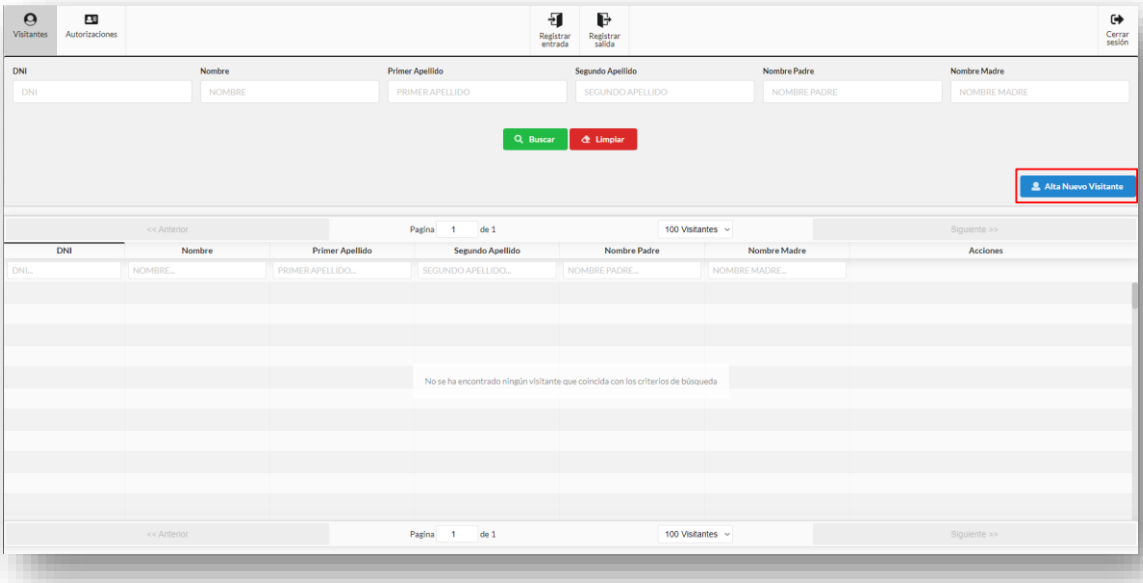


Ilustración 99 - Pantalla visitantes

En la pantalla de visitantes pulsa sobre “Alta nuevo visitante” para dar de alta un nuevo visitante en la aplicación. La aplicación le redirige automáticamente al formulario de creación de un nuevo visitante.

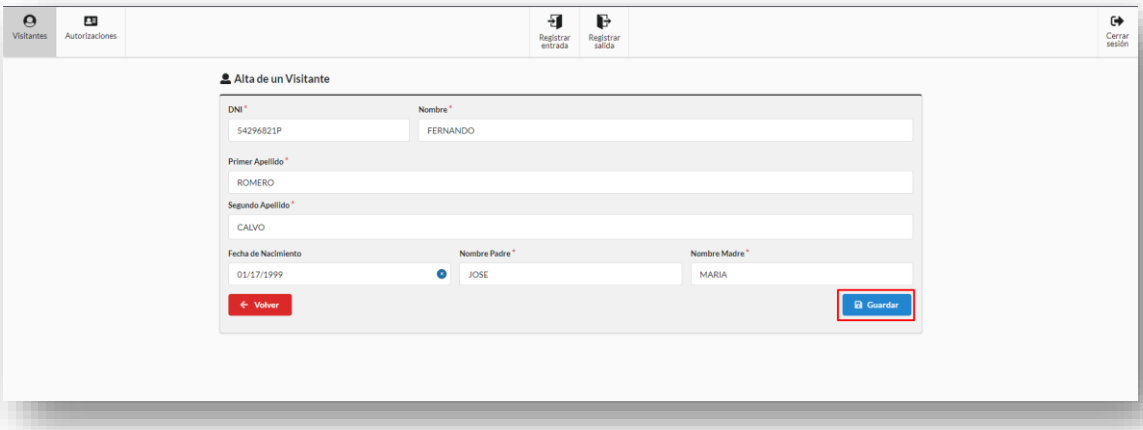


Ilustración 100 - Formulario creación nuevo visitante

El usuario cumplimenta el formulario y pulsa sobre el botón “Guardar”, dando de alta el visitante y siendo redirigido a la pantalla de visitantes.

El usuario realiza una búsqueda usando el filtro de DNI esperando obtener la información del visitante que acaba de crear.

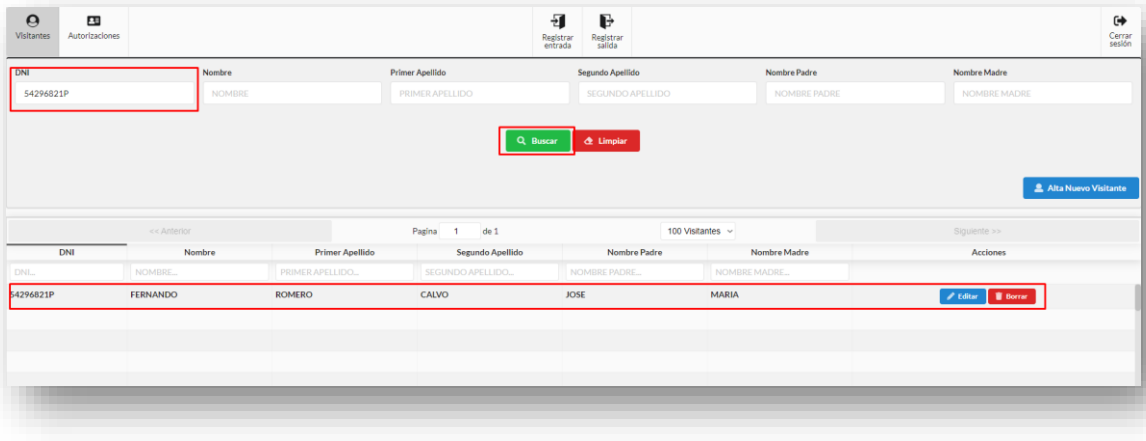


Ilustración 101 - Búsqueda visitante

El usuario se dirige a la pantalla de autorizaciones, donde pulsa en alta nueva autorización que le redirigirá a la pantalla donde se mostrará el formulario de creación de una nueva autorización.

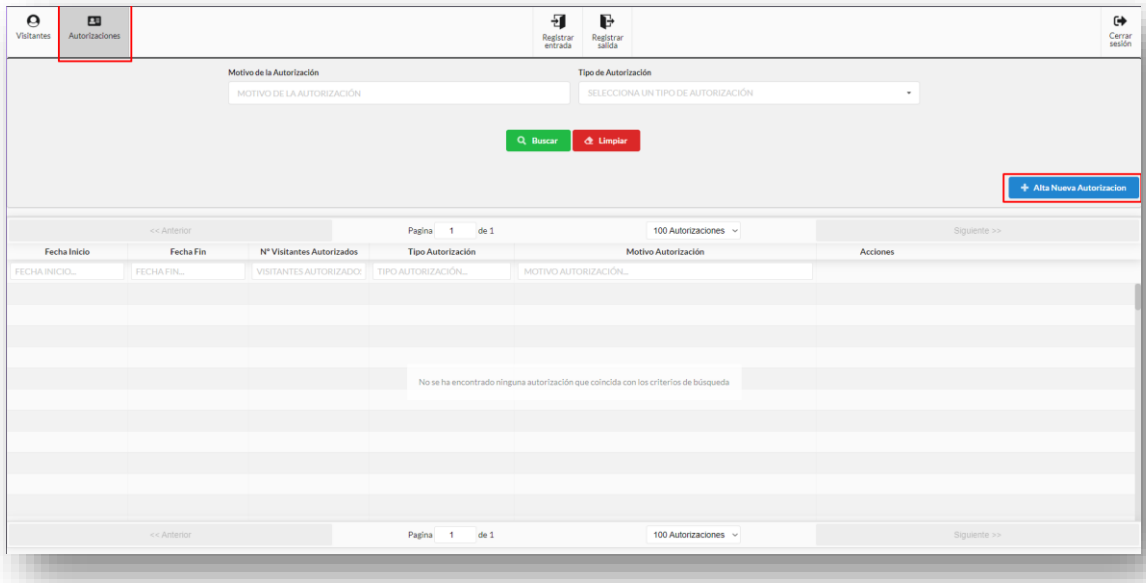


Ilustración 102 - Pantalla autorización

El usuario cumplimenta el formulario y pulsa sobre el botón guardar.

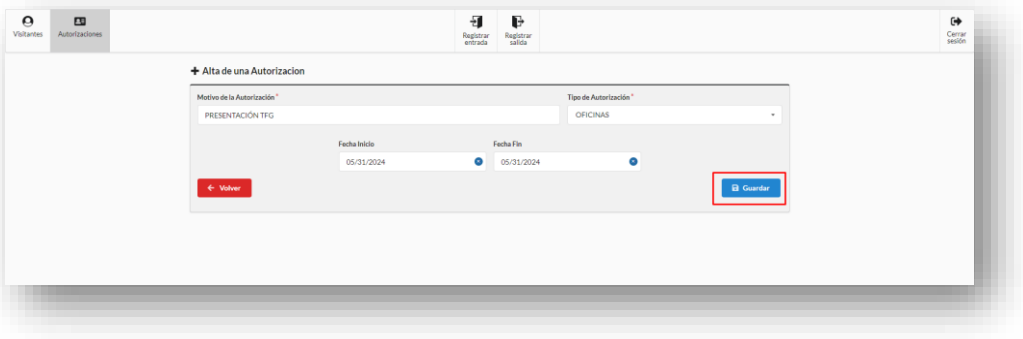


Ilustración 103 - Formulario creación autorización

Al ser creada la autorización, se habilita la tabla que permite asignar visitantes a la misma. El usuario busca el DNI del visitante que ha creado y lo asigna a la autorización pulsando en “Asignar”

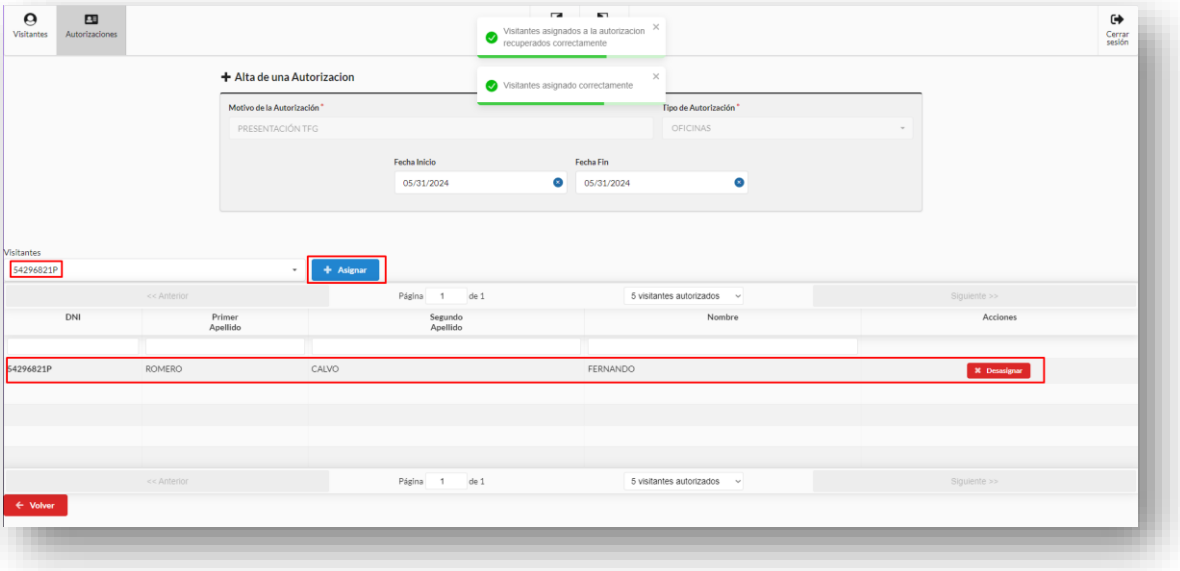


Ilustración 104 - Asignación visitante-autorizacion

El usuario se dirige al módulo de registrar entrada, donde se muestran todos los visitantes autorizados, busca el visitante que ha creado y le registra la entrada.

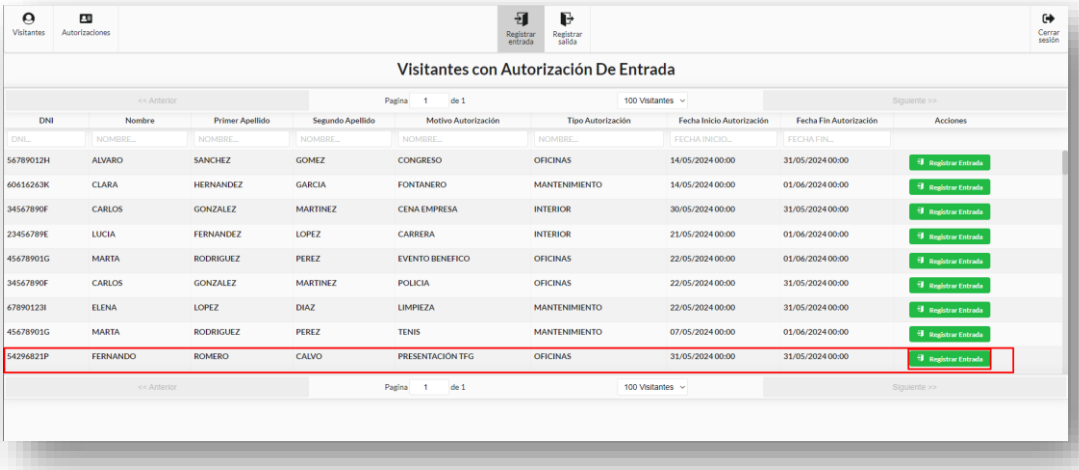


Ilustración 105 - Registrar entrada

Al registrar la entrada del visitante este desaparece de la tabla. Al rato usuario accede a la pantalla de registrar salida para registrar la salida del visitante. Como el visitante está dentro del control aparecerá en la tabla de esta pantalla y el usuario podrá registrar su salida pulsando en el botón “Registrar salida”.

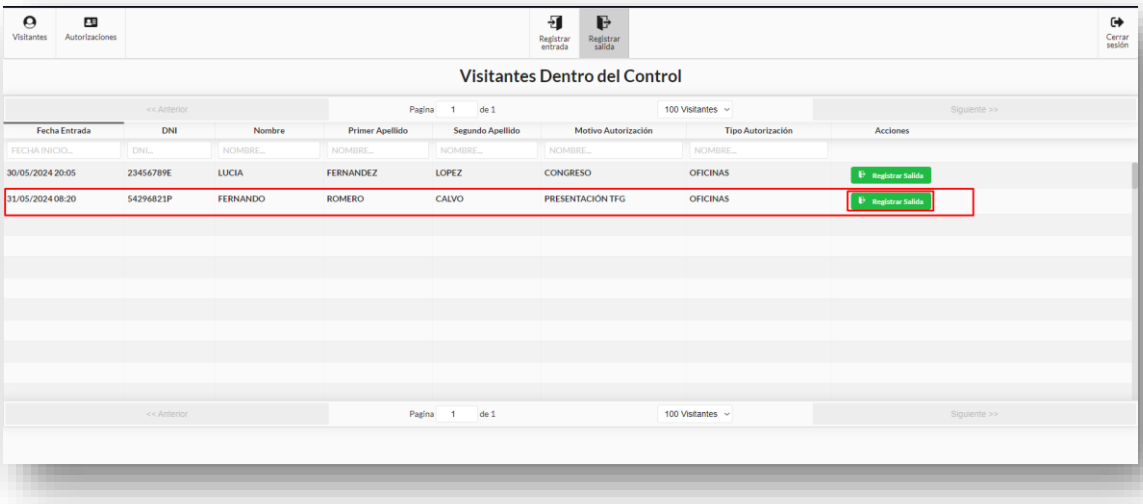


Ilustración 106 - Registrar salida

6 Objetivos de Desarrollo Sostenible 2030

Creo que este proyecto puede contribuir en gran parte a dos de los 17 objetivos de desarrollo sostenible 2030, en concreto al objetivo 9: Industria, Innovación e infraestructura y al objetivo 11: ciudades y comunidades sostenibles.

6.1 Contribución al objetivo 9: Industria, Innovación e Infraestructura.

- **Promoción de la innovación tecnológica**: la aplicación constituye una solución tecnológica innovadora para la gestión de registros de accesos a instituciones. Al implementar esta herramienta digital, se está fomentando la innovación en el ámbito de la gestión de acceso, utilizando tecnología de la información y comunicación (TIC) para optimizar los procesos y mejorar la eficiencia en la administración de registros.
- **Facilitación del Acceso a Servicios y Recursos**: Al proporcionar una plataforma digital para la gestión de accesos, la aplicación facilita el acceso a servicios y recursos en diversas instituciones. Esto es especialmente relevante en entornos donde la gestión manual de registros puede ser ineficiente o limitada. La tecnología digital ayuda a superar barreras de acceso y mejora la experiencia de usuarios y administradores.
- **Infraestructura Digital Avanzada**: La implementación de esta aplicación implica el desarrollo de una infraestructura digital avanzada que permite la captura, almacenamiento y gestión eficiente de datos relacionados con el acceso a las instituciones. Esto contribuye al desarrollo de infraestructuras tecnológicas sólidas y modernas que son fundamentales para el progreso económico y social.
- **Promoción de la Eficiencia y la productividad**: La automatización de procesos a través de esta aplicación contribuye a aumentar la eficiencia y la productividad en la gestión de accesos. Al eliminar tareas manuales y propensas a errores, se optimizan los recursos y se reduce el tiempo y el esfuerzo dedicados a la administración de registros. Esto, a su vez, libera recursos que pueden destinarse a otras áreas de desarrollo.

6.2 Contribución al objetivo 11: Ciudades y Comunidades Sostenibles

- **Mejora de la Seguridad y la Gestión Urbana**: La gestión eficaz de los registros de accesos es esencial para garantizar la seguridad y el orden en las instituciones y comunidades urbanas. Esta aplicación ayuda a mejorar la gestión urbana al proporcionar una herramienta que facilita el control y la supervisión de quienes acceden a determinados espacios. Esto contribuye a crear entornos más seguros y ordenados.
- **Promoción de la Transparencia y la Participación Ciudadana**: Al proporcionar una plataforma digital para la gestión de accesos, la aplicación promueve la transparencia y la participación ciudadana en la administración de instituciones y espacios públicos. Los registros digitales pueden ser fácilmente accesibles y auditables, lo que aumenta la confianza en las instituciones y permite una mayor participación de la comunidad en la toma de decisiones.

- **Reducción de la Huella Ambiental**: La transición de sistemas de gestión de registros basados en papel a soluciones digitales tiene el potencial de reducir la huella ambiental asociada con la producción y el manejo de documentos físicos. Al eliminar o reducir la necesidad de papel y otros recursos materiales, esta aplicación contribuye a la conservación de recursos naturales y a la mitigación del impacto ambiental.
- **Mejora de la Calidad de Vida Urbana**: Una gestión eficiente de accesos a instituciones y espacios públicos puede mejorar la calidad de vida urbana al reducir la congestión, facilitar el flujo de personas y garantizar un entorno más seguro y ordenado. La aplicación contribuye a crear comunidades más sostenibles y habitables al promover prácticas de gestión urbana inteligente y eficiente.

7 Conclusiones y líneas futuras

Con el desarrollo de esta aplicación se ha logrado ofrecer una solución innovadora y eficiente que no solo mejora la seguridad a la hora de dar acceso a persona a ciertas instituciones, sino que también optimiza la experiencia del usuario y aumenta la productividad de la organización que la utilice.

La implementación de una aplicación web basada en una arquitectura moderna y escalable ha permitido cumplir con los objetivos planteados al inicio del proyecto. La combinación de tecnologías avanzadas, como React.js para el frontend y Spring Boot con java para el backend, ha garantizado un gran rendimiento y una experiencia de usuario intuitiva y fluida. Además, el uso de APIs REST para la comunicación entre las diferentes capas de la arquitectura de la aplicación ha proporcionado una integración eficiente.

El uso de herramientas como Hibernate para la comunicación con la base de datos ha proporcionado un manejo eficiente de los datos, asegurando la integridad y la seguridad de estos.

Mirando hacia el futuro, y gracias a la tecnología empleado, creo que esta aplicación da las herramientas para ser altamente escalable y evolucionable, desde aspectos básicos que se podrían mejorar como la implementación de mejores capas de seguridad entre los usuarios, con la implementación de roles dentro de la aplicación o un módulo de gestión de usuarios, como la exportación de los datos recogidos en las distintas tablas a formatos Excel o pdf para tener un acceso todavía más fácil a ellos.

La tecnología empleada permite infinidad de soluciones, como por ejemplo la integración con componentes externos, como la autenticación de usuarios mediante un sistema LDAP o la recepción de datos de forma externa mediante la habilitación de servicios web que permitan importar datos en la base de datos de esta aplicación. A parte de la exportación de información a archivos Excel o pdf también se podría integrar con alguna aplicación de generación de documentos como Pentaho-Desing reporter, para crear documentos diseñados a medida que muestren la información tal y como la solicite el cliente, como, por ejemplo, la creación de un documento generado dinámicamente con la información de una autorización y los visitantes asociados a la misma.

Otra vía de evolución sería la diferenciación de diferentes puestos de control dentro de una misma institución, teniendo que ir pasándolos de forma progresiva o estando autorizados a acceder solo hasta cierto punto.

En conclusión, el desarrollo de esta aplicación ha sido una experiencia profundamente enriquecedora a nivel personal. Desde el inicio, el proyecto planteaba desafíos que requerían una combinación de habilidades técnicas, así como comprensión profunda de las necesidades de seguridad y un profundo conocimiento de la tecnología y esto ha permitido un desarrollo personal tanto a nivel técnico y profesional como a nivel personal.

8 Anexo

8.1 Bibliografía

Publicaciones utilizadas en el estudio y desarrollo del trabajo.

[1] "Desarrollo de aplicaciones web". Juanda.gitbook.io. https://juanda.gitbooks.io/webapps/content/api/arquitectura-api-rest.html

[2] "API: qué es y para qué sirve". Xataka. https://www.xataka.com/basics/api-que-sirve

[3] "¿Qué es una API REST?. IBM. https://www.ibm.com/es-es/topics/rest-apis

[4] "Qué es una API REST, para qué sirve y ejemplos". blog.hubspot.es. https://blog.hubspot.es/website/que-es-api-rest

[5] "¿Qué es Java Spring Boot?". IBM. https://www.ibm.com/es-es/topics/java-spring-boot

[6] "Qué es MySQL: Características y ventajas". openwebinars. https://openwebinars.net/blog/que-es-mysql/

[7] "Qué es React: definición, características y funcionamiento". Hostinger. https://www.hostinger.es/tutoriales/que-es-react#Caracteristicas_de_React

[8] "Qué es SonarLint y cómo mejora la calidad de tu código". excentia.es. https://www.excentia.es/que-es-sonarlint-y-como-mejora-calidad-codigo

[9] "Explora la arquitectura de tres niveles". community.fs.com. https://community.fs.com/es/article/explore-three-level-architecture.html

9 Tabla de ilustraciones

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Mon Jun 03 21:42:20 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)