Universitat Oberta de Catalunya

uoc.edu

Asistente para la planificación de horarios

Pablo Díaz Muñiz
Grado de Ingeniería Informática
Desarrollo Web

Gregorio Robles Martínez
Santi Caballé Llobet
David García Solórzano

Junio de 2023

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Asistente para la planificación de horarios*
Nombre del autor:	*Pablo Díaz Muñiz*
Nombre del consultor/a:	*Gregorio Robles Martínez*
Nombre del PRA:	*Santi Caballé Llobet–David García Solórzano*
Fecha de entrega:	06/2023
Titulación:	*Grado de Ingeniería Informática*
Área del Trabajo Final:	*Desarrollo Web*
Idioma del trabajo:	*Castellano*
Palabras clave	*Horario, Web, REST.*

Resumen del Trabajo:

La elaboración de horarios es una tarea de especial relevancia en un amplio espectro de empresas. Si no se cuenta con las herramientas adecuadas esta labor puede llegar a consumir excesivo tiempo sin llegar a alcanzarse un resultado óptimo. Este es el motivo de la puesta en marcha del presente proyecto.

La aplicación que se ha creado pretende ser un apoyo para managers al implementar una serie de características que mejoran la visibilidad del resultado de una planificación de horarios a medida que se trabaja sobre ella.

Para el desarrollo del proyecto se ha seguido una metodología iterativa, con el fin de poner de manifiesto cualquier problema que pueda surgir en el menor tiempo posible. Además, se han elegido tecnologías de desarrollo de extensa implantación que cuentan con una gran variedad de recursos disponibles y amplio soporte de la comunidad de desarrolladores.

El producto obtenido, una aplicación web basada en una API REST como *backend* y una *Single Page Application* como *frontend*, cumple con los requisitos establecidos y es totalmente funcional, pudiendo utilizarse para el fin para el que ha sido diseñado.

Una vez finalizado el trabajo, ha quedado patente la importancia de las labores de obtención de requisitos y de planificación, así como la gestión de la comunicación con los *stakeholders* durante todo el desarrollo. Estos son

aspectos clave para una correcta toma de decisiones que garanticen el éxito final del proyecto.

Abstract:

Staff scheduling is a task of special relevance in a wide spectrum of companies. Without the appropriate tools this work would consume an excessive amount of time, not reaching an optimum result. This is the main aim of the implementation of this project.

The application that has been created is intended to be a useful support for managers through the implementation of several features that improve the visibility of a schedule planning while working on it.

An iterative methodology has been applied for the development of the project, in order to highlight any problem that may arise in the shortest possible time. Furthermore, widely implanted development technologies, which feature a wide variety of resources and support from the developer community, have been chosen.

The result of this project, a web application based on an API REST, as backend, and a Single Page Application, as frontend, meets the requirements, is completely functional and can be used for the purpose for which it was designed.

Once the work is finished, it has become clear the importance of requirements and planning tasks, as well as communication management with stakeholders throughout the development process. These are essential keys to make accurate decisions that ensure the eventual success of the project.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

La optimización de los recursos disponibles es una cuestión central en la toma de decisiones de cualquier empresa. Por ello, la elaboración de horarios es una tarea crítica en muchos ámbitos de negocio.

Pensemos por un momento en un ejemplo, como puede ser una empresa que desarrolla su actividad en el sector *retail*, y sigamos el circuito de los productos que se comercializan en una unidad de negocio concreta:

1. La mercancía llega a unas horas determinadas de unos días determinados, lo que generará unas necesidades extraordinarias de personal para el departamento de logística.
2. En las áreas comerciales, una vez que la mercancía haya sido tratada por el departamento de logística y esté disponible para reponer los lineales, se deberá contar con el número de empleados acorde a la situación descrita.
3. Finalmente, el público accederá en diferente número al establecimiento según la hora del día, el día de la semana y la época del año. Así, se generarán nuevas necesidades de personal para atender a estos clientes en los departamentos comerciales y de atención al cliente.

Todas estas consideraciones, entre otras muchas, tienen un fuerte impacto en la calidad de los servicios ofrecidos por la empresa y la forma en la que se gestione la contratación de nuevos empleados: perfiles necesarios, tipos de contrato, campañas especiales y otras cuestiones que han de tenerse en cuenta para el funcionamiento óptimo de la empresa. Hemos de tener en cuenta que la masa salarial es una de las partidas presupuestarias más elevadas en una amplia tipología de empresas y de su correcta gestión dependerá, en gran medida, el resultado operativo.

Además, los horarios impactan directamente en la vida de todas las personas que están sujetas a ellos. La motivación, la satisfacción y la productividad de los empleados están directamente relacionadas con unas jornadas de trabajo equilibradas y que se puedan adaptar a las necesidades de cada persona.

En resumen, contar con una herramienta adecuada para confeccionar los horarios supone una gran ventaja ya que:

- Mejora la toma de decisiones y permite reducir costes mediante una contratación ajustada a las necesidades específicas de cada momento.

- Mejora el servicio al cliente y los procesos dentro de la empresa mediante la aplicación de los recursos en los momentos precisos.
- Permite disponer de información de calidad y de forma más rápida. También puede ser una herramienta útil para auditorías, tanto internas como externas.
- Facilita la implantación de políticas de conciliación de la vida personal del colectivo trabajador.
- Permite adaptarse de forma sencilla a los requisitos legales en materia de horarios, presentes tanto en el Estatuto de los Trabajadores como en los convenios colectivos, acuerdos a nivel de empresa o cualquier otra disposición legal de obligado cumplimiento.

1.2 Objetivos del Proyecto

El objetivo principal de este proyecto es construir una aplicación que permita a sus usuarios planificar los horarios de grupos de trabajo proporcionando información relativa a estos en tiempo real. De esta forma, se podrá tener constancia, en todo momento, de las coberturas por horas, aspecto especialmente importante para el funcionamiento de la empresa, así como del cumplimiento de toda la normativa asociada a la jornada laboral de los trabajadores.

No obstante, es importante tener en cuenta los intereses de todos los *stakeholders* para asegurar el éxito del proyecto. Así, se ha procedido a la identificación de los más destacados con el objetivo de gestionar la relación con ellos de forma efectiva:

- **Empleado**: Necesita acceder a sus horarios y a los de sus compañeros para poder planificar su vida profesional y privada. Además, necesita comprender de forma sencilla la información que se presenta en dichos horarios.
- **Manager**: Necesita crear horarios de trabajo para los empleados que están a su cargo y trasladar la información al resto de la empresa. Necesita, además, soporte para cumplir con toda la normativa aplicable en materia de horarios.
- **Administrador**: Necesita acceder a todo el sistema y gestionar su seguridad (autenticaciones y autorizaciones).
- **Empresa**: Necesita una solución que satisfaga las necesidades de todos los empleados y garantice el cumplimiento de la normativa legal en materia de protección de datos y de jornada laboral.

De aquí en adelante, se utilizará el término genérico “usuario” para designar a cualquier persona que pueda estar incluida en alguno de los grupos anteriores.

Por lo tanto, el producto final puesto a disposición del cliente debe cumplir los siguientes requisitos, que serán detallados con posterioridad:

- Dar soporte para la confección de planificaciones anuales de horarios basados en turnos por día.
- Permitir la configuración de las planificaciones creadas según las necesidades de cada unidad de negocio. De este modo, se deberán poder indicar datos como el número máximo de jornadas de trabajo por año, el límite de jornadas continuas de trabajo o el tiempo mínimo de descanso entre jornada, entre otros. Asimismo, deberá informar del incumplimiento de estas restricciones en tiempo real.
- Proporcionar un informe de coberturas por tramos horarios en tiempo real.
- Mantener un registro de usuarios, así como permitir el acceso a determinadas partes de la aplicación según roles preestablecidos.

1.3 Enfoque y metodología seguida

Existen multitud de posibilidades a la hora de decidir cómo llevar a cabo el desarrollo de una aplicación como la descrita anteriormente y la forma en la que se presente la información en pantalla puede variar radicalmente de unas opciones a otras.

De todas las opciones manejadas, se ha decidido partir de un enfoque basado en asignaciones de turnos de trabajo por días a los diferentes empleados disponibles, dado el conocimiento del dominio que tiene el autor del proyecto que aquí se describe, gracias a su experiencia en la confección de horarios. Esta experiencia, y el hecho de haber utilizado el mencionado enfoque con anterioridad, animó a pensar que, utilizando las adaptaciones oportunas, la aplicación pueda convertirse en una herramienta con capacidad para satisfacer plenamente las necesidades de todos los interesados.

De esta forma se pueden aprovechar todas las virtudes y mejorar las deficiencias de las soluciones con las que se ha trabajado a lo largo de los años y que no ofrecían los resultados esperados.

Por otro lado, el hecho de construir la aplicación sobre las ideas de otras herramientas existentes hace el cambio menos traumático para los usuarios, ya acostumbrados a unas determinadas rutinas de trabajo, y facilitará esquivar la amenaza de la resistencia al cambio, que podría poner en peligro el éxito del proyecto si no se maneja de forma adecuada.

1.4 Planificación del Trabajo

1.4.1 Arquitectura general, herramientas y frameworks

Una de las principales preocupaciones que se ha tenido a la hora de desarrollar el asistente para la planificación de horarios es conseguir que la solución final sea lo más ágil posible. Es decir, se deben eliminar todos los tiempos de espera mientras el usuario está elaborando una planificación y restringir al mínimo aquellos ineludibles.

Dado que el resultado del presente proyecto es una aplicación destinada a su uso en entornos empresariales, debe tenerse en cuenta que el rendimiento de las redes de comunicaciones podría ser un cuello de botella en determinados casos. Además, las necesidades de cómputo no serán elevadas, siendo cualquier equipo que se utilice en la actualidad capaz de acometer la tarea sin complicaciones. Así, se ha optado por utilizar una arquitectura cliente-servidor, con un *backend* que será una API REST [27][28] y un *frontend* que será una SPA (*Single Page Application*) [24]. De esta forma, se traslada al cliente la responsabilidad de hacer la mayor parte de los cálculos necesarios y se limita la frecuencia y el tamaño de las comunicaciones.

Así, tomando como base la decisión arquitectónica expuesta previamente se ha decidido utilizar las siguientes herramientas y *frameworks*:

Angular [1]: Es un proyecto *open source* mantenido por Google y una amplia comunidad de desarrolladores, que utiliza TypeScript [46] (superconjunto de JavaScript) como lenguaje por defecto. Angular permite el desarrollo de las llamadas Single Page Applications (SPA), propiciando el desacoplamiento de la capa de presentación de la aplicación y reduciendo la interacción con el servidor al intercambio de datos, realizando una única carga de código para generar todas las vistas necesarias.

Como *framework*, Angular proporciona una estructura común para desarrollar proyectos, además de otras ventajas como la posibilidad de crear componentes a medida que se pueden combinar y reutilizar; vincular el modelo y la vista de forma inmediata, continua y transparente para el desarrollador (*data binding*) o facilitar la inyección de dependencias favoreciendo el desacoplamiento entre los diferentes componentes y simplificando la realización de test. Además, proporciona una serie de librerías para las comunicaciones con el servidor, *routing* y otras funcionalidades requeridas en la construcción de aplicaciones.

Para la utilización de Angular se requiere la instalación de Node.js [2], entorno de ejecución para JavaScript.

El autor/desarrollador no poseía ninguna experiencia con Angular en el momento de comenzar el desarrollo del proyecto. La elección de este *framework* se ha basado en el hecho de tener una comunidad de usuarios muy amplia, abundante documentación y tutoriales y ofrecer un conjunto de funcionalidades suficiente para el desarrollo de la aplicación.

Spring [4]: Es un conjunto de *frameworks*/librerías para el desarrollo de todo tipo de aplicaciones Java empresariales, aunque también puede ser utilizado con Kotlin (lenguaje de programación que también está basado en la *Java Virtual Machine*). Spring es capaz de proveer, a través de sus muchos proyectos, de una estructura sobre la que construir aplicaciones de muy diversos tipos, así como las funcionalidades básicas que permiten la implementación de seguridad (autenticación y autorización), la comunicación con otras aplicaciones, la persistencia de datos, inyección de dependencias y un largo etcétera.

En el presente proyecto se hará uso de los siguientes *frameworks*/librerías proporcionadas por el ecosistema Spring:

- **Spring Data JPA** [30]: Proporciona una API para dar soporte a las comunicaciones con sistemas gestores de bases de datos. En concreto, se usará la implementación con Hibernate [5] (herramienta que permite el mapeo entre objetos Java y relaciones SQL).
- **Spring Web MVC** [47]: Proporciona soporte para la construcción de APIs REST.
- **Spring Security** [34]: Para la implementación de autenticación y autorización en el sistema.
- **Spring Boot** [49]: Para automatizar la construcción y configuración de la base sobre la que se creará la aplicación.

La elección de Spring parte de las mismas consideraciones presentadas con anterioridad para Angular. En este caso, el autor del proyecto sí cuenta con experiencia, adquirida en las asignaturas de *Ingeniería del software de componentes y sistemas distribuidos* y *Proyecto de desarrollo de software*.

Eclipse Temurin [5]: Distribución del *OpenJDK* de Adoptium (proyecto de Eclipse Foundation).

Maven [8]: Herramienta para la gestión de proyectos. Se utilizará en la construcción del *backend*. Entre otras capacidades, permitirá la gestión de dependencias, así como el empaquetado del producto final.

PostgreSQL [9]: Es un sistema gestor de bases de datos relacional orientado a objetos. Se ha elegido, principalmente, por la buena integración que tiene con Spring Data JPA.

Docker [10]: Herramienta que permite la creación y gestión de contenedores de software. Un contenedor es una instancia ejecutable de una imagen Docker que se puede crear, iniciar, mover o borrar usando las herramientas que Docker proporciona. Estos contenedores se pueden ejecutar sobre cualquier máquina y sistema operativo y están aislados del resto de procesos que se ejecutan en el sistema.

Una imagen Docker es un archivo que contiene todo lo necesario para ejecutar una aplicación: binarios, dependencias, configuraciones, scripts, además de configuración para el contenedor, como pueden ser variables de entorno y otros metadatos.

Azure [11]: Es la plataforma de computación en la nube creada por Microsoft. Se ha elegido por la disponibilidad de cuenta gratuita a través de la UOC.
Se utilizarán 3 servicios ofrecidos por la plataforma:

- **App Service**: Para alojar el contenedor Docker que ejecute el *frontend* de la aplicación.

- **Máquina Virtual**: Para desplegar los contenedores que ejecuten el *backend* de la aplicación y el sistema gestor de bases de datos (PostgreSQL).
- **Container Registry**: Repositorio de contenedores donde se alojarán los creados con Docker para el *frontend* y el *backend* del sistema.

Git [12]: herramienta de control de versiones.

GitLab [14]: Es una plataforma de desarrollo colaborativo que permite el alojamiento de repositorios de código y que utiliza el sistema de control de versiones Git.

Todas las herramientas y tecnologías utilizadas son *open source*, a excepción de Azure, servicio para el que se cuenta con una suscripción proporcionada por la UOC.

1.4.2 Planificación temporal

La siguiente planificación temporal se ha elaborado teniendo en cuenta una serie de requisitos generales que debe cumplir la aplicación. Este calendario podrá variar a lo largo del proyecto para adaptarse a todos los cambios que se deban introducir con el fin de aportar valor al usuario final.

Como se puede observar, se han marcado como hitos en el diagrama de *Gantt* (figura 1), las entregas de las diferentes PEC en las que se divide la asignatura:

PEC1: Plan de trabajo.
PEC2: Análisis y Diseño de la aplicación.
PEC3: Implementación.
PEC4: Entrega de documentación, productos, presentaciones y memoria.

Por otra parte, se ha propuesto seguir un enfoque iterativo de desarrollo. Cada iteración deberá poner en producción toda la funcionalidad asociada a una serie de requisitos. De este modo, cada etapa conllevará el desarrollo del código (tanto en *frontend* como en *backend*), pruebas, creación de contenedores y despliegue en producción. También se ha adaptado la duración de cada iteración a los conocimientos del desarrollador, aumentando el tiempo dedicado a implementar aquellas funcionalidades con más incertidumbre.

	INICIO	FIN	DÍAS
TFG PLANIFICACIÓN HORARIOS	**01/03/2023**	**26/06/2023**	**118**
PLAN DE TRABAJO	**01/03/2023**	**09/03/2023**	**9**
Oportunidad del proyecto	01/03/2023	02/03/2023	2
Requerimientos Alto Nivel	03/03/2023	06/03/2023	4
Planificación temporal	07/03/2023	08/03/2023	2
Herramientas y frameworks	07/03/2023	08/03/2023	2
Entrega de la PEC1	09/03/2023	09/03/2023	1
ANÁLISIS Y DISEÑO	**10/03/2023**	**14/04/2023**	**36**
Historias de usuario	10/03/2023	12/03/2023	3
Fichas de historias de usuario	13/03/2023	14/03/2023	2
Diagrama arquitectura	15/03/2023	24/03/2023	10
Diseño BBDD	25/03/2023	05/04/2023	12
Prototipo de pantallas	06/04/2023	13/04/2023	8
Entrega de la PEC2	14/04/2023	14/04/2023	1
IMPLEMENTACIÓN	**15/04/2023**	**05/06/2023**	**52**
Funcionalidad AppUser	15/04/2023	21/04/2023	7
Funcionalidad Planning	22/04/2023	28/04/2023	7
Funcionalidad WorkShift	29/04/2023	05/05/2023	7
Funcionalidad PlanningUser	06/05/2023	19/05/2023	14
Seguridad	20/05/2023	04/06/2023	16
Entrega de la PEC3	05/06/2023	05/06/2023	1
MEMORIA Y PRESENTACIÓN	**06/06/2023**	**26/06/2023**	**21**
Desarrollo de pruebas	06/06/2023	25/06/2023	20
Confección memoria	06/06/2023	25/06/2023	20
Preparación video	20/06/2023	25/06/2023	6
Preparación diapositivas	20/06/2023	25/06/2023	6
Informe autoevaluación	24/06/2023	25/06/2023	2
Entrega de la PEC4	26/06/2023	26/06/2023	1

Figura 1. Planificación temporal. Diagrama de Gantt.

1.4.3 Evaluación de riesgos

Los principales riesgos detectados en el proyecto son:

- Incumplimiento del calendario.
- Escasos conocimientos del desarrollador con algunas de las tecnologías a utilizar para el desarrollo de la aplicación en el inicio del proyecto.
- Alcance del proyecto demasiado ambicioso.
- Problemas asociados al despliegue por las limitaciones de la cuenta de estudiante de Azure que proporciona la UOC.
- Pérdida de datos.
- Asuntos no previstos por la falta de experiencia del autor del proyecto.

Los riesgos asociados a la inexperiencia y los conocimientos limitados del autor pueden mitigarse mediante un desarrollo iterativo de la aplicación. De esta forma se pretende conocer lo antes posible cualquier problema no contemplado en la planificación del proyecto. En este mismo sentido, el riesgo de incumplimiento del calendario y de haber estimado el alcance del proyecto por encima de las posibilidades del autor también pueden mitigados mediante este desarrollo iterativo, que deberá permitir tomar medidas correctoras en el menor tiempo posible.

La pérdida de datos puede evitarse fácilmente mediante el uso del sistema de control de versiones elegido, Git, y el uso de repositorios remotos en GitLab.

Los problemas asociados al despliegue pueden ser minimizados iniciando los servicios en la nube y haciendo pruebas en etapas tempranas del desarrollo. Sin embargo, esto puede significar el agotamiento de recursos antes de la finalización del proyecto. Esta situación se pone en conocimiento de los profesores de la asignatura para poder evaluar alternativas.

1.5 Productos obtenidos

El resultado del desarrollo serán dos imágenes Docker, una con el *frontend* y otra con el *backend,* que podrán desplegarse junto con una imagen PostgreSQL, necesaria para la persistencia de datos. Aparte de tener una instancia de Docker en ejecución, no será necesario otro requisito para el funcionamiento de la aplicación en cualquier máquina.

Por otra parte, la finalización del proyecto en su conjunto implica la elaboración de una presentación con diapositivas y un vídeo en el que se ilustren el desarrollo del proyecto y su resultado, la presente memoria y un informe de autoevaluación del autor.

1.6 Contenido de la memoria

En los siguientes capítulos de la memoria se tratará de desgranar todo el proceso de creación del asistente para la planificación de horarios. Se considerarán 3 secciones claramente diferenciadas.

Inicialmente se abordará el análisis y diseño de la aplicación, detallándose los requisitos que ha de cumplir el producto final y mostrándose el diseño conceptual y lógico de la base de datos, y la arquitectura y diagramas de clases del *frontend* y el *backend*.

Una segunda parte en la que se presentará el proceso y resultados de la implementación de la aplicación, desde la configuración del entorno de desarrollo hasta el despliegue, detallando los aspectos más importantes. Al final de esta parte se hará un breve estudio de la deuda técnica acumulada.

Finalmente, se presentarán las conclusiones a las que se ha llegado tras la finalización del proyecto. Primero se abordan las lecciones aprendidas y a continuación se analizarán los objetivos no cumplidos y el seguimiento del desarrollo del producto. Por último, se presentan posibles líneas de trabajo futuro, independientemente de que forme parte de la deuda técnica o sean mejoras que se pueden incorporar a la funcionalidad de la aplicación.

2. Análisis y Diseño

2.1 Requisitos: Historias de usuario y modelo de pantallas

A continuación se presentan los requisitos que deberá cumplir la aplicación en su primera versión. Se ha intercalado el modelo de pantallas con la intención de facilitar la comprensión de las necesidades de los *stakeholders* con respecto a la aplicación que se pretende desarrollar:

R-001	**Como** manager **quiero** acceder al listado de empleados **para** tener información actualizada del personal disponible para hacer horarios.
Detalles conversaciones	El usuario manager podrá acceder a través del área *Empleados* a un listado completo de los usuarios registrados en el sistema (figura 2) con los datos que se han de guardar para cada uno de ellos.
Criterios de aceptación	Dado un manager que se ha autenticado en el sistema, cuando accede al área *Empleados*, entonces el sistema mostrará una relación de todos los empleados disponibles en el sistema para la confección de horarios.

Figura 2. Pantalla principal del área Empleados.

R-002	**Como** manager **quiero** gestionar turnos de trabajo **para** asignarlos a los empleados en diferentes fechas en cada planificación.
Detalles conversaciones	El manager podrá acceder a un listado completo de los turnos que han sido creados en el sistema a través del área *Turnos* (figura 3). La información que se ha de detallar para cada uno de ellos es la que allí se especifica. Desde esa pantalla podrá eliminar los turnos existentes, crear nuevos turnos o modificarlos (figura 4 y figura 5).
Criterios de aceptación	Dado un manager que se ha autenticado en el sistema y que accede al área *Turnos*, cuando el manager realiza operaciones de creación, modificación y borrado de turnos, entonces la base de datos se actualizará de forma acorde con la solicitud y el sistema mostrará el resultado. En caso de que no sea así el sistema deberá ofrecer un mensaje de aviso informando de la situación.

Figura 3. Pantalla principal del área Turnos.

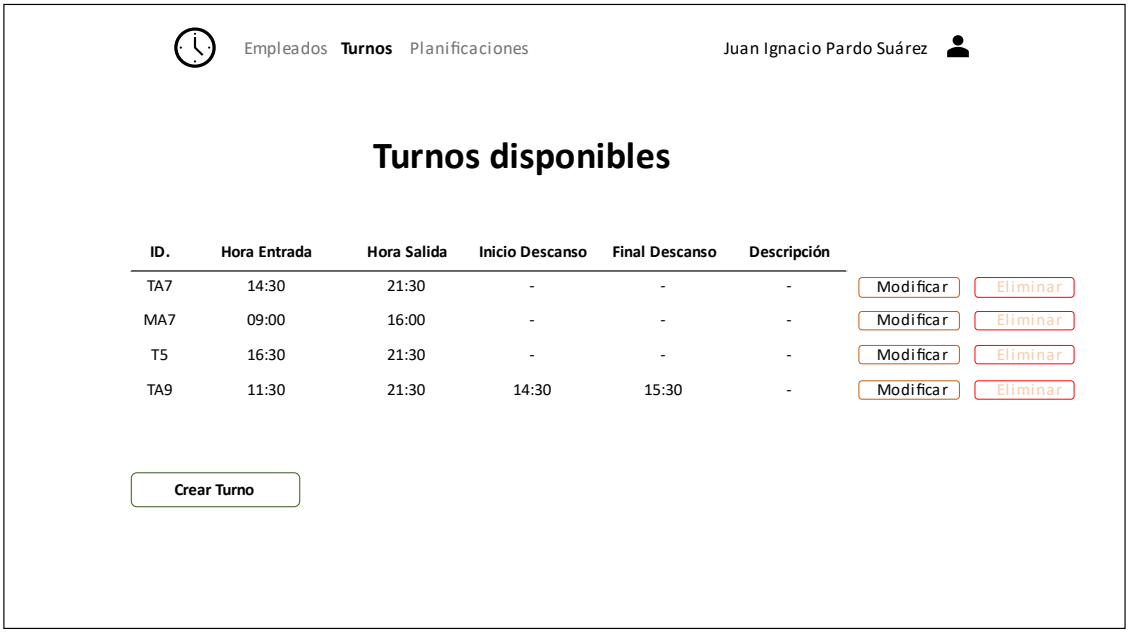


Figura 4. Pantalla de creación de un nuevo turno.

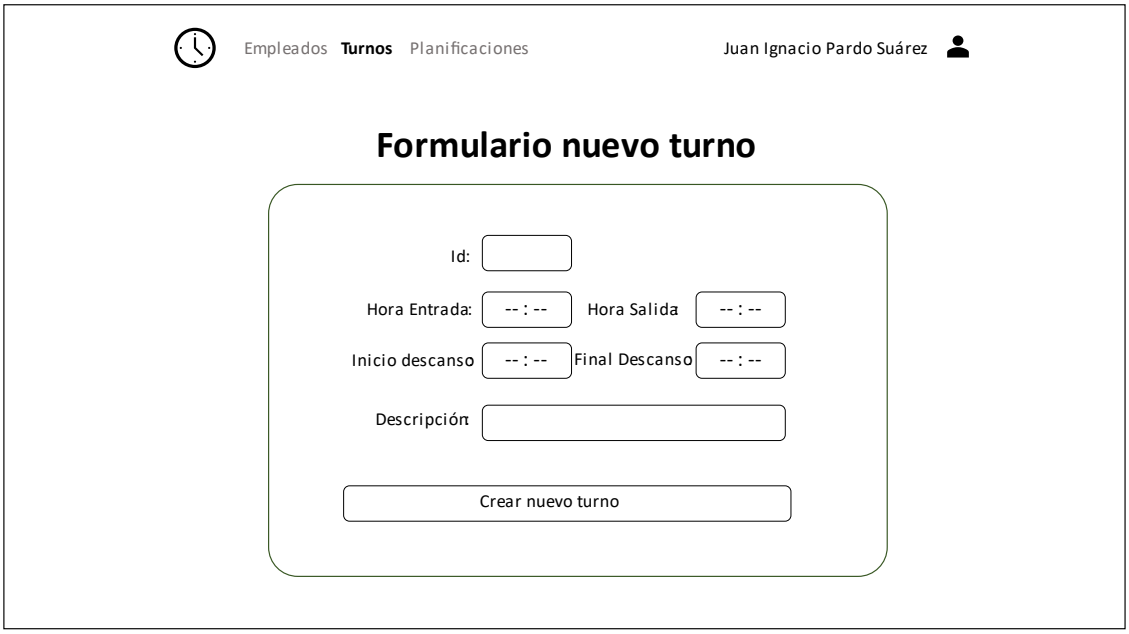
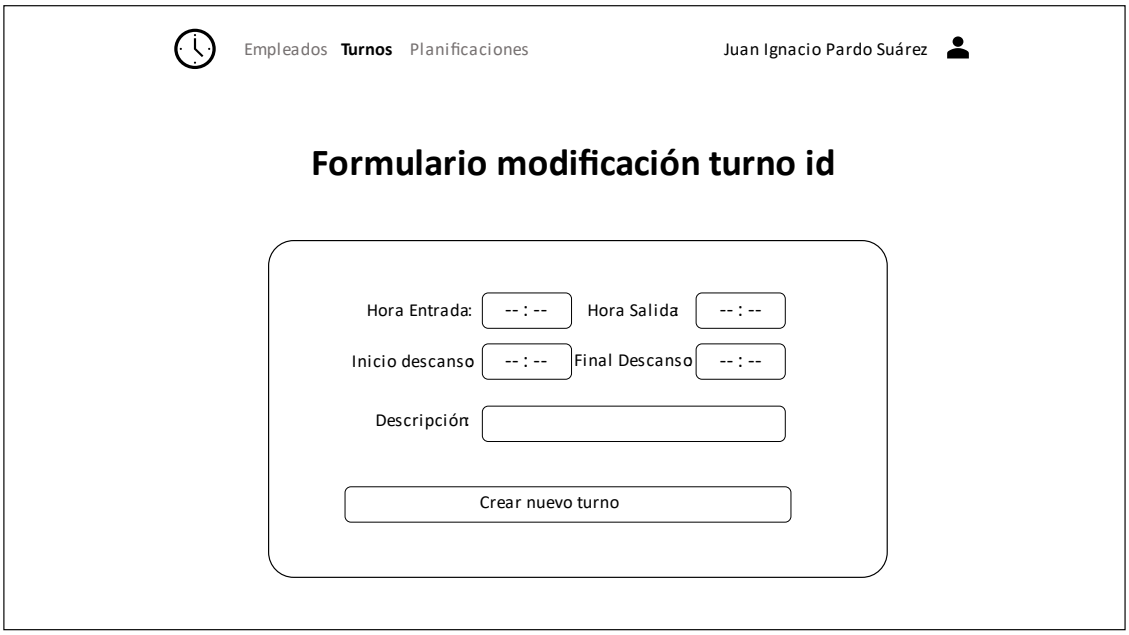


Figura 5. Pantalla de modificación de turno.



R-003	**Como** manager **quiero** crear planificaciones de horarios con diferentes posibilidades de configuración de restricciones legales e internas de la compañía y añadir y eliminar empleados de una planificación concreta **para** crear el marco de trabajo para la asignación de horarios.
Detalles conversaciones	Los managers necesitan crear planificaciones y configurarlas con determinadas restricciones que pueden responder a normativas legales o internas de la empresa. Además, deben tener la posibilidad de añadir o quitar empelados de una planificación independientemente de que se les hayan asignado turnos de trabajo en determinadas fechas. Las restricciones que se podrán configurar son: • número máximo de horas trabajadas en un año, • número máximo de jornadas trabajadas en un año, • número máximo de jornadas de trabajo seguidas, • número máximo de horas trabajadas en un día, • número mínimo de horas de descanso entre jornadas Además, para una planificación se deberá conocer el año y se podrá incluir una descripción. Un manager podrá acceder a las planificaciones que existen en el sistema desde el área *Planificaciones* (figura 6). Desde ahí podrá eliminar planificaciones, crear nuevas planificaciones (figura 7), modificar la configuración de restricciones y añadir o quitar empleados (figura 8 y figura 9).
Criterios de aceptación	Dado un manager que se ha autenticado en el sistema y que accede al área de Planificaciones, cuando el manager realiza operaciones de creación, modificación y borrado de planificaciones, entonces la base de datos se actualizará de forma acorde con la solicitud y el sistema mostrará el resultado de la operación. En caso de que la solicitud no se haya ejecutado con éxito el sistema mostrará el correspondiente aviso.

Figura 6. Pantalla principal del área planificaciones.

Figura 7. Pantalla de creación de nueva planificación.

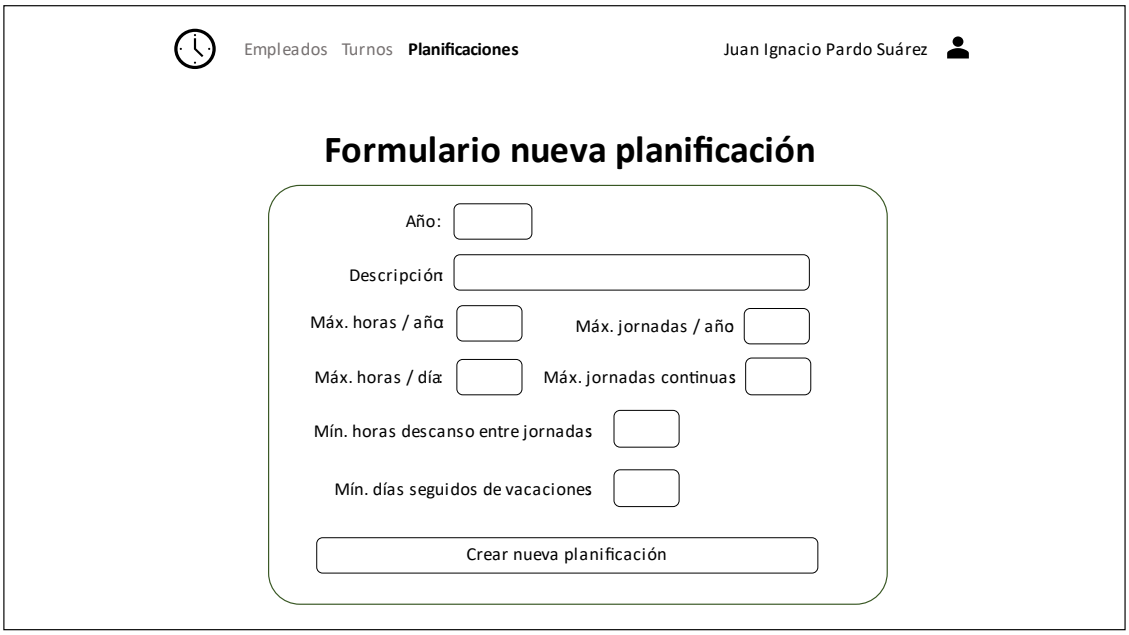


Figura 8. Pantalla de detalles de una planificación.

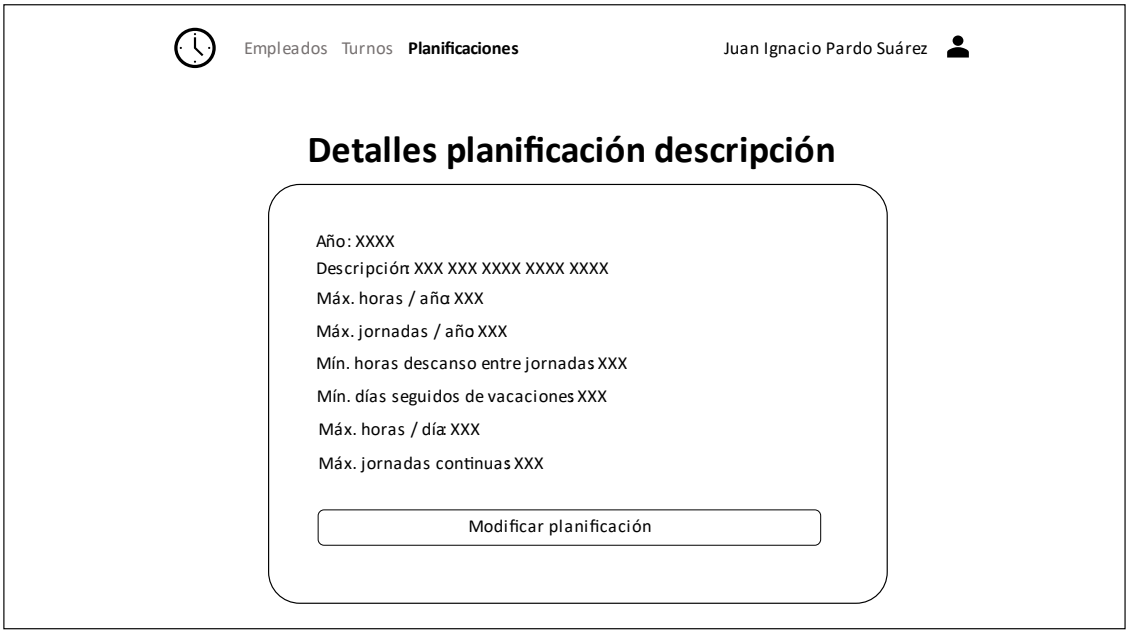
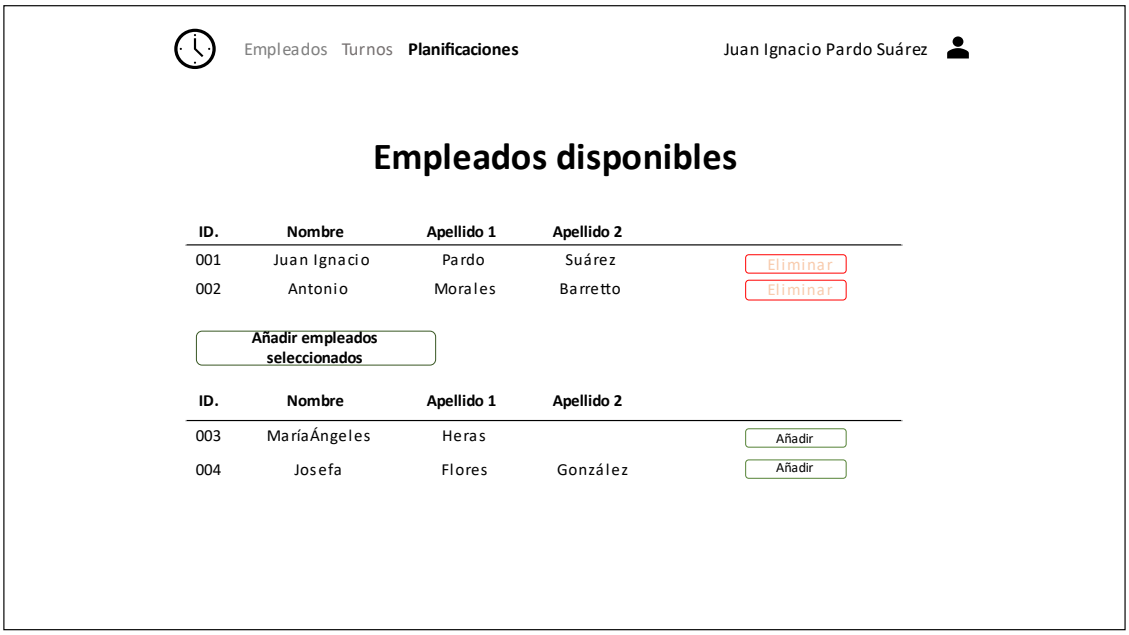


Figura 9. Pantalla de asociación de empleados a una planificación.



R-004	**Como** manager **quiero** ver y asignar turnos de trabajo a diferentes empleados para una determinada planificación **para** crear horarios que satisfagan las necesidades de la unidad de negocio de la que estoy a cargo.
Detalles conversaciones	Los managers necesitan establecer turnos de trabajo en distintas fechas para un determinado empleado en una planificación concreta. Se ha decidido reflejar las jornadas de trabajo mediante turnos preestablecidos que se asignan a los diferentes trabajadores en distintas fechas. El resultado final es una tabla en cuyas filas están los diferentes trabajadores y en cuyas columnas están las fechas. En cada intersección, en forma de celda, se asignan los diferentes turnos (figura 10).
Criterios de aceptación	Dado un manager que se ha autenticado en el sistema y que accede al área *Planificaciones* y selecciona una planificación en concreto, cuando el manager introduce los identificadores de turnos en la tabla correspondiente y guarda la planificación, entonces la base de datos se actualizará de forma acorde con la solicitud y el sistema mostrará el resultado de la operación.

Figura 10. Pantalla de asignación de turnos en una planificación.

Enero	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
FGJ	TA7	TA7	TA7	TA7	TA7			TA7	TA7	TA7	TA7	TA7			TA7	TA7	TA7	TA7	TA7	TA7	TA7	TA7	TA7	TA7	TA7	TA7			TA7	TA7	TA7
MBA	T5	T5	T5	T5	T5			T5	T5	T5	T5			T5	T5	T5	T5				T5	T5	T5	T5	T5	T5				T5	T5

Enero	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
19:00 20:00	2	2	2	2	2	0	0	2	2	2	2	1	0	1	2	2	2	1	1	1	2	2	2	2	2	2	0	0	1	2	2

R-005	**Como** manager **quiero** tener *feedback* inmediato del cumplimiento de las restricciones indicadas en la configuración de una planificación **para** cumplir con toda la normativa vigente.
Detalles conversaciones	El *feedback* se ha de tener en el menor tiempo posible. Los *managers* no quieren esperar a guardar la planificación en la base de datos y volver a cargarla. Básicamente, necesitan que el *frontend* sea capaz de dar la información necesaria para conocer los incumplimientos de restricciones. En otro caso, el trabajo se hace tedioso debido a los tiempos de espera. Así, los avisos han de ser claros y visibles.
Criterios de aceptación	Dado un manager que se ha autenticado en el sistema y que accede al área *Planificaciones*, cuando el manager introduce turnos en la tabla de la planificación de horarios de empleados y existe algún tipo de incumplimiento de las restricciones indicadas en la configuración de la planificación, el sistema responderá de forma inmediata indicando el origen del problema sin necesidad de guardar la planificación en la base de datos.

R-006	**Como** manager **quiero** tener *feedback* inmediato de la cobertura por horas de una planificación **para** poder garantizar el servicio que mi departamento ha de prestar en cada momento.
Detalles conversaciones	Este es otro aspecto básico para que los *managers* puedan trabajar de forma ágil. En el fondo sigue el mismo espíritu que la historia R-005. Se ha pensado que la información puede residir en una tabla de doble entrada con las horas en las filas y los días en las columnas. En cada celda se mostraría un número correspondiente a la cantidad de personas que tienen turno de trabajo asignado en ese momento. Los *managers* también han apuntado la posibilidad de crear un código de color para destacar aquellos momentos en los que la cobertura esté por encima o por debajo de unos valores que se podrían configurar previamente.
Criterios de aceptación	Dado un manager que se ha autenticado en el sistema, cuando el manager accede a una planificación, entonces tendrá a su disposición una tabla de coberturas por horas con información sobre el número de trabajadores con turno asignado en cada franja horaria, y dicha información se actualizará a medida que el usuario introduce variaciones en los turnos asignados a los empleados, sin necesidad de guardar la información en la base de datos. El manager podrá mostrar u ocultar esta tabla a voluntad.

R-007	**Como** empleado **quiero** poder acceder a mis horarios y los de mis compañeros de trabajo **para** saber cuándo tengo que ir a trabajar, poder planificar mi tiempo y solicitar cambios de turno con otros compañeros.
Detalles conversaciones	Las principales inquietudes de los empleados se refieren a comprender la información que se consigna en la planificación. Además, están muy interesados en conocer los turnos que realizarán otros compañeros para poder solicitar cambios en caso de que sea necesario. En un primer momento se ha pensado que los empleados puedan acceder a la misma tabla en la que se consignan los turnos en cada planificación sin la información asociada al cumplimiento de requisitos ni la información asociada a coberturas.
Criterios de aceptación	Dado un empleado que se ha autenticado en el sistema y accede al área *Planificaciones* y selecciona una planificación en concreto, entonces el usuario tendrá a su disposición una tabla con los turnos asignados a los diferentes empleados asociados a la planificación.

R-008	**Como** empleado **quiero** tener información sobre los códigos utilizados por la empresa en la elaboración de los horarios **para** poder interpretarlos de forma correcta.
Detalles conversaciones	Relativo también a la historia R007, los empleados tienen interés en conocer los códigos utilizados para identificar los turnos con el fin de saber las horas a las que deben prestar sus servicios. Al acceder al área *Turnos*, un empleado podrá ver un cuadro con la descripción de los turnos que se han creado en el sistema similar al que tienen acceso los managers (figura 3), pero con funcionalidad restringida.
Criterios de aceptación	Dado un empleado que se ha autenticado en el sistema y accede al área *Turnos*, entonces el usuario tendrá a su disposición una tabla con la información asociada a cada turno que se ha creado en el sistema.

R-009	**Como** administrador **quiero** registrar, eliminar y modificar usuarios **para** que el sistema ofrezca información actualizada del personal disponible para hacer horarios.
Detalles conversaciones	El usuario administrador podrá acceder, a través del área *Empleados,* a un listado completo de los usuarios registrados en el sistema (figura 11) con los datos que se han de guardar para cada uno de ellos. A través de esta pantalla podrá acceder a todas las acciones disponibles (figura 12 y figura 13). Las funcionalidades de creación y modificación de usuarios son críticas porque conllevan la adjudicación de privilegios a través del campo *rol*. Así, la empresa desea que sean solamente los administradores quienes puedan tener acceso a ellas.
Criterios de aceptación	Dado un administrador que se ha autenticado en el sistema y que accede al área *Empleados*, cuando introduce los datos de un nuevo empleado o modifica o elimina los datos de uno existente, entonces la base de datos se actualizará de forma acorde con la solicitud y el sistema responderá mostrando el resultado de la operación. En caso de que no sea así, el sistema deberá ofrecer un mensaje de aviso informando de la situación.

Figura 11. Pantalla principal del área de empleados con funcionalidad extendida para administradores.

Figura 12. Pantalla de creación de un nuevo usuario.

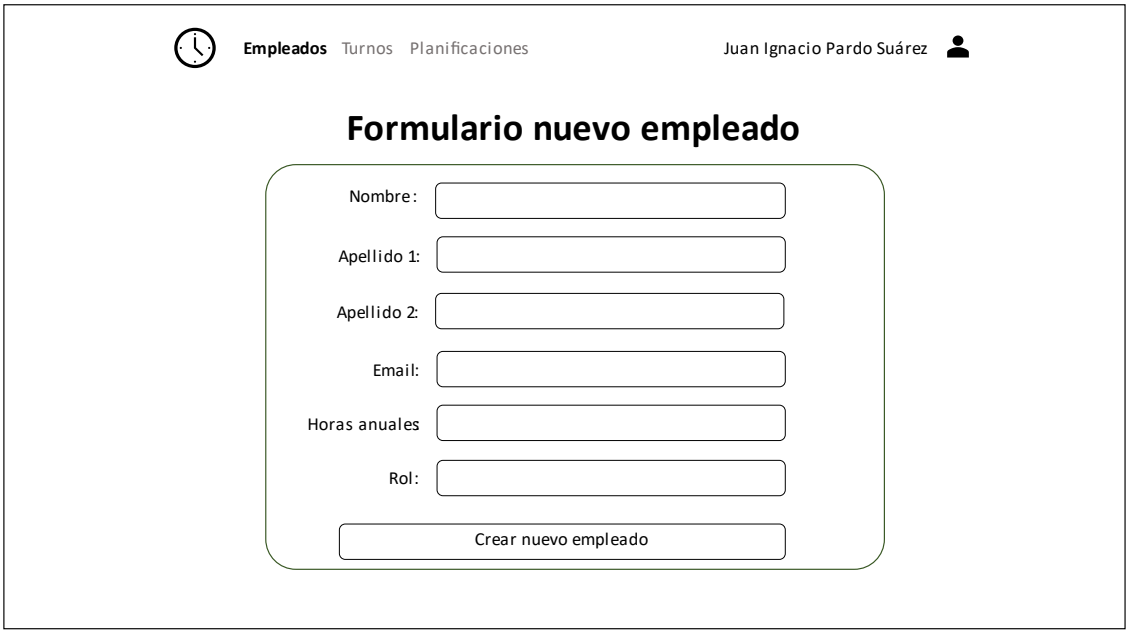
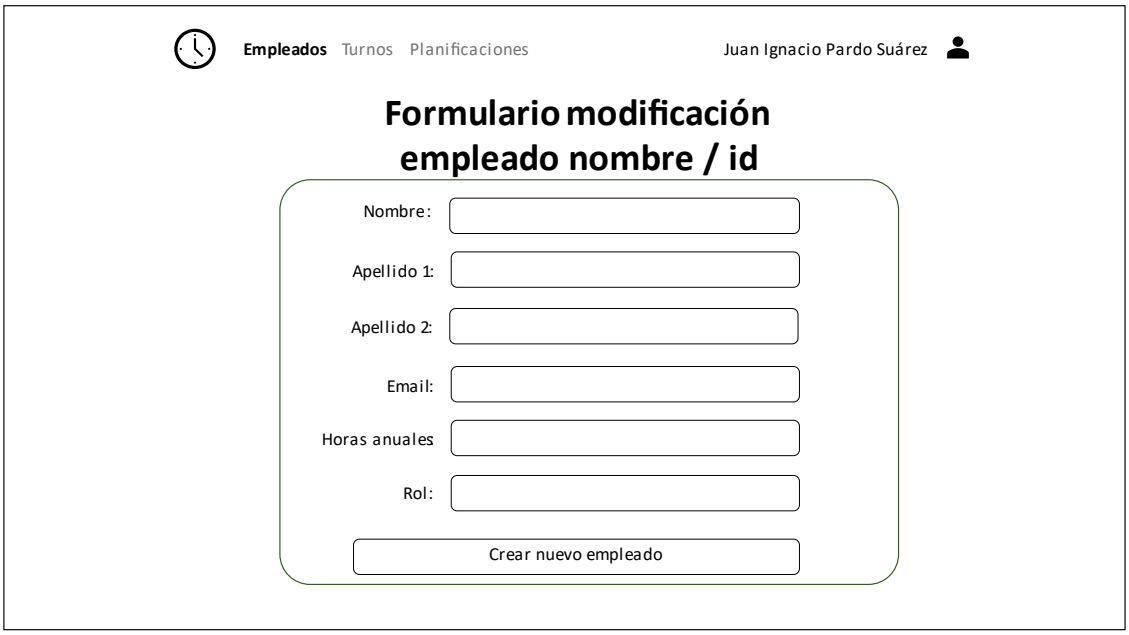


Figura 13. Pantalla de modificación de datos de un usuario.



R-010	**Como** usuario **quiero** que mi cuenta esté protegida por una contraseña **para** impedir accesos no autorizados.
Detalles conversaciones	La pantalla de acceso a la aplicación es un formulario en el que el sistema solicita email y contraseña para poder autenticarse (figura 14). Una vez se ha accedido a la funcionalidad de la aplicación, el nombre de usuario aparecerá en la cabecera y se podrá acceder a las opciones de cambio de contraseña en el icono situado a su lado (figura 15). La asignación inicial de contraseña se producirá en el momento en el que un nuevo usuario sea dado de alta. Esta será la contraseña que se entregue en un primer momento hasta que el usuario proceda al cambio de esta mediante la correspondiente funcionalidad. En caso de que el usuario no pueda acceder al sistema, bien por no introducir los datos requeridos de forma correcta o por no estar dado de alta, recibirá un aviso del sistema indicando la forma de solucionar cualquier incidencia con la aplicación
Criterios de aceptación	Dado un usuario que desea acceder a la funcionalidad del sistema, cuando se dirige a la url de la aplicación, entonces el sistema muestra un formulario de *login* y, si el usuario introduce unos datos de acceso válidos, el sistema le concederá acceso a las funcionalidades correspondientes a su rol. En caso contrario, el sistema mostrará un mensaje indicando la situación. Por otra parte, dado un usuario que se ha autenticado en el sistema y que entra en el área de Empleados, cuando accede a las opciones de usuario a través del icono situado a la derecha de su nombre, entonces tendrá la posibilidad de modificar su contraseña, y la base de datos se actualizará de forma acorde con la solicitud y el sistema responderá con un aviso de que la operación ha concluido con éxito. En caso de que no sea así, el sistema deberá ofrecer un mensaje de aviso informando de la situación.

Figura 14. Pantalla con el formulario de autenticación.

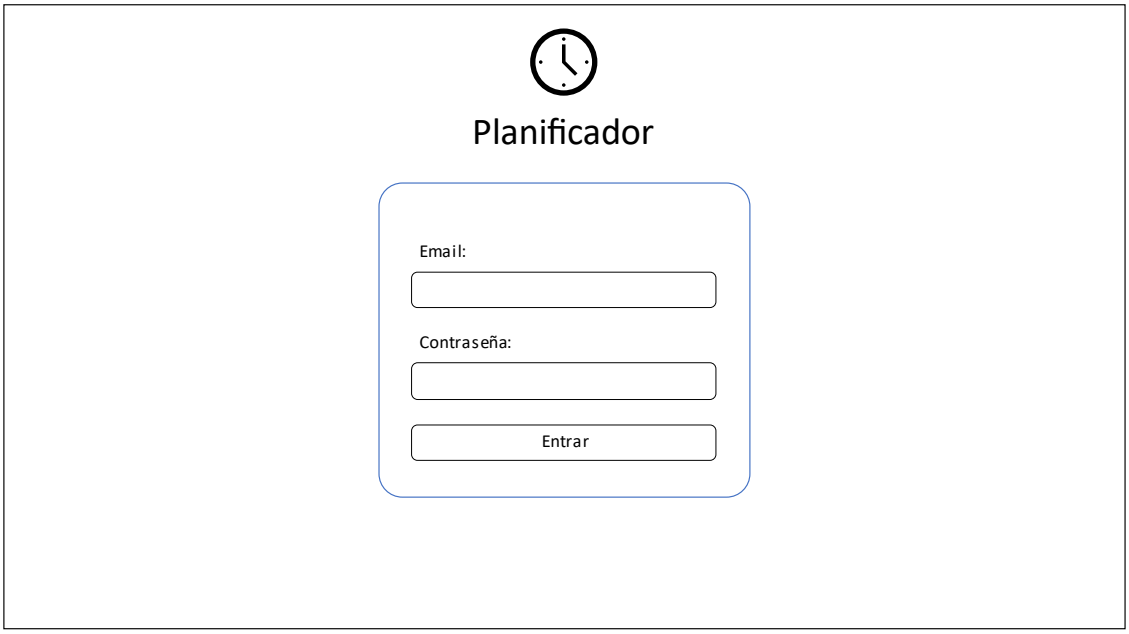
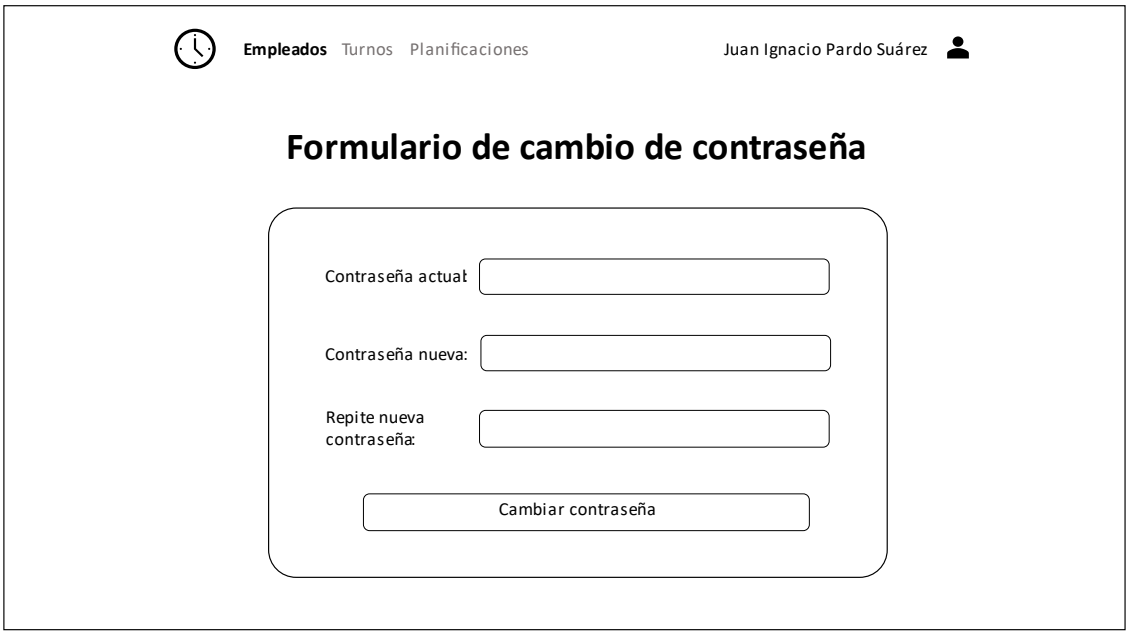


Figura 15. Pantalla con el formulario de cambio de contraseña.



R-011	**Como** empresa **quiero** que la aplicación tenga un sistema de seguridad que controle los accesos a diferentes partes de esta en función de las categorías profesionales de los usuarios y sus necesidades **para** garantizar el cumplimiento de la normativa de protección de datos y de seguridad interna.
Detalles conversaciones	La empresa está interesada en un sistema de autorización para permitir el acceso a las diferentes partes del sistema en función del rol asociado a cada usuario.
Criterios de aceptación	Dado un usuario que se ha autenticado en el sistema, dicho usuario solo podrá tener acceso a la funcionalidad que le ha sido asignada según su rol de usuario.

2.2 Diseño de la base de datos

2.2.1 Diseño conceptual

La base de datos que se ha de construir debe permitir guardar los datos necesarios para el funcionamiento de la aplicación, que han sido descritos en las historias de usuario y pantallas presentadas con anterioridad.

De forma general, se han de guardar datos de usuarios de la aplicación, planificaciones, turnos y asignaciones de turnos a usuarios en el contexto de una planificación.

En cualquier caso, se han de tener en cuenta las siguientes consideraciones:

- Una planificación se debe poder crear sin usuarios asociados, que se podrán añadir o eliminar posteriormente.
- Los turnos solo se pueden asignar a un usuario que ha sido asociado a una determinada planificación. El usuario y la planificación son el contexto de la asignación de un turno, que toma sentido si hay algún usuario que realiza ese turno en una determinada fecha dentro de una planificación.
- De lo anterior se deriva que un usuario en una planificación solo puede tener asociado un turno, como máximo, por cada fecha del año al que corresponde esa planificación.

Con la intención de satisfacer todos los requisitos presentados con anterioridad, se proponen dos opciones para el diseño conceptual de la base de datos. El hecho de decantarse por una u otra opción dependerá de las necesidades de la aplicación.

Figura 16. Opción 1 para el diseño conceptual de la base de datos.

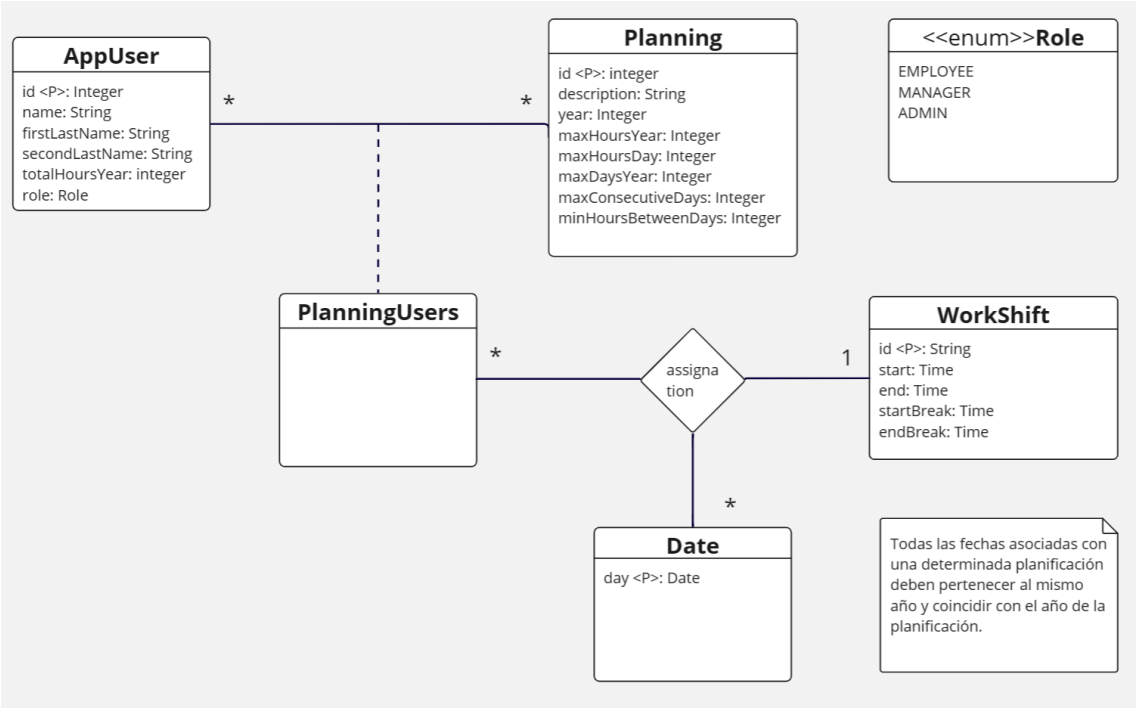
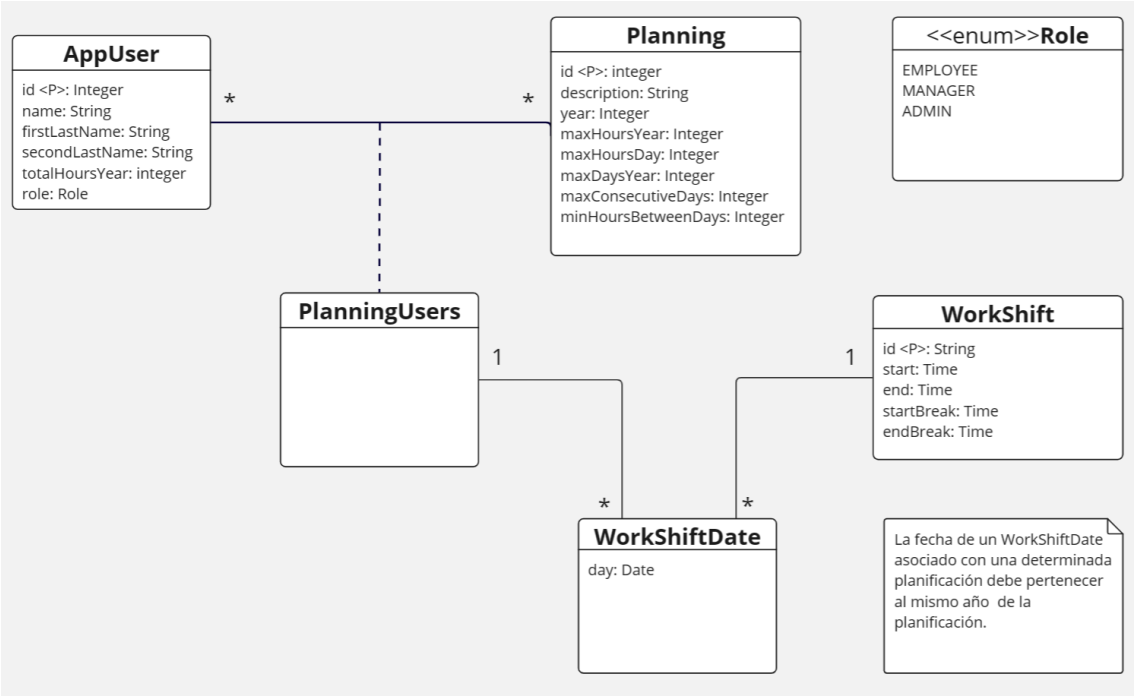


Figura 17. Opción 2 para el diseño conceptual de la base de datos.



Como puede apreciarse, en la figura 16, en la opción 1 se establece una relación ternaria (*Assignation*) entre *PlanningUsers*, *WorkShift* y *Date*, con cardinalidad tal que para una combinación de *Planning, User* y *Date* concreta solo le puede corresponder un determinado *WorkShift*. Esta opción intenta satisfacer todas las restricciones necesarias para representar el dominio del problema.

La opción 2 (figura 17) relaja las restricciones impuestas a la base de datos (que deberán implementarse en la lógica de la aplicación) con el fin de simplificar su diseño y facilitar la obtención y tratamiento de los datos.

2.2.2 Diseño lógico

Para realizar el diseño lógico de la base de datos seguiremos la siguiente notación, utilizada por Burgués y Cuartero [20]:

- Se denotarán las relaciones a partir del nombre, seguido de la lista de atributos entre paréntesis y separados por comas.
- Se denotarán las claves primarias subrayando con una línea continua los atributos que las forman.
- Se denotarán las claves alternativas subrayando con una línea discontinua los atributos que las forman.
- Se denotarán las claves foráneas mediante notación textual en la que indicaremos qué atributos son claves foráneas y a qué relaciones referencian.
- Se utilizará el tipo de letra **negrita** en los nombres de atributos que queremos declarar *NOT NULL*.

En cada caso, el diagrama lógico quedaría de la siguiente forma:

Relaciones comunes a las dos opciones:

- **AppUser** (id, **name**, **firstLastName**, secondLastName, **totalHoursYear**, **role**)
- **Planning** (id, description, **year**, maxHoursYear, maxHoursDay, maxDaysYear, maxConsecutiveDays, minHoursBetweenDays)
- **WorkShift** (id, **start**, **end**, startBreak, endBreak)
- **PlanningUsers** (appUser, planning)
  - {appUser} es clave foránea de AppUser
  - {planning} es clave foránea de Planning

Opción 1:

- **Date** (day)
- **Assignation** (appUser, planning, day, **workShift**)
  - {appUser, planning} es clave foránea de PlanningUsers
  - {day} es clave foránea de Date

<u>Opción 2</u>:

- **WorkShiftDate** (**appUser, planning, day, workShift**)
  - {appUser, planning} es clave foránea de PlanningUsers
  - {workShift} es clave foránea de WorkShift

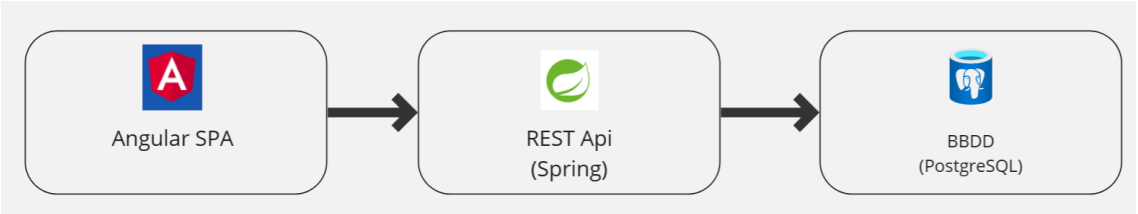
La opción 1 representa de forma más acertada el dominio del problema con sus restricciones asociadas. Sin embargo, el resultado es más complejo y también dificulta la obtención y tratamiento de los datos.

Dado que el riesgo de inconsistencias en la base de datos no es crítico se ha decidido elegir la opción 2 y aplicar las restricciones oportunas desde el *frontend/backend.*

2.3 Arquitectura de la aplicación

Para el desarrollo de la aplicación se ha elegido una arquitectura basada en un *backend* que expone una API REST [27][28] construida con *Spring*, que a su vez será consumido por una SPA [29] desarrollada con *Angular* como frontend (figura 18). De esta forma, se pretende desacoplar la capa de presentación de la lógica del negocio y la persistencia de datos. Además, de cara a mejorar la experiencia de usuario, se ha optado por añadir en la capa de presentación casi toda la funcionalidad relativa al soporte para la construcción de horarios, restringiendo de esta forma las comunicaciones con el *backend* a las estrictamente necesarias para garantizar la persistencia de datos.

Figura 18. Arquitectura de la aplicación.



2.3.1 Arquitectura backend

Un servicio para la confección de horarios como el propuesto podría encajar perfectamente dentro de una intranet corporativa que ofreciese otras herramientas relacionadas con la actividad empresarial (marketing, ventas, R.R.H.H. etc.). Una empresa que requiere software tolerante a fallos, elástico, fácil de evolucionar y desplegar, y altamente escalable, es el escenario perfecto para utilizar una arquitectura basada en microservicios. Por este motivo se ha tratado de desarrollar la API de la forma más cercana posible a un microservicio, de forma que pueda ser fácilmente incorporada a dicha arquitectura con los mínimos cambios posibles. Se deja como mejora la separación del servicio de usuarios y el establecimiento de comunicaciones síncronas con el servicio de

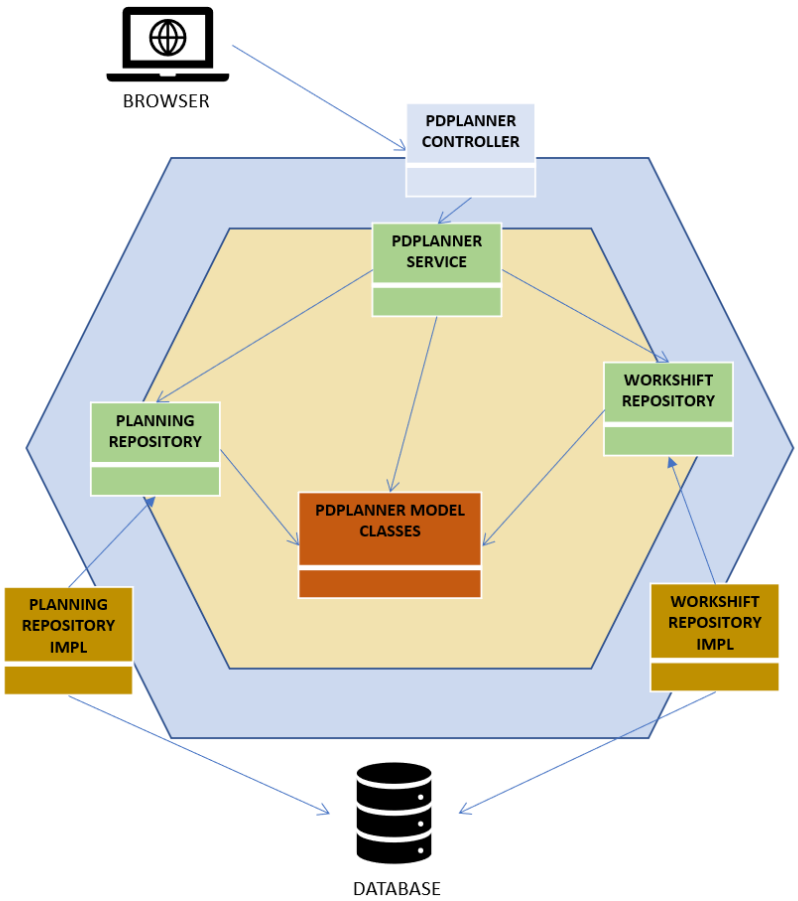
planificación de horarios para que coincida con lo esperado en una arquitectura de microservicios.

Con este pensamiento en mente, se ha tomado la decisión de abandonar la clásica arquitectura en 3 capas (presentación, lógica de negocio y persistencia), que tal y como explica Richardson [23] presenta las siguientes deficiencias:

- No representa el hecho de que una aplicación puede ser invocada por más de un sistema.
- No representa el hecho de que una aplicación puede interactuar con más de una base de datos.
- Define la lógica de negocio como dependiente de la capa de persistencia.

Por lo tanto, se ha decidido utilizar una arquitectura hexagonal para el desarrollo del *backend* de la aplicación [23]. El esquema simplificado de la propuesta sería el presentado en la figura 19:

Figura 19. Arquitectura hexagonal.



En el centro del esquema se encuentra la lógica de negocio, en la que se definen los conocidos como puertos de entrada y salida. Los puertos de entrada son generalmente interfaces que exponen la funcionalidad de la lógica de negocio (en el ejemplo PDPLANNER SERVICE). Los puertos de salida, que también son generalmente interfaces, definen la forma en que la lógica de negocio se comunica con el exterior (en el ejemplo PLANNING REPOSITORY y WORKSHIFT REPOSITORY).

Por otro lado, rodeando la lógica de negocio se encuentran los adaptadores de entrada y de salida. Los adaptadores de entrada son los encargados de recoger las peticiones del exterior invocando los puertos de entrada. En el ejemplo, PDPLANNER CONTROLLER manejará las peticiones desde los navegadores de los usuarios invocando a PDPLANNER SERVICE. En cuanto a los adaptadores de salida, estos implementan la funcionalidad indicada en los puertos de salida y permiten la comunicación de la lógica de negocio con el exterior. En el ejemplo esto se materializa en los *REPOSITORY IMPL, que serán los encargados de la comunicación con un SGBD.

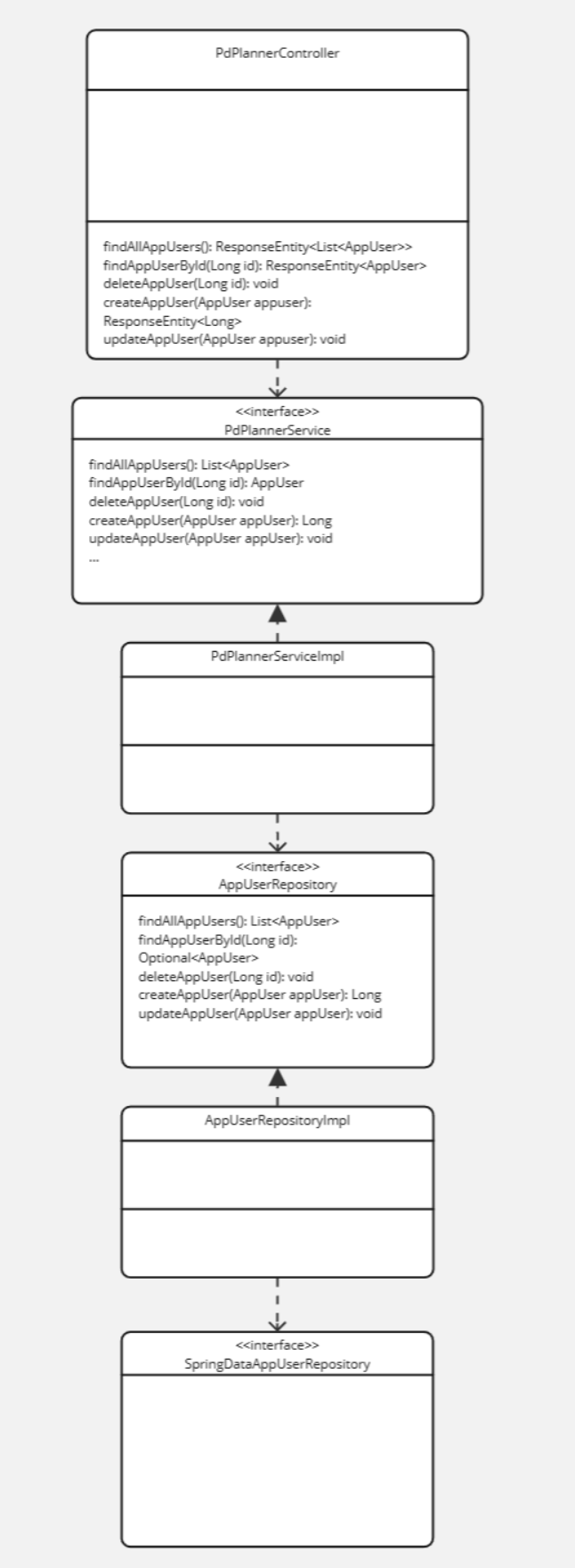
La arquitectura hexagonal permite un alto grado de desacoplamiento entre la lógica de negocio, la persistencia de datos y la presentación por medio del uso de los puertos y adaptadores comentados anteriormente, que no son más que una materialización del principio de inversión de dependencias mediante la utilización del patrón *Adapter* [48].

2.3.2 Diagrama de clases (backend)

Es importante destacar el hecho de que se ha detallado solo una muestra de las operaciones necesarias en el controlador (PdPlannerController) y la interfaz del servicio (PdPlannerService) para no repetir las operaciones de creación, actualización, lectura y borrado necesarias para las diferentes entidades del dominio (son en todos los casos similares). Además, no se han representado todas las clases del dominio, indicadas en el apartado correspondiente al diseño de la base de datos, con el ánimo de simplificar el diagrama y facilitar su comprensión pues presentan el mismo tipo de relaciones que las que se muestran en las figuras siguientes.

En la figura 20 se puede observar la ruta que seguiría una petición generada desde un cliente a través de las diferentes clases.

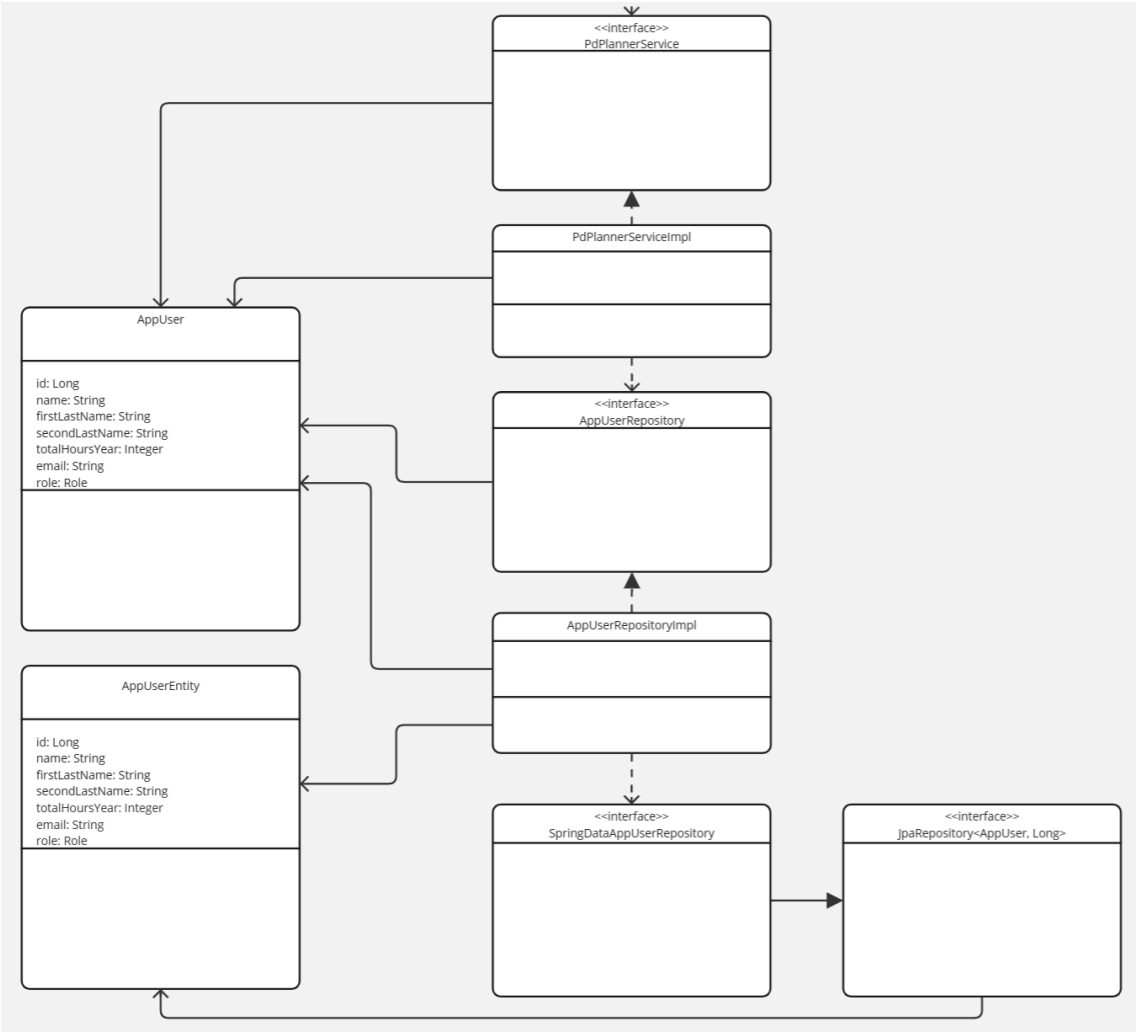
Figura 20. Diagrama de clases del backend parte 1.



Se pueden observar con facilidad las interfaces a las que se hacía referencia con anterioridad, en la presentación de la arquitectura hexagonal, y la aplicación del principio de inversión de dependencias que ello supone.

A continuación, en la figura 21 se muestra un detalle de las relaciones y funcionalidad de la entidad AppUser con su DTO (Data Transfer Object, cuyo significado e importancia se analizará más adelante) asociado.

Figura 21. Relaciones de la entidad AppUser.



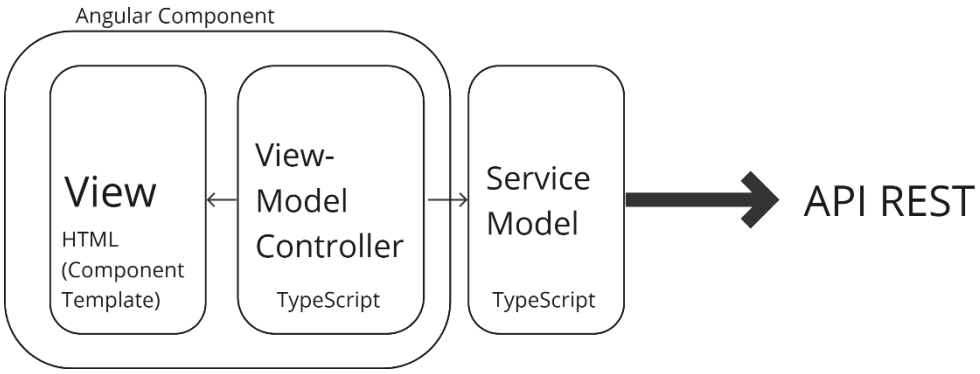
Por último, se presenta una muestra de los *endpoints* de la API. Los relacionados con el resto de las entidades tienen una estructura similar:

HTTP R.	Dirección	Descripción
GET	/pdplanner/appusers/	Recupera todos los usuarios
GET	/pdplanner /appusers/:id	Recupera un usuario
DELETE	/pdplanner /appusers/:id	Borra un usuario
UPDATE	/pdplanner /appusers/:id	Actualiza un usuario
POST	/pdplanner /appusers	Crea un usuario

2.3.3 Arquitectura frontend

En lo que respecta al *frontend*, Angular sigue una arquitectura que se puede aproximar como un híbrido de MVVM (Model-View-ViewModel) y MVC (Model-View-Controller), es la denominada arquitectura MV* [22], que podemos ver en la siguiente figura:

Figura 22. Arquitectura Angular [22]



La principal diferencia entre el modelo MVC y el modelo MV* es la incorporación del llamado ViewModel, que en Angular se materializa mediante la vinculación del modelo y las vistas de forma totalmente transparente para el programador (Angular *binding*), permitiendo que cambios en estas últimas se reflejen de forma automática en el modelo y viceversa.

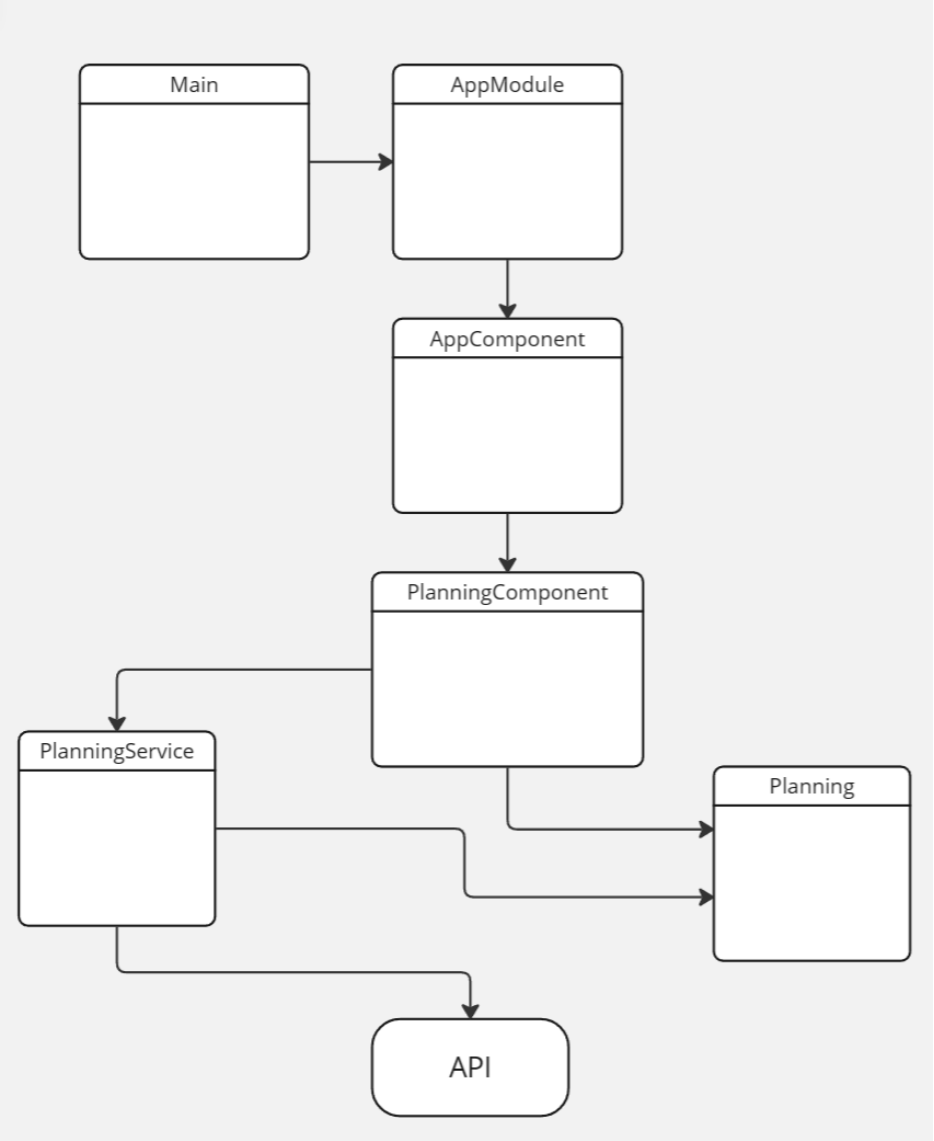
Por otro lado, se dice que Angular tiene una arquitectura orientada a componentes, ya que son éstos (Angular *Component*), las unidades básicas de construcción en este framework. Un Componente Angular, en general, está formado por una clase escrita con TypeScript y una plantilla *html* (*template*) escrita con HTML y CSS, y es precisamente en él donde residen la vista y el controlador (plantilla y clase TypeScript respectivamente) del modelo MV* que sigue el *framework,* así como la vinculación entre vista y modelo antes comentada (ViewModel).

Finalmente, a través de los servicios implementados se harán las llamadas a la API REST. De esta forma se obtendrán los datos para que el frontend ofrezca las vistas necesarias a los usuarios de forma que estos puedan llevar a cabo sus tareas relativas a la planificación de horarios.

2.3.4 Diagrama de clases (frontend)

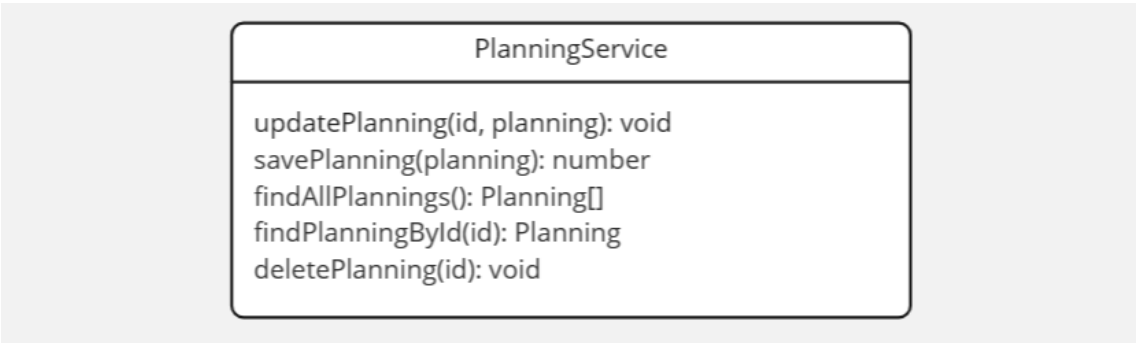
En el siguiente diagrama de clases (figura 23) se muestra únicamente la clase Planning del modelo, junto con sus componentes y servicios asociados. El resto de las clases del modelo tienen relaciones similares a estas.

Figura 23. Diagrama de clases/componentes del frontend.



Los campos de clase del modelo (clases Planning, WorkShift, PlanningUsers y AppUser) son similares al caso del backend. Las operaciones y campos de las clases de los componentes están aún por definir en función de la forma en la que finalmente se construyan. Por otro lado, los diferentes servicios han de implementar la funcionalidad necesaria para interactuar con la Api que se he diseñado para el backend (figura 24).

Figura 24. Ejemplo funcionalidad de servicio en Angular. PlanningService.



3. Implementación

3.1 Configuración del entorno de desarrollo.

Tal y como se había avanzado con anterioridad, se ha tomado la decisión de utilizar las siguientes herramientas en el desarrollo del proyecto:

- **Visual Studio Code**. Editor de código ligero que permite ampliar su funcionalidad a través de extensiones. Se utilizará en el desarrollo del *frontend*.
- **IntelliJ IDEA**. Entorno de desarrollo integrado para aplicaciones Java. Se utilizará para el desarrollo del *backend*.
- **Docker**. Herramienta para la gestión de contenedores. Nos permitirá ejecutar una instancia de PostgreSQL para la persistencia de datos de la aplicación.
- **Postman**. Herramienta que permite la creación de peticiones http y el análisis de las respuestas. Es de gran utilidad en el desarrollo de la API REST.
- **Git**. Sistema de control de versiones.
- **GitLab**. Plataforma de desarrollo colaborativo.

Con relación al sistema de control de versiones, se ha elegido una estrategia de *branching* basada en una rama principal, *main*, y diversas ramas secundarias, cada una de ellas asociada con una determinada funcionalidad o etapa de desarrollo. Una vez finalizado el trabajo en una rama secundaria, ésta se fusionará con la rama principal.

IMPORTANTE: A efectos de revisión y entrega final del código de los proyectos, se ha dado acceso al profesor colaborador de la asignatura, Gregorio Robles Martínez, a los repositorios de GitLab en los que se encuentra alojado. En dichos repositorios se detallan las instrucciones para ejecutar la aplicación en un entorno local. En cualquier caso, si fuese necesario conceder acceso a cualquier

otra persona relacionada con el proyecto, simplemente será necesario enviar un correo electrónico a la cuenta de la UOC del alumno autor del trabajo.

3.2 Desarrollo

3.2.1 Backend.

Para la implementación del *backend* se ha elegido una arquitectura hexagonal. El punto de acceso a la API es un controlador *Spring MVC*, la clase *PdplannerRESTController*, que funciona a modo de adaptador de entrada. Esta clase será la encargada de recoger las peticiones del exterior, dirigidas a alguno de los *endpoints* definidos, redireccionarlas a un determinado servicio y, finalmente, generar una respuesta que se ha de enviar al cliente. En la figura 25 se puede ver un detalle de la mencionada clase:

Figura 25. Detalle de la clase PdplannerRESTController.

```java
@Log4j2
@RestController
@RequestMapping("/api/v1/pdplanner")
@RequiredArgsConstructor
@PreAuthorize("hasAnyRole('EMPLOYEE', 'MANAGER', 'ADMIN')")
public class PdplannerRESTController {

    private final PdplannerService pdplannerService;


    no usages  pdiazmz
    @GetMapping("/users")
    @PreAuthorize("hasAuthority('manager:get')")
    public List<AppUser> findAllAppUsers() {
        log.info("PdPlanner Controller: findAllAppUsers");
        return pdplannerService.findAllAppUsers();
    }
```

Continuando con la descripción de la arquitectura hexagonal y siguiendo el camino de una supuesta petición que llega a la API, se encuentra el puerto de entrada *PdplannerService*, que es una *interface* que da acceso a las clases que forman la lógica del negocio y de la cual hace uso el controlador antes descrito. Esta *interface* está implementada por la clase *PdplannerServiceImpl*, servicio Spring, que será la encargada de realizar todas las operaciones necesarias para devolver una respuesta al controlador. En la figura 26 se muestra un detalle del método *createAppUser* de este servicio a modo de ejemplo:

Figura 26. Detalle de la clase PdplannerServiceImpl

```java
/**
 * Saves an AppUser into the database. If the email exists in the database throws an EmailAlreadyExistsException.
 * This method is also in charge of encoding the password using a PasswordEncoder.
 *
 * @author pdiazmz
 * @param appUser AppUser
 * @return The id of the AppUserEntity/AppUser saved in the database
 */
3 usages  pdiazmz
@Override
public Long createAppUser(AppUser appUser) {
    log.info("PdPlanner Service: createAppUser");
    Optional<AppUser> appUserWithTheSameEmail = appUserRepository.findAppUserByEmail(appUser.getEmail());
    if(appUserWithTheSameEmail.isPresent()) throw new EmailAlreadyExistsException();
    var encodedPassword = passwordEncoder.encode(appUser.getPassword());
    appUser.setPassword(encodedPassword);
    Long id = appUserRepository.createAppUser(appUser);
    return id;
}
```

Como se puede observar en la figura, este método está encargado de comprobar que no existe otro usuario con el mismo email, y en caso de no ser así, codificar la contraseña y guardarlo en la base de datos.

Los servicios acceden a las bases de datos mediante el uso de los llamados *repositorios* (en inglés *repository*), ofrecidos por *Spring*. Su uso se concreta a través de varias *interfaces*, una por cada entidad definida en el sistema, que operan a modo de puertos de salida. En ellas se define toda la funcionalidad necesaria para la persistencia de datos, figura 27.

Figura 27. Detalle interface AppUserRepository.

```java
public interface AppUserRepository {

    1 usage  1 implementation  pdiazmz
    List<AppUser> findAllAppUsers();
    4 usages  1 implementation  pdiazmz
    Optional<AppUser> findAppUserById(Long id);
    2 usages  1 implementation  pdiazmz
    Optional<AppUser> findAppUserByEmail(String email);
    2 usages  1 implementation  pdiazmz
    Optional<AppUserEntity> findByEmailSecurityService(String email);
    5 usages  1 implementation  pdiazmz
    Long createAppUser(AppUser appUser);
    1 usage  1 implementation  pdiazmz
    void updateAppUser(AppUser appUser);
    1 usage  1 implementation  pdiazmz
    void deleteAppUser(Long id);
}
```

La implementación de estas *interfaces* tiene lugar en clases como *AppUserRepositoryImpl*, fuera ya del dominio, que a su vez son las que harán uso de un *Spring repository*. Estos *repository* son *interfaces* que heredan de *JpaRepository* y que ofrecen, *out-of-the-box*, la mayor parte de la funcionalidad necesaria para operar sobre una determinada base de datos, previamente configurada en el archivo *application.properties*.

Toda la funcionalidad descrita para los repositorios *Spring*, y materializada a través de *JpaRepository*, está soportada por el *framework Spring Data JPA (Jakarta Persistence API)* [30]. Y los repositorios, como ya se ha comentado, están preparados para ofrecer la mayor parte de la funcionalidad de acceso a una base de datos. Sin embargo, no es su única capacidad, también brindan la posibilidad de declarar, siguiendo una serie de patrones preestablecidos, un amplio abanico de métodos de acceso, así como la opción de generar métodos que obedecen a consultas totalmente personalizadas por los usuarios. En las figuras 28 y 29 se pueden apreciar algunos detalles relativos a las funcionalidades descritas.

Figura 28. Detalle de la clase PlanningRepositoryImpl.

```java
@Component
@RequiredArgsConstructor
public class PlanningRepositoryImpl implements PlanningRepository {

    private final SpringPlanningRepository repository;

    1 usage  pdiazmz
    @Override
    public List<Planning> findAllPlannings() {
        return repository.findAll().stream().map(PlanningEntity::toDomain).collect(Collectors.toList());
    }
```

Figura 29. Detalle de la clase SpringWorkShiftDateRepository.

```java
@Repository
public interface SpringWorkShiftDateRepository extends JpaRepository<WorkShiftDateEntity, Long> {

    1 usage  pdiazmz
    List<WorkShiftDateEntity> findAllByPlanningUser_AppUser_IdAndPlanningUser_Planning_Id(Long idAppuser, Long idPlanning);

    1 usage  pdiazmz
    List<WorkShiftDateEntity> findAllByPlanningUserPlanningId(Long idPlanning);

    1 usage  pdiazmz
    @Transactional
    void deleteByPlanningUserPlanningId(Long idPlanning);
}
```

Son las clases como la detallada en la figura 28 las que conforman los adaptadores de salida.

En el corazón de la arquitectura hexagonal se encuentran las clases que conforman el modelo de negocio. En ellas se contiene la información que se desea manejar para solucionar un problema concreto, en este caso la confección de planificaciones de horarios (figura 30).

Figura 30. Detalle de la clase WorkShift (modelo).

```
public class WorkShift {

    private Long id;
    private String code;
    private LocalTime start;
    private LocalTime end;
    private LocalTime startBreak;
    private LocalTime endBreak;
```

Así pues, esta información será intercambiada con los clientes de la API y persistida en una base de datos. En ambos casos se necesita mapear esta información a otros formatos o modelos y, también en ambos casos, este proceso es realizado por Spring. En concreto, el intercambio de información con los clientes se realizará en formato *JSON* [31].

Pero merece especial atención la transformación necesaria para llevar a cabo la persistencia de datos en *PostgreSQL*. Como ya se ha comentado con anterioridad, *PostgreSQL* en un sistema gestor de bases de datos de tipo relacional. Por otra parte, *Java* es un lenguaje de programación orientado a objetos. Así, se deben transformar objetos en tablas para guardar información y tablas en objetos para recuperar esa información. Todo este proceso se realiza de forma automática con el apoyo de *Spring Data JPA,* que implementa la especificación *Jakarta Persistence API* [32] utilizando, en este caso, el *framework Hibernate ORM* (*Object Relational Mapping*) [33].

Todo este proceso se lleva a cabo mediante las correspondientes anotaciones en las clases que conforman las llamadas *entidades*. En ellas se podrán declarar campos, restricciones, claves primarias y foráneas y cualquier otro dato necesario para realizar el mapeo objeto-relacional. En la figura 31 se presenta un detalle de una entidad "*Jakarta"*, donde se detallan los campos, claves y relaciones con otras entidades necesarias para la persistencia de usuarios en el sistema.

Figura 31. Detalle de la clase AppUserEntity.

```
@Entity
public class AppUserEntity implements DomainTranslatable<AppUser>, UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String firstName;
    private String firstLastName;
    private String secondLastName;
    @Column(name = "email", nullable = false, unique = true)
    private String email;
    private Integer totalHoursYear;
    @Enumerated(EnumType.STRING)
    private Role role;
    private String password;
    @OneToMany(mappedBy = "user")
    private List<TokenEntity> tokens;
```

Lo que se ha descrito hasta ahora es la transformación de las llamadas entidades, pero no las clases del dominio, cuya información será la que

finalmente se intercambie con los clientes. Debe existir, por tanto, un paso intermedio que transforme las clases del dominio, en el corazón de la arquitectura hexagonal, en entidades preparadas para la persistencia, en la periferia de la arquitectura y que dan soporte a los correspondientes adaptadores de salida.

Todo el procedimiento de transformación de clases de dominio a entidades y viceversa se ejecuta dentro de las implementaciones de las clases de tipo *Entity*. Mediante este proceso se consiguen varios objetivos muy interesantes. Por un lado, se controla la información que se envía a los clientes, que no tiene por qué ser la misma que se guarda en la base de datos. Por ejemplo, en la transformación entre la entidad *AppUserEntity* y la correspondiente clase del dominio, *AppUser*, no se incluye el campo *password* por motivos obvios. Por otro lado, la utilización de clases diferentes a las que se persisten permite modificar la cantidad de información que se transfiere a través de las redes de comunicación, punto crítico en el rendimiento de sistemas distribuidos. Esta solución aportada se corresponde con el llamado patrón DTO (*Data Transfer Object*) [42]. Aunque no se ha utilizado en el presente proyecto, es común el empleo del sufijo “Dto” en la nomenclatura y también el uso de *Java Records,* desde su implementación en *Java* 14, para la creación de estas clases. En la figura 32 se puede ver un detalle de los métodos creados para la transformación de DTO’s en entidades relativos a los usuarios de la aplicación.

Figura 32. Detalle de los métodos para las transformaciones AppUser – AppUserEntity incluidos en la clase AppUserEntity.

```
    public AppUser toDomain() {
        return AppUser.builder()
                .id(this.getId())
                .firstName(this.getFirstName())
                .firstLastName(this.getFirstLastName())
                .secondLastName(this.getSecondLastName())
                .email(this.getEmail())
                .totalHoursYear(this.getTotalHoursYear())
                .role(this.getRole())
//                .password(this.getPassword())
                .build();
    }

    pdiazmz
    public static AppUserEntity fromDomain(AppUser appUser) {
        if(appUser == null) return null;

        return AppUserEntity.builder()
                .id(appUser.getId())
                .firstName(appUser.getFirstName())
                .firstLastName(appUser.getFirstLastName())
                .secondLastName(appUser.getSecondLastName())
                .email(appUser.getEmail())
                .totalHoursYear(appUser.getTotalHoursYear())
                .role(appUser.getRole())
                .password(appUser.getPassword())
                .build();
    }
```

En lo referente al manejo de excepciones, se han creado clases correspondientes a las específicas de la aplicación, como la que se puede

producir al intentar guardar en la base de datos un usuario con un email que ya existe, violando de esta forma la restricción impuesta al campo, de tipo *UNIQUE*.

A su vez, se ha creado la clase *PdplannerExceptionHandler* para manejar las excepciones producidas en los controladores, figura 33. Mediante la anotación *@RestControllerAdvise* se indica a Spring la función que tendrá esta clase. El funcionamiento es sencillo. *PdplannerExceptionHandler* implementa una serie de métodos que serán ejecutados en el momento en el que se lanza una excepción del tipo indicado en *@ExceptionHandler*. En el método correspondiente se detallan las acciones que se deben llevar a cabo, siendo de especial importancia construir una respuesta adecuada para enviar al cliente de la API.

Figura 33. Detalle de la clase PdplannerExceptionHandler

```java
@Log4j2
@RestControllerAdvice
public class PdplannerExceptionHandler {


    no usages  pdiazmz
    @ExceptionHandler({EmailAlreadyExistsException.class})
    public ResponseEntity<Object> handleEmailAlreadyExistsException(final EmailAlreadyExistsException exception) {
        log.error("Error: EmailAlreadyExistsException");
        return ResponseEntity.status(HttpStatus.CONFLICT).body("EMAIL_ALREADY_EXISTS");
    }
```

3.2.2 Backend. Seguridad.

Hasta el momento se ha descrito la funcionalidad del *backend* del sistema en desarrollo, es decir, cómo son tratadas las peticiones externas y persistida la información. Sin embargo, hay una pieza fundamental que hace falta para cumplir con los requisitos de la aplicación, y es dotarla de un sistema de autenticación y autorización. Se debe conceder acceso al sistema solamente a determinados usuarios y a cada uno de ellos se debe permitir utilizar determinada funcionalidad en relación con el grupo al que pertenezca (empleado, manager o administrador).

Para solucionar la problemática anterior se ha optado por el uso de *Spring Security* y *JWT* (*JSON Web Tokens*) [35], que tal y como se puede leer en Wikipedia [36], “ *es un estándar abierto basado en JSON […] para la creación de tokens de acceso que permiten la propagación de identidad y privilegios*”.

El primer paso para la implementación de seguridad con *tokens* es la creación de un filtro que analice las peticiones entrantes, compruebe la existencia de un *token* y su validez, actualice el llamado en *Spring Security* [34] “*SecurityContext*”, que contiene la autenticación del usuario para la petición en curso, y transfiera dicha petición. En el sistema, este filtro se ha creado en la clase *JwtAuthenticationFilter* mediante la especialización de la clase *OncePerRequestFilter*, y la sobreescritura del método *doFilterInternal* con las especificaciones oportunas. Se ha de tener en cuenta que las API REST no

tienen estado y, por este motivo, este filtro funcionará para todas las peticiones entrantes, que deberán adjuntar el correspondiente token para que puedan ser validadas y transferidas.

Por otro lado, en el citado filtro se hace uso de la *interface UserDetailsService,* provista por *Spring Security* para la recuperación de los datos de usuario de la base de datos. Previamente se debe crear una entidad de usuario para este fin - en este caso se ha utilizado la clase *AppUserEntity*-. Para ello la clase deberá implementar toda la funcionalidad definida en la *interface UserDeatils*, que, entre otros, exige métodos para la recuperación del nombre de usuario, contraseña, rol etc. En la figura 34 se muestra en detalle la implementación de la *interface UserDetails.*

Figura 34. Detalle de la implementación de UserDetails en AppUserEntity.

```
1 usage  pdiazmz
@Override
public Collection<? extends GrantedAuthority> getAuthorities() { return role.getAuthorities(); }

 pdiazmz
@Override
public String getUsername() { return email; }

no usages  pdiazmz
@Override
public boolean isAccountNonExpired() { return true; }

no usages  pdiazmz
@Override
public boolean isAccountNonLocked() { return true; }

no usages  pdiazmz
@Override
public boolean isCredentialsNonExpired() { return true; }

 pdiazmz
@Override
public boolean isEnabled() { return true; }

// A getPassword method is also needed but there in no need to implement it due to the use of Lombok
// and the presence of the password field.
// Otherwise, getPassword method must be declared.
```

Asimismo, se debe proporcionar una implementación de la *interface UserDetailsService*, lo que se hará en una nueva clase a la que se ha denominado *ApplicationConfig*. La función *lambda* que devuelve el método que se presenta en la figura 35 es la implementación del método *loadUserByUserName* que define la *interface UserDetailsService*, y mediante el cual se podrán recuperar los datos del usuario que trata de acceder a la API.

Figura 35. Implementación de la interface UserDetailsService.

```
@Bean
public UserDetailsService userDetailsService() {
    // The lambda function is the implementation of loadUserByUsername method
    // that must be implemented in any class using UserDetailsService interface
    // So, in the end, we return an UserDetailsService object
    return username -> userRepository.findByEmailSecurityService(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
}
```

En la clase *ApplicationConfig* también se ponen a disposición de *Spring Security* otros *beans* necesarios para la implementación de la capa de seguridad.

En todo momento, el manejo de *tokens* se realiza mediante la clase *JwtServiceImpl*, que hace uso de la librería *jjwt* [38]. En esta clase se encuentra la funcionalidad necesaria para recuperar la información del usuario codificada en el token, así como para generar nuevos *tokens* a partir de los datos de un usuario y validarlos.

Una vez creado el filtro, clase *JwtAuthenticationFilter*, debe ponerse en funcionamiento. Se hará mediante la creación de la clase *SecurityConfig*, a la que se añadirán las anotaciones *@Configuration* y *@EnableWebSecurity,* con el fin de que sea utilizada por *Spring Security* para la implementación de la seguridad que se ha desarrollado. En concreto, *Spring Security* necesita un *bean* de tipo *SecurityFilterChain* en el que se ha de detallar todo el proceso por el que han de pasar todas las peticiones entrantes. En la figura 36 se puede ver un detalle de la creación del mencionado *bean*.

Figura 36. Detalle de la implementación de SecurityFilterChain.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {


    http
            .csrf() CsrfConfigurer<HttpSecurity>
            .disable() HttpSecurity
            .cors() CorsConfigurer<HttpSecurity>
            .and() HttpSecurity
            .authorizeHttpRequests() AuthorizationManagerRequestMat...
            .requestMatchers( ...patterns: "/api/v1/pdplanner/auth/**")// white list
            .permitAll()
            .anyRequest()   // Any other request do need authorization
            .authenticated()
            .and() HttpSecurity
            .sessionManagement() SessionManagementConfigurer<HttpSecurity>
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS) // new session for each request
            .and() HttpSecurity
            .authenticationProvider(authenticationProvider)
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
            .logout() LogoutConfigurer<HttpSecurity>
            .logoutUrl("/api/v1/pdplanner/logout") // in order to require a token
            .addLogoutHandler(logoutService) // The logout mechanism is implemented in LogoutService
            .logoutSuccessHandler(((request, response, authentication) -> SecurityContextHolder.clearContext()));


    return http.build();
}
```

Finalmente, se crea un *endpoint* en un nuevo *controller*, *AuthenticationController,* que se pone a disposición de los clientes para la autenticación en el sistema y que hará uso de la clase *AuthenticationServiceImpl,* donde finalmente reside toda la lógica del proceso (figura 37).

Figura 37. Detalle del método authenticate de la clase AuthenticationServiceIml.

```
@Override
public AuthenticationResponse authenticate(CreateAuthenticationRequest request) {
    authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                    request.getEmail(),
                    request.getPassword()
            )
    );
    // if this point is reached the user is authenticated, otherwise an exception is thrown
    var user = repository.findByEmailSecurityService(request.getEmail()).orElseThrow();
    var jwtToken = jwtService.generateToken(user);
    var token = TokenEntity.builder()
            .user(user)
            .token(jwtToken)
            .tokenType(TokenType.BEARER)
            .revoked(false)
            .expired(false)
            .build();
    revokeAllUserTokens(user);
    tokenRepository.save(token);
    return AuthenticationResponse.builder()
            .token(jwtToken)
            .email(user.getEmail())
            .role(user.getRole().name())
            .displayName(user.getFirstName() + " " + user.getFirstLastName())
            .expiration(jwtService.extractExpirationDate(jwtToken))
            .userId(user.getId())
            .build();
}
```

Una vez autenticado un usuario, la aplicación debe contar con algún tipo de mecanismo que impida el acceso a determinadas partes de ésta dependiendo del rol del susodicho usuario. Es decir, se ha de implementar un sistema de autorización.

Como primer paso para implementar un sistema de autorización, se han diseñado un conjunto de roles y permisos asociados a cada rol que se pueden ver en las enumeraciones *Role* y *Permission*. En la figura 38 se muestra un detalle del método mediante el cual se pueden obtener los permisos asociados a un determinado rol.

Figura 38. Obtención de permisos asociados a un rol. Enumeración Role.

```
public List<SimpleGrantedAuthority> getAuthorities() {
    var authorities = getPermissions() Set<Permission>
            .stream() Stream<Permission>
            .map(permission -> new SimpleGrantedAuthority(permission.getPermission())) Stream<SimpleGrantedAuthority>
            .collect(Collectors.toList());
    authorities.add(new SimpleGrantedAuthority( role: "ROLE_" + this.name()));
    return authorities;
}
```

A continuación, se deben indicar los permisos que un usuario debe tener para poder acceder a la funcionalidad de la API. Para ello, se puede optar por dos alternativas. En la primera, se introduce la información relativa a *endpoints* y permisos en el *bean* creado con anterioridad, *SecurityFilterChain,* mediante los métodos *requestMatchers* y *hasRole* o *hasAnyRole* así como *hasAuthority* o *hasAnyAuthority* entre otros. En la segunda opción, por la que se ha optado, se utilizan anotaciones (*@PreAuthorize*) en la clase *PdplannerRESTController* y los métodos asociados a los diferentes *endpoints* para conseguir el mismo resultado.

Figura 39. Utilización de la anotación @PreAuthorize en PdplannerRESTController.

```java
@Log4j2
@RestController
@RequestMapping("/api/v1/pdplanner")
@RequiredArgsConstructor
@PreAuthorize("hasAnyRole('EMPLOYEE', 'MANAGER', 'ADMIN')")
public class PdplannerRESTController {

    private final PdplannerService pdplannerService;

    no usages  pdiazmz
    @GetMapping("/users")
    @PreAuthorize("hasAuthority('manager:get')")
    public List<AppUser> findAllAppUsers() {
        log.info("PdPlanner Controller: findAllAppUsers");
        return pdplannerService.findAllAppUsers();
    }
```

Como se puede observar en la figura 39, la clase es accesible, en general, a los roles *'EMPLOYEE', 'MANAGER' Y 'ADMIN.* Por otra parte, la utilización del método *findAllAppUsers* requiere el permiso *'manager:get'*.

El hecho de utilizar una u otra opción, autorización a nivel de configuración o a nivel de métodos mediante anotaciones, vendrá definido por las necesidades de cada aplicación. Obviamente, ambas tienen sus ventajas y desventajas. Por ejemplo, en caso de necesitar controlar el acceso de forma detallada para cada método o preferir mantener las reglas de autorización lo más cerca posible a los *endpoints* de la API para facilitar su lectura y mantenimiento, motivo este último por el cual se ha decidido utilizar este método en el presente proyecto, podría ser más interesante utilizar el sistema basado en anotaciones. En caso de necesitar mantener las reglas de autorización de forma centralizada, si el mapeo entre *endpoints* y permisos no es complicado o si se desea aplicar reglas de forma global, el método de autorizaciones a nivel de configuración puede ser ventajoso.

3.2.3 Frontend

Tal y como se ha explicado con anterioridad, se dice que Angular es un *framework* orientado a componentes. El *frontend* de la aplicación se ha construido en torno a una colección de dichos componentes, que serán los encargados de la interacción con los usuarios finales. En ellos se hará uso de varios servicios, mediante los cuales se implementarán diferentes

funcionalidades, como interacción con la API, manejo de errores, interceptores de peticiones http etc.

A modo de ejemplo se analizará la funcionalidad del componente *login.*

El componente *login* es el encargado de mostrar al usuario un formulario en el que se puedan introducir los datos de autenticación y acceder al sistema.

Figura 40. Detalle de la plantilla html del componente login.

```
<form (ngSubmit)="onSubmit()" #f="ngForm">
  <div id="user-data"
       ngModelGroup="appUser">
    <!-- userName form control -->
    <div class="mt-3">
      <label for="email">Email:</label>
      <input type="email" id="email" class="form-control"
             [(ngModel)]="loginValues.email" name="email" #email="ngModel" required email>
    </div>
    <span class="help-block" *ngIf="!email.valid && email.touched">Por favor, introduce un nombre.</span>
    <!-- password form control -->
    <div class="mt-3">
      <label for="password">Contraseña:</label>
      <input type="password" id="password" class="form-control"
             [(ngModel)]="loginValues.password" name="password" required>
    </div>
    <!-- submit button -->
    <button class="btn btn-primary mt-3"
            type="submit" [disabled]="!f.valid">Enviar</button>
  </div>
```

Figura 41.Detalle del controlador del componente login.

```
export class LoginComponent {

  loginValues: AuthRequest = {email: "", password: ""}
  invalidLogin: boolean = false;
  error?: string;



  constructor(private route: ActivatedRoute, private router: Router, private authService: AuthService) {}

  onSubmit() {
    this.authService.signin(this.loginValues).subscribe({
      next: data => {
        console.log('on singin...');
      },
      error: errorMessage => {
        this.error = errorMessage;
        console.log(errorMessage);
      },
      complete: () => {
        console.log('signin proccess completed!');
        this.router.navigate(['/pdplanner']);
      }
    })

  }
```

Figura 42. Detalle del servicio AuthService.

```
authResponseSubject = new BehaviorSubject<AuthResponse>(new AuthResponse('', '', '', '', new Date()));

authData?: AuthResponse;


constructor(private http: HttpClient) { }

signin(authRequest: AuthRequest) {
  return this.http.post<AuthResponse>(this.signinUrl, authRequest)
    .pipe(
      catchError(errorRes => {
        let errorMessage = 'Algo raro ha pasado! Ponte en contacto con una administrador!';
        if(errorRes.status === 403) errorMessage = 'Los datos de acceso no son válidos!';
        if(!errorRes.error || !errorRes.error.error) {
          return throwError(() => new Error(errorMessage));
        }
        return throwError(() => new Error(errorRes.error.error.message));
      }),
      tap(resData => {
        this.authResponseSubject.next(resData);
        localStorage.setItem('authorizationData', JSON.stringify(resData));
      })
    )
}
```

Como se puede observar en la figura 40, se ha utilizado *two-way data binding* mediante la directiva *ngModel* para vincular el formulario al objeto de tipo *AuthRequest,* definido en el controlador del componente, en el que se guardarán los datos de autenticación en la API.

En la figura 41 se puede apreciar la implementación del método *onSubmit* (perteneciente al controlador del componente), que se ejecutará cuando el usuario pulse el botón correspondiente en la plantilla. Este método hace uso del servicio *authService.* En este servicio se define el método que será encargado de lo siguiente:

- Generar la petición de autenticación a la API (figura 42),
- Utilizar un objeto de tipo *BehaviorSubject* con el fin de facilitar las suscripciones desde cualquier parte de la aplicación para obtener la información de los datos de acceso a la API, donde se incluye el *token* enviado por el *backend*,
- Guardar la información de autenticación del usuario en el almacenamiento local de cara a la implementación de una funcionalidad de *auto-login* para casos de recarga de la página, por ejemplo.

El presentado anteriormente es un patrón recurrente en el frontend.

Mención especial merece el componente *planning-table*, donde reside la lógica correspondiente a las funcionalidades relativas a la confección de horarios y supervisión de restricciones de planificaciones.

Con relación a dicho componente, cabe destacar que la elección del diseño para la base de datos presenta algunos inconvenientes de cara al manejo de los datos asociados a las planificaciones y los horarios. Así, se ha decidido implementar un método (figura 43) que permita mapear la estructura de datos obtenida de la base de datos, un array de objetos de tipo *WorkShiftDate*, a una estructura

basada en objetos de tipo *PlanningUserTable*, compuestos por un campo *PlanningUser*, donde se guarda la información relativa al *Planning* y el *AppUser* y un array, *shifts*, en el que se guardarán los diferentes turnos, *WorkShifts*, que le sean asignados. El array está ordenado de tal forma que sus posiciones se corresponden con los sucesivos días del año. De esta forma será mucho más sencillo iterar sobre los diferentes elementos para poder presentar y recuperar la información introducida por el usuario. La posibilidad de acceso por posición y su bajo coste, la posibilidad de iterar sobre un array, su tamaño contenido independientemente de su aprovechamiento y el número de posiciones vacías, y el hecho de que estemos trabajando en un proyecto a pequeña escala, es la razón por la que esta estructura de datos ha sido elegida para guardar la información de los turnos de trabajo.

Figura 43. Método mapWorkShiftDatesToPlanningUserTables de planning-table component.

```
/**
 * Maps tha values obtained from the database as an array of WorkShiftDate (the workShiftDates field) into
 * an easier to handle PlanningUserTable array (the planningUserTables field).
 *
 * @author pdiazmz
 */
private mapWorkShiftDatesToPlanningUserTables() {

  let key: number | undefined;
  let value = 0;
  // This map keeps the position of every PlanningUser in the PlanningUserTables array
  let planningUserTablePosition = new Map();

  // Push into planningUserTables every planningUser in the planning
  for(let planningUser of this.planningUsers) {
    let planningUserTable = new PlanningUserTable(planningUser);
    this.planningUserTables.push(planningUserTable);
    key = planningUser.id;
    planningUserTablePosition.set(key, value);
    value++;
  }

  // If there is any info in workShiftDates object from the DB
  if(this.workShiftDates.length > 0) {
    for(let i = 0; i < this.workShiftDates.length; i++) {

      // Add workshift to planningUserTable shifts array
      let planningUser = this.workShiftDates[i].planningUser;
      let workShift = this.workShiftDates[i].workShift;

      let dateS = this.workShiftDates[i].date;
      let dateString = dateS?.toString();
      if(dateString !== undefined) {
        let date = this.parseDate(dateString);
        key = this.workShiftDates[i].planningUser?.id;

        if(planningUser !== undefined && workShift !== undefined && date !== undefined && key !== undefined) {
          if(planningUserTablePosition.has(key)) {
            this.planningUserTables[planningUserTablePosition.get(key)].shifts[this.dateMonthToDateYear(date)] = workShift;
          }
        }
```

En el mencionado componente se incluyen además una serie de métodos de soporte para efectuar este mapeo, así como otros dedicados a la recopilación de información sobre una planificación y comprobación de restricciones.

3.2.4 Frontend. Seguridad.

En lo que respecta a la seguridad, además de la utilización de la directiva **ngIf*, con fines puramente relacionados con la presentación, se ha restringido el acceso a los usuarios con determinados roles a diferentes partes de la aplicación mediante el uso de las llamadas *"guards*", funciones que se asocian a determinadas rutas para controlar el acceso a las mismas. La que se presenta en la figura 44 será la función utilizada para comprobar si el usuario posee un *token* de acceso en el almacenamiento local y, en caso negativo, redirigirlo al componente *login*.

Figura 44. Detalle función authGuardFn.

```
export const authGuardFn: CanActivateFn = () => {
  const router = inject(Router);
  const authDataString = localStorage.getItem('authorizationData');
  let token;
  let role;
  if(authDataString !== null) {
    const authData = JSON.parse(authDataString);
    token = authData.token;
    role = authData.role;
  }

  if(!token) {
    router.navigate(['/login']);
    return false;
  }
  return true;
}
```

En la figura 45 se puede apreciar la utilización de las diferentes *guards* declaradas en la aplicación en el módulo donde se han registrado las rutas.

Figura 45. Detalle módulo AppRoutingModule.

```
const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full'},
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'pdplanner', canActivate: [authGuardFn], component: PdplannerComponent, children: [
    { path: '', component: WelcomeComponent },
    { path: 'appuser-create', canActivate: [employeeGuardFn, managerGuardFn], component: AppuserCreateComponent },
    { path: 'appusers', component: AppusersComponent },
    { path: 'planning-create', canActivate: [employeeGuardFn], component: PlanningCreateComponent },
    { path: 'shift-create', canActivate: [employeeGuardFn], component: ShiftCreateComponent },
    { path: 'workshifts', component: ShiftsComponent },
    { path: 'workshift/:id', component: ShiftCreateComponent },
    { path: 'plannings', component: PlanningsComponent },
    { path: 'plannings/:id', component: PlanningDetailsComponent },
    { path: 'planning-table/planning/:id', component: PlanningTableComponent },
    { path: 'appuser-mod/:id', canActivate: [employeeGuardFn, managerGuardFn], component: AppuserModComponent }
  ] }
];
```

Por último, también relacionado con la seguridad del sistema, es interesante destacar la utilización de *interceptors* para el tratamiento de peticiones http

(*HttpInterceptor*). Mediante este tipo de construcciones se pueden manipular todas las peticiones y respuestas http para, por ejemplo, declarar una cabecera que contenga el *token* de autenticación del usuario a fin de poder ser validado por la API (figura 46) o manejar las excepciones que se reciban como respuestas.

Figura 46.Detalle de AuthInterceptorService.

```
@Injectable()
export class AuthInterceptorService implements HttpInterceptor {

  constructor(private authService: AuthService) { }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    return this.authService.authResponseSubject.pipe(take(1), exhaustMap(authData => {
      if(authData.token === '') {
        return next.handle(req);
      }
      const reqWithToken = req.clone({ headers: new HttpHeaders().append('Authorization', 'Bearer ' +  authData.token)})
      return next.handle(reqWithToken);
    }))

  }
}
```

3.2.5 Pruebas

Se han desarrollado ejemplos de test unitarios para la clase *PdplannerRESTController* y para la clase *PdplannerServiceImpl*. Asimismo, se ha incluido una muestra de test de integración para el repositorio *AppUserRepository*. Para confeccionar los test se han utilizado las librerías *Junit* [50], *AssertJ* [51] y *Mockito* [52], todas ellas contenidas en las dependencias generadas por defecto por *Spring Boot*. A continuación, se muestran algunas capturas de pantalla de los test mencionados y sus resultados.

Figura 47. Detalle de test shouldFindAllAppUsers de PdplannerRESTControllerTest.

```
@Test
@Tag("Unit")
@DisplayName("Testing PdPlannerRESTController findAllAppUsers method...")
@WithMockUser(username = "manager", authorities = {
        "manager:get"
})
void shouldFindAllAppUsers() throws Exception {

    AppUser appUser1 = AppUser.builder().id(1L).firstName("TestName").firstLastName("FirstLastNameTest")
            .secondLastName("SecondLastNameTest").email("email@test.ts").totalHoursYear(1800).role(Role.ADMIN)
            .password("testPassword").build();

    AppUser appUser2 = AppUser.builder().id(2L).firstName("TestName2").firstLastName("FirstLastNameTest2")
            .secondLastName("SecondLastNameTest2").email("email2@test.ts").totalHoursYear(1800).role(Role.EMPLOYEE)
            .password("testPassword2").build();

    List<AppUser> appUsers = List.of(appUser1, appUser2);

    given(pdplannerService.findAllAppUsers()).willReturn(appUsers);
    this.mockMvc.perform(get( urlTemplate: "/api/v1/pdplanner/users")
                    .contentType(MediaType.APPLICATION_JSON)
                    .with(csrf()).with(SecurityMockMvcRequestPostProcessors.user( username: "manager").roles("manager:get")))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$.size()", Matchers.is( value: 2)))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[0].email", Matchers.is( value: "email@test.ts")))
            .andExpect(MockMvcResultMatchers.jsonPath( expression: "$[1].email", Matchers.is( value: "email2@test.ts")));
}
```

Figura 48. Detalle de test shouldCreateAppUser de PdplannerServiceImplTest.

```
@Test
@Tag("Unit")
@DisplayName("Testing PdPlannerService createAppUser method...")
void shouldCreateAppUser() {
    Long id = underTest.createAppUser(appUser1);
    ArgumentCaptor<AppUser> appUserArgumentCaptor = ArgumentCaptor.forClass(AppUser.class);
    verify(appUserRepository, VerificationModeFactory.times( wantedNumberOfInvocations: 1)).createAppUser(appUserArgumentCaptor.capture());
    AppUser capturedAppUser = appUserArgumentCaptor.getValue();
    assertThat(capturedAppUser).isEqualTo(appUser1);
    assertThat(id).isEqualTo( expected: 1L);
    verify(appUserRepository, VerificationModeFactory.times( wantedNumberOfInvocations: 1)).createAppUser(appUser1);
    verify(appUserRepository, VerificationModeFactory.times( wantedNumberOfInvocations: 1)).findAppUserByEmail(appUser1.getEmail());

}
```

Figura 49. Detalle de test shouldFindAppUserById de AppUserRepositoryIntegrationTest.

```
@Test
@DisplayName("Testing AppUserRepository findAppUserById (Integration test)")
void shouldFindAppUserById() {

    AppUser appUser = AppUser.builder().firstName("TestName").firstLastName("FirstLastNameTest").email("email@test.ts")
            .role(Role.ADMIN).password("testPassword").build();

    AppUser appUser2 = AppUser.builder().firstName("TestName2").firstLastName("FirstLastNameTest2").email("email2@test.ts")
            .role(Role.ADMIN).password("testPassword2").build();

    AppUserEntity appUserEntity = AppUserEntity.fromDomain(appUser);
    entityManager.persist(appUserEntity);
    AppUser appUserPersisted = appUserRepository.findAppUserById(1L).orElse( other: null);

    AppUserEntity appUserEntity2 = AppUserEntity.fromDomain(appUser2);
    entityManager.persist(appUserEntity2);
    AppUser appUserPersisted2 = appUserRepository.findAppUserById(2L).orElse( other: null);

    AppUser appUserPersisted3 = appUserRepository.findAppUserById(3L).orElse( other: null);

    assertThat(appUserPersisted.getId()).isEqualTo( expected: 1L);
    assertThat(appUserPersisted.getFirstName()).isEqualTo(appUser.getFirstName());
    assertThat(appUserPersisted.getEmail()).isEqualTo(appUser.getEmail());

    assertThat(appUserPersisted2.getId()).isEqualTo( expected: 2L);
    assertThat(appUserPersisted2.getFirstName()).isEqualTo(appUser2.getFirstName());
    assertThat(appUserPersisted2.getEmail()).isEqualTo(appUser2.getEmail());

    assertThat(appUserPersisted3).isEqualTo( expected: null);
}
```

Figura 50. Detalle resultados test controller.

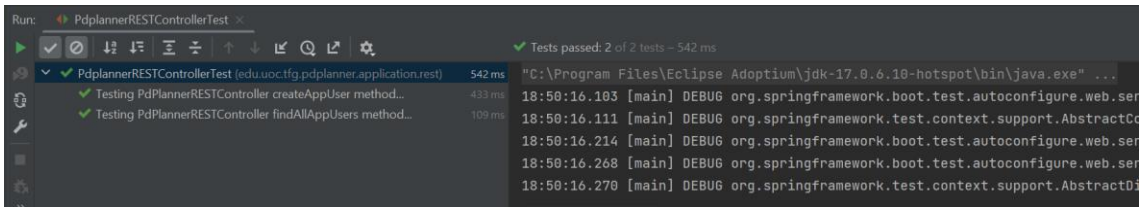


Figura 51. Detalle resultados test Service.

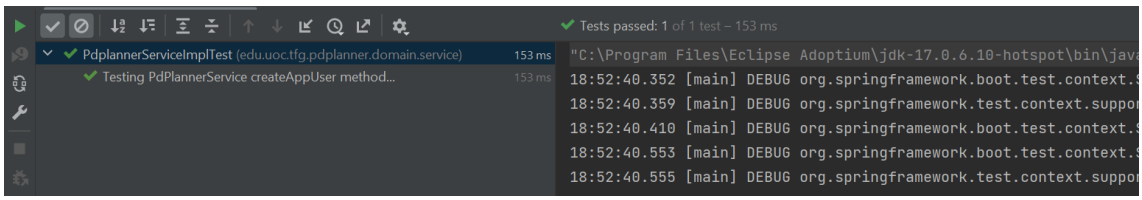
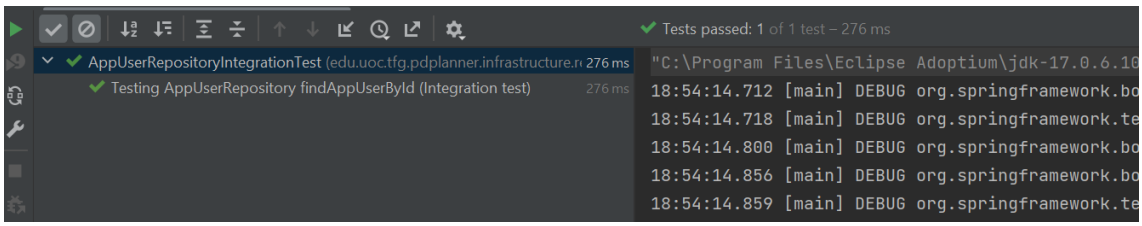


Figura 52. Detalle resultados test de integración Repository.



Por otra parte, en el frontend se ha utilizado *Jasmine* [49] como herramienta para la creación de test. A continuación se muestran ejemplos de test y sus resultados.

Figura 53. Detalle test auth service.

```
beforeEach(() => {
  httpClientSpy = jasmine.createSpyObj('HttpClient', ['post']);
  service = new AuthService(httpClientSpy as any);
});

it('should return an AuthResponse', (done: DoneFn) => {
  const mockUserCredentials = {
    email: 'rhalford@uoc.pdplanner',
    password: '123'
  };

  const mockResponseLogin = new AuthResponse("eyJhbGiwiYwk",
                                             "rhalford@uoc.pdplanner",
                                             "ADMIN",
                                             "Rob Halford",
                                             new Date(2023, 10, 10)
                                             )

  httpClientSpy.post.and.returnValue(of(mockResponseLogin));


  service.signin(mockUserCredentials).subscribe( response => {
    expect(response).toEqual(mockResponseLogin);
    done();
  })
});
```

Figura 54. Detalle resultados test auth service.

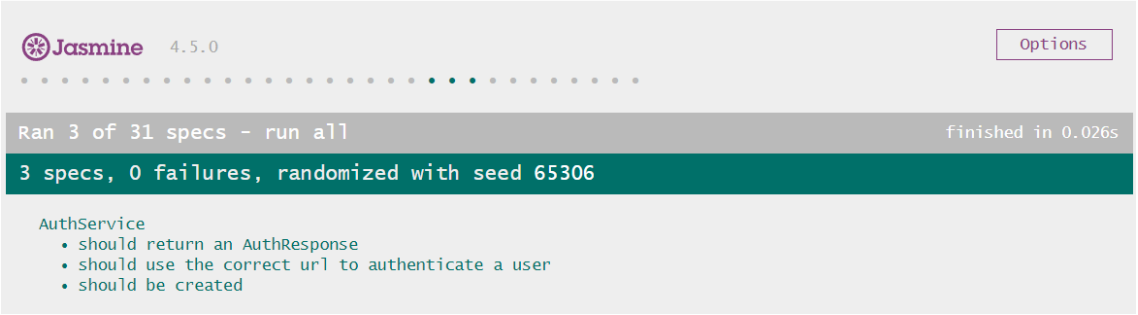
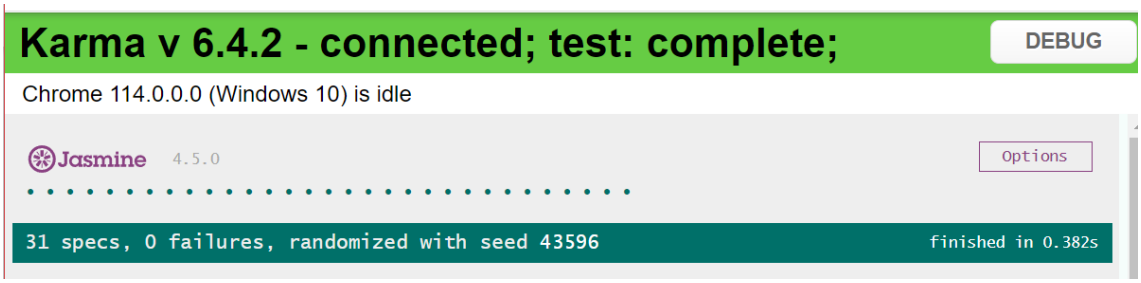
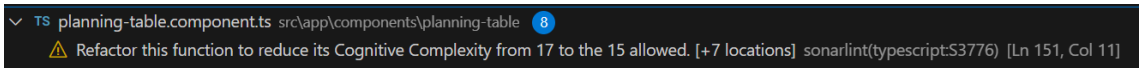


Figura 55. Detalle resultados test frontend.



Por último, se ha utilizado SonarLint [53] como herramienta de análisis estático de código. De esta forma se pueden corregir errores, *bugs*, problemas de seguridad etc. y mejorar la calidad del código resultante.

Figura 56. Ejemplo de uso de SonarLint,

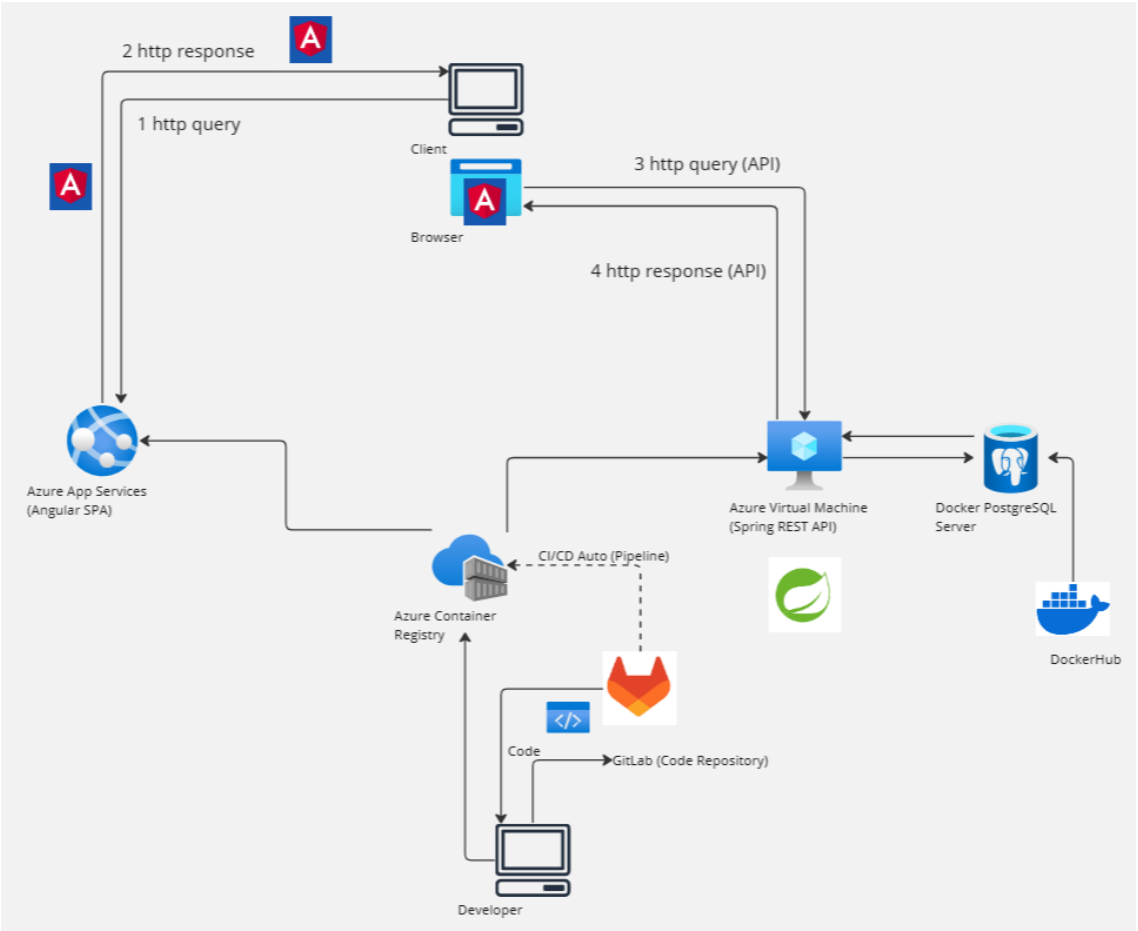


3.3 Despliegue de la aplicación.

Pese a que, finalmente, las limitaciones temporales han impedido el despliegue de la aplicación, se ha decidido mantener la sección con el ánimo de presentar el funcionamiento de la solución planteada. En las conclusiones de la memoria se hace referencia a este hecho tanto en las líneas de trabajo futuras como en los objetivos no conseguidos.

En la figura 50 se incluye un diagrama en el que se trata de explicar, no solo el despliegue que se ha elegido para la aplicación, sino también detalles asociados a su construcción.

Figura 57. Diagrama de despliegue.



Como se puede apreciar, en una primera fase el desarrollador genera el código necesario para construir la aplicación y lo aloja en un repositorio remoto en GitLab. A continuación, bien desde la propia máquina del desarrollador o bien desde GitLab, mediante las funciones de integración y despliegue continuo que ofrece, se generan los contenedores Docker, con el frontend y el backend de la aplicación, que se alojarán en un repositorio de contenedores de Azure (Azure Registry). Estos contenedores serán desplegados en una instancia Azure App Service (frontend) y una Máquina Virtual Azure (backend). Igualmente, en la máquina virtual también se desplegará un contenedor Docker de PostgreSQL importado desde Dockerhub, el repositorio de contenedores de Docker.

Lo más recomendable para la persistencia de datos en un entorno de producción sería utilizar los propios servicios que ofrece Azure, como *Azure Database for PostgreSQL*. No obstante, con el ánimo de simplificar el despliegue, y dada la naturaleza del proyecto, se ha tomado la decisión de utilizar un contenedor.

Una vez que la aplicación haya sido completamente desplegada, cualquier usuario que esté previamente registrado en el sistema podrá hacer uso de ella. Desde su navegador hará una petición http al AppService que le devolverá todo el código necesario para la ejecución del frontend (pasos 1 y 2 de la figura 57). Una vez que se haya hecho esta primera carga ya no será necesario hacer más interacciones con el AppService de Azure para obtener diferentes vistas de la aplicación. A partir de este momento, todos los datos necesarios para la actualización de contenido serán obtenidos de la API REST construida en el *backend* (pasos 3 y 4 de la figura 57, que se repetirán tantas veces como sea necesario).

El despliegue de la aplicación se podrá realizar mediante el uso de contenedores Docker también en local. Cada uno de los proyectos contiene un archivo Dockerfile con el código necesario para generar una imagen de éste. Además, también en cada uno de los proyectos, se ha adjuntado un archivo *docker-compose.yml* que servirá como base para levantar la red de contendores en la máquina, local o remota, que se considere necesario. De esta manera, como único requisito para la ejecución de la aplicación, se necesita una instancia de Docker en ejecución allí donde se pretenda desplegar el producto. En las figuras 58, 59 y 60 se pueden ver los contenidos de los archivos Dockerfile de cada proyecto y el contenido del archivo docker-compose.yml. Los archivos *readme* de cada repositorio contienen las instrucciones precisas, comandos incluidos, para la ejecución de la aplicación.

Figura 58. Dockerfile frontend.

```
#Primera Etapa
FROM node:20.2.0 as build-step
# ARG apiURL
# ENV apiURL=http://localhost:8080/api/v1/pdplanner/
RUN mkdir -p /app
WORKDIR /app
COPY package.json /app
RUN npm install --legacy-peer-deps
COPY . /app
RUN npm run build --prod

#Segunda Etapa
FROM nginx:1.25.0
COPY --from=build-step /app/dist/tfg-pdplanner-front /usr/share/nginx/html
COPY /nginx.conf  /etc/nginx/conf.d/default.conf
```

Figura 59. Dockerfile backend.

```
FROM maven:3.9.2-eclipse-temurin-17-alpine as BUILD

COPY . /usr/src/app
RUN mvn --batch-mode -Dmaven.test.skip=true -f /usr/src/app/pom.xml clean package

FROM eclipse-temurin:17.0.7_7-jdk-alpine
COPY --from=BUILD /usr/src/app/target /opt/target
WORKDIR /opt/target

ENTRYPOINT ["java", "-jar", "pdplanner-1.0-SNAPSHOT.jar"]
```

Figura 60. Archivo docker-compose.yml

```
version: '2'
services:

  db:
    image: postgres:15.3
    platform: linux/amd64
    restart: always
    ports:
      - 54320:5432
    environment:
      POSTGRES_USER: tfg
      POSTGRES_PASSWORD: tfg

  adminer:
    image: adminer:4.8.1
    platform: linux/amd64
    restart: always
    depends_on:
      - db
    ports:
      - 18080:8080

  tfg-pdplanner-back:
    build:
      context: ./tfg-pdplanner-back
    container_name: tfg-pdplanner-back
    environment:
      DB_SERVER: db
      POSTGRES_DB: tfg
      POSTGRES_USER: tfg
      POSTGRES_PASSWORD: tfg
    depends_on:
      - db
    ports:
      - 8080:8080

  tfg-pdplanner-front:
    build:
      context: ./tfg-pdplanner-front
      # args:
      #   apiURL: http://localhost:18081
    container_name: tfg-pdplanner-front
    depends_on:
      - tfg-pdplanner-back
    ports:
      - 4200:80
```

3.3.1 La aplicación en funcionamiento

Seguidamente se muestran algunas capturas de pantalla de la aplicación en funcionamiento.

Figura 61. La aplicación en funcionamiento. Pantalla login.

Figura 62. La aplicación en funcionamiento. Pantalla de listado de empleados.

Figura 63. La aplicación en funcionamiento. Pantalla de alta de nuevo turno.

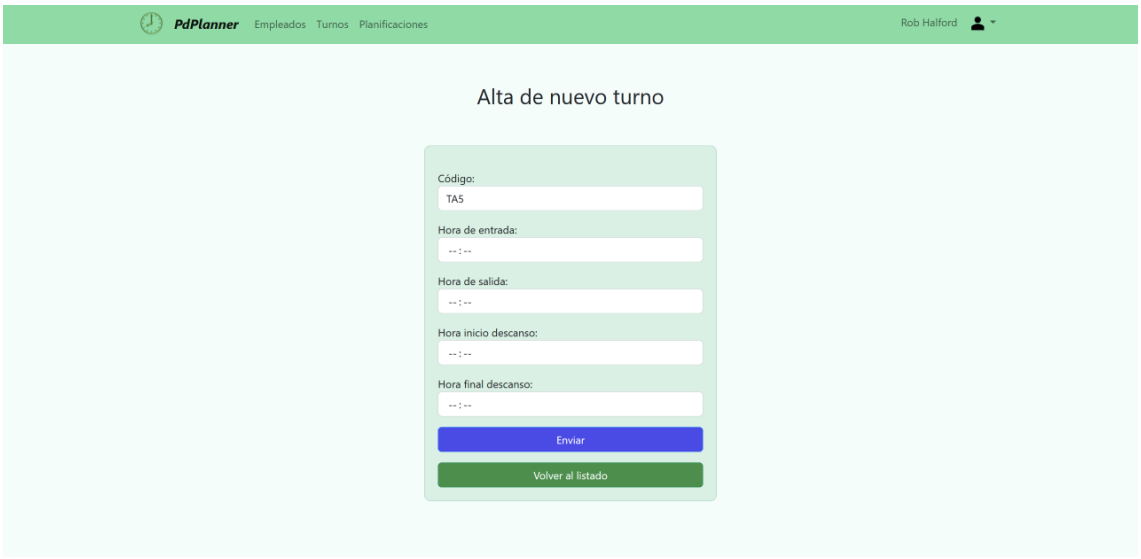


Figura 64. La aplicación en funcionamiento. Pantalla de detalles de planificación 1.

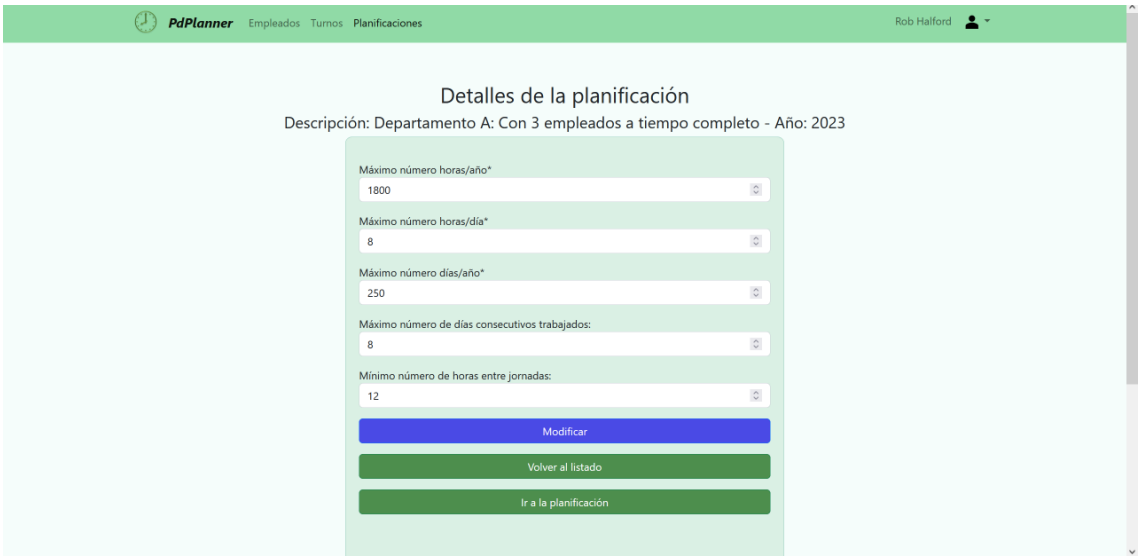


Figura 65. La aplicación en funcionamiento. Pantalla de detalles de planificación 2.

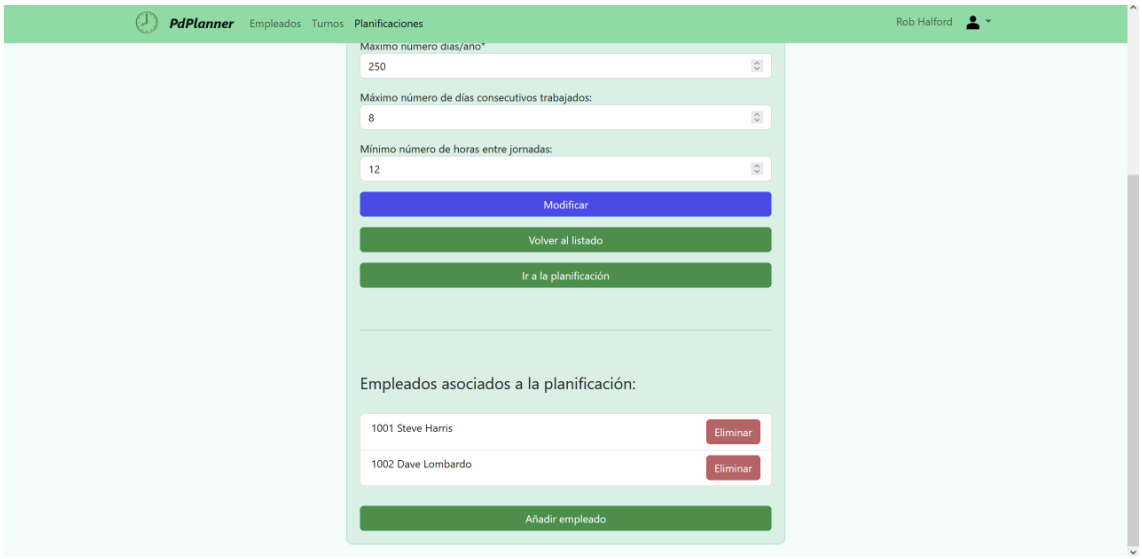


Figura 66. La aplicación en funcionamiento. Pantalla de horarios con muestra de coberturas.

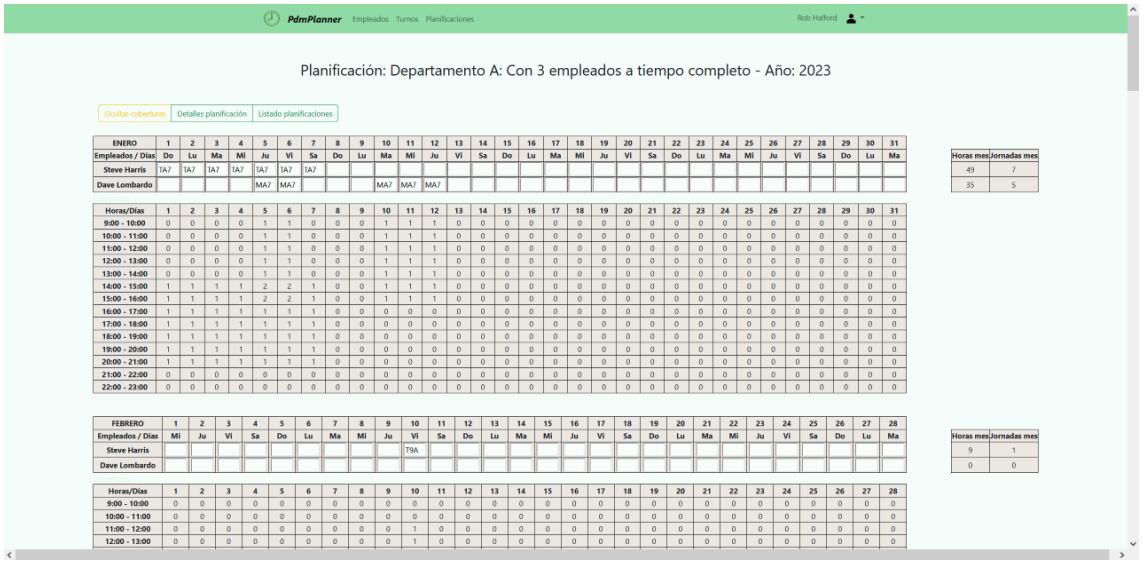


Figura 67. La aplicación en funcionamiento. Aviso de incumplimiento de restricción de planificación.

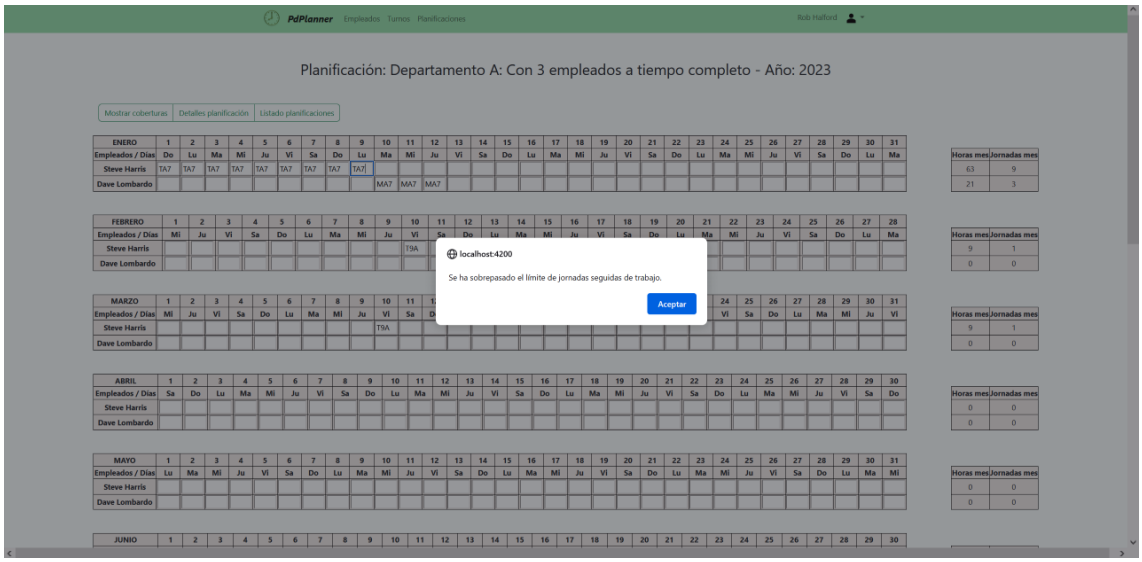
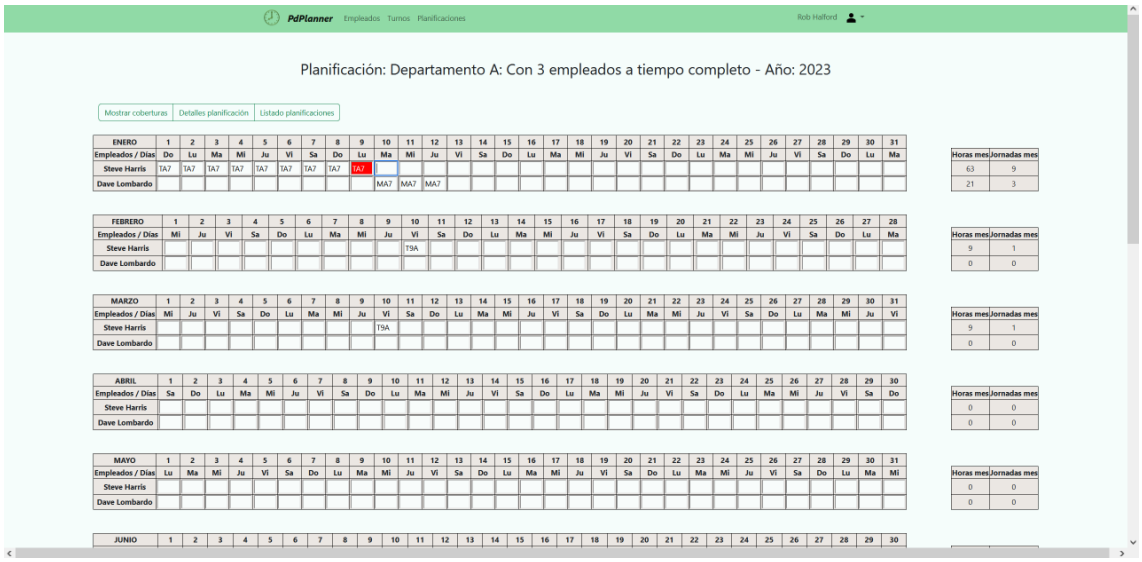


Figura 68. La aplicación en funcionamiento. Aviso de incumplimiento de restricción de planificación 2.



3.4 Deuda técnica

En este apartado se trata de poner de manifiesto la deuda técnica acumulada, sus causas y una posible estrategia para su gestión.

3.4.1 Identificación de la deuda técnica.

En el momento de la entrega de la primera versión del producto se ha constatado, principalmente, la siguiente deuda técnica:

- Los test no cubren la totalidad del producto, existe riesgo de problemas persistentes no localizados que pueden poner en peligro el desarrollo futuro.

- No existe automatización para la ejecución de los test, con la consiguiente pérdida de tiempo en su ejecución manual.
- No se ha desarrollado una estrategia de integración y despliegue continuos (CI/CD) automatizada.
- Pese a la sustancial mejora experimentada hasta el momento presente, el desarrollador cuenta con poca experiencia. Este hecho puede hacer que la deuda desconocida suponga un problema de difícil gestión.

3.4.2 Causas de la deuda técnica

Las principales causas de la deuda técnica descrita son la propia inexperiencia del autor (*naive technical debt*) y las decisiones conscientes tomadas con el objetivo de alcanzar un producto con una funcionalidad mínima que pudiese permitir su uso antes de la fecha límite prevista para su entrega (*strategic technical debt*).

3.4.3 Gestión de la deuda técnica

Tal y como explica Kenneth S. Rubin [45], la deuda técnica debe gestionarse adecuadamente. Para ello, el autor describe varios pasos que deben ser tenidos en cuenta:

1. **Prestar atención a la acumulación de deuda técnica**. Para ello, se propone lo siguiente:
    a. Usar buenas prácticas:
        i. Diseños simples
        ii. Desarrollo dirigido por pruebas (TDD)
        iii. Integración continua.
        iv. Automatizar las pruebas etc.
    b. Usar una buena descripción de requisitos que permita saber cuándo se ha terminado realmente la implementación de una funcionalidad.
    c. Entender y hacer entender la dimensión económica de arrastrar deuda técnica.
2. **Hacer visible la deuda técnica**:
    a. A nivel de negocio, con las implicaciones que tiene y los riesgos que puede acarrear.
    b. A nivel técnico, describiendo la deuda y tratándola como otra parte más que debe ser tenida en cuenta para el desarrollo del producto.
3. **“Pagar” la deuda técnica**. Teniendo en cuenta que no tiene por qué ser pagada de una sola vez y que no siempre tiene por qué ser pagada. Se deberá analizar cuándo y cómo es más conveniente invertir un determinado esfuerzo en solventar la deuda en función de muchas otras cuestiones relativas al desarrollo del producto.

3.4.4 Estrategia de reducción de la deuda técnica

Teniendo en cuenta lo comentado en los anteriores apartados se plantea la posibilidad de introducir, como punto de partida, un plan de reducción de deuda que se puede resumir en los siguientes pasos:

1. Crear un inventario de la deuda técnica.
2. Clasificar y agrupar la deuda, en función de su complejidad, coste e impacto, tanto en el caso de implementar una solución como en el de no hacerlo.
3. Tasar la deuda. Se debe conocer la magnitud del problema.
4. Publicar la deuda. Por ejemplo, en un backlog, para que sea visible y tenida en cuenta.
5. Reservar un determinado porcentaje de tiempo para reparar la deuda y así controlar su acumulación y el riesgo que supone. Es importante remarcar que la deuda técnica no tiene por qué pagarse siempre y que el hecho de hacerlo ha de responder a las necesidades globales del proyecto y no simplemente a cuestiones puramente técnicas.

4. Conclusiones

4.1 Lecciones aprendidas:

Como resumen de este apartado, se podría decir que la principal lección aprendida es el peso de la correcta planificación y diseño en el resultado final de un proyecto. En cualquier caso, a continuación se hace un análisis más detallado de todo lo aprendido durante este trabajo de fin de grado:

- **Importancia del conocimiento del dominio**. Este ha sido, sin duda, un aspecto que ha sido de gran ayuda en el desarrollo del proyecto, dada la amplia experiencia del autor como gestor de equipos de trabajo y elaborando horarios. Sin embargo, solamente en el momento de tratar de explicar determinadas cuestiones relacionadas con el diseño de la base de datos se ha tenido verdadera consciencia de la importancia de este aspecto. Conocer el ámbito en el que se va a utilizar el producto y las implicaciones directas que tendrá en el trabajo de los diferentes *stakeholders* supone una gran ventaja, que puede llegar a ser decisiva en el éxito del proyecto. En aquellos casos en los que un equipo se deba enfrentar a la creación de una herramienta sin el suficiente conocimiento del dominio del problema será necesario volcarse en las etapas de recogida de requisitos con el fin de aprender lo suficiente como para poder entenderse con todas las partes interesadas y asegurarse de que se alcanza una visión común de la solución que se busca. En cualquier caso, tener un alto conocimiento del dominio no implica obviar la creación de prototipos o elegir una metodología iterativa con el fin de asegurar la correcta comunicación entre todas las partes.
- **Importancia de la recogida y definición de requisitos**. Esta etapa es fundamental en el éxito de cualquier proyecto. No sirve de nada desarrollar una aplicación magnífica que nadie use porque no es lo que necesitaba para hacer su trabajo. Se ha de tener en cuenta que los requisitos no siempre se conocen en un primer momento y que, además, pueden variar con el paso del tiempo. En definitiva, los requisitos son algo vivo, que no se puede perder de vista en ningún momento y que deben estar sometidos a constante revisión. Como dato anecdótico, el autor del proyecto, con 20 años de experiencia haciendo horarios y siendo la única persona que ha decidido sobre los requisitos del presente trabajo, ha variado su opinión con respecto a varias cuestiones relacionadas con la forma en que determinadas funcionalidades deberían implementarse. Otra cuestión destacada con relación a los requisitos es su definición de “hecho”. Es decir, en qué momento se pude dar por terminada la implementación de una funcionalidad. Por ejemplo, una mejorable definición de “hecho” para el requisito R-005, hubiese conllevado, probablemente, una mejor implementación del sistema de avisos de incumplimiento de restricciones.
- **Importancia de la gestión stakeholders**. En los siguientes apartados del presente capítulo se abordará con más detalle el cumplimiento de los

objetivos iniciales y las posibles líneas de trabajo futuro. Como adelanto, se puede decir que, probablemente, serán los managers el colectivo más satisfecho con la primera versión de la aplicación. La empresa, y en especial el departamento de recursos humanos, posiblemente estén decepcionados ya que la aplicación no cuenta con un informe de cumplimiento de restricciones de las planificaciones, cuestión esta fundamental para los colectivos antes indicados. En el desarrollo de un proyecto siempre se deben tomar decisiones que no sean del agrado de todos, pero la forma en que se ejecuten puede ser decisiva a la hora de no perder la confianza en el equipo que desarrolla el proyecto. La comunicación debe ser veraz y continua, a través de los canales más adecuados y que previamente se han debido acordar.

- **Importancia de la planificación temporal**. El alcance global y de cada etapa, cuál será el producto mínimo viable etc. son también decisiones fundamentales para alcanzar el éxito en cualquier proyecto. La comunicación del equipo encargado del proyecto con todos los *stakeholders* tiene que permitir llegar a un consenso sobre todos estos asuntos. Y, ya dentro del equipo de desarrollo, éste debe contar con la suficiente experiencia como para saber dimensionar correctamente la magnitud de la empresa y elegir la metodología (ágil, cascada etc.) más adecuada en cada ocasión. Sin embargo, se puede alcanzar un producto con una funcionalidad suficiente sin necesidad de una gran experiencia, como ha ocurrido en el presente trabajo de fin de grado, donde gracias a las decisiones tomadas y la metodología elegida, se ha conseguido mitigar el riesgo inicial.
- **Importancia de la elección de tecnologías**. Este aspecto también ha sido especialmente importante en el desarrollo del proyecto. La elección de *frameworks* muy extendidos y con una gran comunidad de desarrolladores ha permitido acceso a abundante documentación y tutoriales que han facilitado las tareas de aprendizaje y desarrollo. Sin embargo, la decisión de no utilizar herramientas como *Keycloak* [54] o el soporte nativo para *OAuth2* [55] que ofrece Spring Boot, por motivos puramente educativos, ha complicado la implementación de la seguridad de la aplicación y restado funcionalidad, lo cual también ha afectado al alcance del proyecto y la ausencia de despliegue en Azure. Existen otras muchas cuestiones relativas a la elección de herramientas y tecnologías que deben ser tomadas en cuenta: como si son *open-source*, la fiabilidad de la comunidad o empresa que se encarga de su desarrollo, el entorno en que se vaya a implantar el producto final, por poner algunos ejemplos.
- **Importancia del diseño de la base de datos**. Debe representar el dominio del problema pero también debe ser manejable y adaptarse no solo a requerimientos técnicos sino también a la realidad del entorno. No es lo mismo diseñar una base de datos para el sector de la banca que para un juego de “la liga fantástica”. En el asistente para la planificación de horarios se ha optado, por ejemplo, por relajar las restricciones de la base de datos a fin de simplificarla y trasladar dichas restricciones a la lógica de la aplicación.
- **Importancia de la elección de la arquitectura del producto**. Esto marcará las características de la aplicación que finalmente se desarrolle. ¿Se necesitará un gran rendimiento? , ¿La escalabilidad será un factor

determinante? , ¿Hasta qué punto el coste limitará el desarrollo? , ¿Se debe prestar especial atención a la elasticidad? Todas estas cuestiones, y algunas más, deben tenerse en cuenta a la hora de elegir una determinada arquitectura. En este proyecto se ha tenido en cuenta que el producto final podría formar parte de un sistema más complejo en el que se ofreciese a la empresa un conjunto de funcionalidades dirigidas a mejorar todos los procesos internos. Se ha tratado de preparar el *backend* de la aplicación para poder adaptarlo con mínimos cambios a la arquitectura de un microservicio que formase parte de una red mayor. Por otra parte, se ha decidido utilizar una API REST y una SPA con la intención de volcar el grueso de la funcionalidad hacia la parte del cliente y limitar la transferencia de información a través de las redes de comunicación. Dado que cualquier equipo actual, por sencillo que sea, tiene suficiente potencia para ejecutar la aplicación, y que las redes empresariales no siempre son rápidas y fiables, se ha pensado que esta solución podría encajar bien en casi cualquier situación.

- **Importancia de la correcta elección de nombres para la legibilidad del proyecto y la documentación**. El código debe ser legible. Lo ideal sería que no fuese necesaria ninguna documentación. El uso de nombres adecuados para clases, variables, métodos etc. facilita la comprensión y el mantenimiento del código. Incluso el único autor del proyecto ha tenido que pararse durante un tiempo para averiguar cómo había implementado algún método días atrás.
- **Importancia de la elección de las estructuras de datos.** Deben ser manejables, sencillas y las operaciones necesarias deben tener los costes más bajos posibles. Si bien en un proyecto de pequeñas dimensiones no tienen una gran relevancia, siempre es correcto pensar en cómo se comportaría la aplicación ante un incremento de los datos a manejar. En el asistente para la planificación de horarios el ejemplo más claro de elección de estructura de datos es la clase *PlanningUserTable*, ya comentada en el apartado 3.2.3.
- **Importancia del uso de patrones de diseño**. Para lograr los objetivos de reusabilidad, mantenimiento, *extensibilidad* etc. no se debe perder de vista el hecho de que hay una serie de patrones de efectividad comprobada, que ayudan a lograr dichos objetivos. El hecho de utilizar *frameworks* como Angular y Spring tiene la ventaja de que estos ya implementan gran cantidad de patrones de diseño que ayudan a estructurar el código. Así, por ejemplo, ambos facilitan el desacoplamiento del código mediante sus propias implementaciones de *Inyección de dependencias*. Otro caso destacado es el uso de los objetos *BehaviorSubject* de la biblioteca RxJS, que utilizan el patrón *Observer*. Fuera del ámbito de los *frameworks*, la implementación que se hace de la arquitectura hexagonal utilizando el patrón *Adapter* para los adaptadores de entrada y salida ya explicados en el apartado dedicado a la arquitectura del backend sería un ejemplo más de utilización de patrones de diseño.
- **La importancia del trabajo en equipo.** Precisamente por no haber un equipo en el que apoyarse y con el que contrastar ideas, la realización del proyecto ha sido más compleja. En un equipo puede haber diferentes roles, *backgrounds*, conocimientos y puntos de vista que se puede poner

a disposición de los demás, facilitando de esa forma el desarrollo del proyecto.

- **Importancia de las pruebas y CI/CD**. En ambos casos se facilita la creación de código libre de fallos y se garantiza en mayor medida la no aparición de problemas en fases posteriores del desarrollo. En este proyecto se ha decidido, no obstante, posponer la construcción de pruebas a la creación de la funcionalidad registrada en los requisitos ante la incertidumbre en la consecución de este último objetivo. Esto ha hecho que el desarrollo del producto haya estado más expuesto a la aparición de errores en momentos en los que su reparación hubiese sido más costosa o, peor aún, la entrega del producto con errores de los que no se es consciente por no haberse hecho las suficientes pruebas.

4.2 Objetivos no cumplidos:

Seguidamente, se detallan los objetivos no cumplidos en el proyecto. En algunos casos, no se apreciará un reflejo de las carencias con la simple lectura de los requisitos y su comparación con la funcionalidad de la aplicación entregada. Sin embargo, se han incluido ya que se considera que la forma en la que se han implantado no cumple con el objetivo de la mejor forma posible. Algunos de los siguientes puntos se han incluido también en el apartado relativo a las líneas futuras de trabajo:

- Las pruebas no cubren la totalidad del código y la funcionalidad de la aplicación. Se ha optado por mostrar diferentes tipos de test en pequeñas partes de código o algunas funcionalidades.
- El manejo de excepciones no se extiende a la totalidad de las situaciones tanto en el *frontend* como en el *backend.*
- La señalización de los incumplimientos de las restricciones de las planificaciones no permite diferenciar su causa una vez que se han mostrado los mensajes. Queda pendiente implementar un sistema que permita su consulta en cualquier momento.
- Algunos avisos de incumplimiento de restricciones no se eliminan de forma adecuada cuando se corrige el turno que los causa. Por ejemplo, cuando se lanza un aviso por incumplimiento del número máximo de jornadas continuadas de trabajo y se elimina un turno intermedio en la secuencia, los siguientes turnos, que ya no están afectados por incumplimiento de esta restricción, siguen manteniendo el color rojo.
- Despliegue Azure.

La causa fundamental de no haber cumplido con el alcance planificado para el proyecto ha sido la inexperiencia del autor. La adquisición de nuevos conocimientos ha hecho que, con relativa frecuencia, se haya tomado la decisión de cambiar un determinado aspecto de la aplicación ya desarrollado, como, por ejemplo, el caso del manejo de errores en el *backend*, que en principio se trató de implementar en el controlador y, posteriormente, se externalizó a la clase *PdPlannerExceptionHandler*. En otras ocasiones, la limitación temporal ha impedido deshacer el trabajo realizado para mejorar el resultado final. Sin ir más lejos, en referencia también al manejo de excepciones, la utilización de

interceptores en Angular daría un resultado más sencillo y limpio. También se ha sacrificado funcionalidad y tiempo de desarrollo por abordar aspectos relativos al aprendizaje de determinadas cuestiones, como en el caso ya comentado de la implementación de autenticación y autorización sin la utilización de herramientas como *Keycloak* o *OAuth2*.

Todos los retrasos experimentados han hecho que se haya extraído el desarrollo de test a una etapa posterior a la marcada en la planificación temporal original, con el fin de poder avanzar lo más rápidamente posible con el desarrollo de la funcionalidad.

4.3 Seguimiento

A lo largo del desarrollo del proyecto se ha hecho un seguimiento de su evolución teniendo en cuenta, sobre todo, los hitos marcados en el calendario.

Así, se han hecho diversas variaciones sobre la planificación temporal original con la intención de asegurar el éxito del proyecto.

La mayor parte de los cambios han sido relativos al calendario establecido inicialmente y el alcance del proyecto. Esta situación ya había sido prevista en la evaluación de riesgos, y por ello se había tomado la decisión de utilizar una metodología iterativa, de forma que cualquier desviación en la planificación original pudiese ser detectada en el menor tiempo posible. Gracias a eso, se ha conseguido alcanzar un producto que cumple los requerimientos mínimos aunque es cierto que hay determinadas características que no han sido completamente desarrolladas, como el sistema de avisos de incumplimiento de restricciones.

Por otra parte, también ha habido variaciones respecto a los requisitos planteados inicialmente. Por ejemplo, la creación, modificación y eliminación de usuarios del sistema se ha limitado a los administradores. Tal y como se había planteado la aplicación en un primer momento, existía la posibilidad de que varios managers asignasen horarios a los mismos empleados en distintas planificaciones. En estas circunstancias, borrar un empleado acarrea riesgo de pérdida de datos. Además, la creación y modificación de empleados en un entorno empresarial no debería dejarse abierta por los problemas que esto puede suponer. En esta misma línea, el borrado de planificaciones también suponía un riesgo de pérdida de datos que sería peligroso. Finalmente, con el ánimo de mostrar diferentes funcionalidades o enfoques para solucionar el mismo tipo de problema, se ha optado por permitir el borrado en este caso, pero mostrando previamente un aviso de las consecuencias que este hecho puede conllevar.

4.4 Líneas de trabajo futuro.

A continuación se detallan una serie de mejoras y correcciones que pueden ser introducidas en la aplicación y la metodología empleada para su desarrollo:

- Establecer un plan para gestionar la deuda técnica, sobre todo:
    - Establecer un sistema de automatización que permita la integración y despliegue continuos (CI/CD).
    - Desarrollar y automatizar test suficientes que sirvan para dirigir el desarrollo del producto.
- Limitar los permisos de los managers de tal forma que solo puedan acceder a información relativa al personal que tienen asignado y las planificaciones y turnos que ellos mismos crean. De esta forma se limita y se protege el acceso a los datos.
- Limitar el acceso de los empleados de tal forma que solo puedan ver las planificaciones de las que formen parte.
- Introducir un resumen de planificación con las jornadas anuales por empleado, horas totales, porcentajes de días trabajados en fin de semana entre otras posibilidades.
- Mejorar la visibilidad de las tablas de horarios permitiendo marcar festivos, domingos y otros días especiales con códigos de color.
- Cambiar la forma en la que se presentan las violaciones de restricciones. No se debería usar el método *alert* de JavaScript ya que puede ser deshabilitado por el usuario.
- Introducir un sistema para colorear las celdas de las tablas de coberturas en función del número de trabajadores. Esto permitiría detectar a primera vista desviaciones en los objetivos marcados.
- Crear un informe de incumplimiento de restricciones de las planificaciones.
- Crear un sistema de avisos de incumplimiento de restricciones de las planificaciones cuando estas se cargan desde la base de datos.
- Añadir nuevas restricciones de planificación relativas a las libranzas y vacaciones de los empleados. Por ejemplo, si es el caso, número de fines de semana libres al año, vacaciones disfrutadas y otros tipos contemplados en la normativa.
- Crear un sistema de fichaje, podría ser un microservicio, y vincularlo con el planificador de horarios. Se podrían desarrollar de esta forma nuevas funcionalidades relativas al registro de retrasos, ausencias, bajas laborales, permisos etc.
- Desarrollo de una IA que permita elaborar planificaciones a partir de los datos de los empleados disponibles, turnos posibles, restricciones, calendario y cargas de trabajo (por ejemplo los presupuestos) por horas a lo largo del año.

4. Glosario

API (*Application programming interface*): es un conjunto de reglas predefinidas que permiten la comunicación entre sistemas. Funcionan como una capa intermedia que procesa la transferencia de datos [27].

API REST (*Representational State Transfer*): es una Api que sigue los principios de diseño de arquitectura REST [28]:

1. Interfaz uniforme: Todas las peticiones a un mismo recurso deben tener la misma URI (*Uniform resource identifier*).
2. El cliente y el servidor deben estar desacoplados.
3. Las peticiones a la API REST no deben tener estado. Así, cada una de ellas debe contener la información suficiente para ser procesada.
4. Cuando sea posible, los datos deben poder ser *cacheados* en el cliente o en el servidor.
5. Deben estar diseñadas de tal forma que tanto el cliente como el servidor puedan comunicarse a través de un número indefinido de intermediarios.

Backend: Es la parte del servidor de una aplicación que sigue la arquitectura cliente-servidor.

DTO (Data Transfer Object) [43]: Estructuras que encapsulan información que será transmitida entre procesos o redes. La finalidad de su uso es reducir el coste de las comunicaciones sobre una red.

Endpoint (API): Son las URL donde se reciben las peticiones hechas a la API. Cada una de esas URL está vinculada a una determinada funcionalidad que ofrece la API.

Framework: En desarrollo de software, es una base que sirve como estructura sobre la que construir software mediante la implementación de soluciones a problemas comunes.

Frontend: Es la parte del cliente de una aplicación que sigue la arquitectura cliente-servidor.

MV*/MVC/MVVM (Model-View-* / Model-View-Controller / Model-View-ViewModel): Son patrones de arquitectura que persiguen la separación de diferentes capas de una aplicación con el fin de mejorar, principalmente, el mantenimiento y la reusabilidad.

Open Source [43]: Se dice que un software es open source cuando su creador cede los derechos de uso, cambio y distribución del código fuente a cualquier usuario y para cualquier propósito.

OpenJDK (Open Java Development Kit) [45]: Es una implementación gratuita y open source de la plataforma Java.

Single Page Application: es una aplicación web que carga todo el código necesario para generar las vistas en el navegador una sola vez. Posteriormente, el contenido (datos) se actualiza mediante la utilización de librerías JavaScript como XMLHttpRequest [29].

Stakeholders: Son todas las partes afectadas en el desarrollo de un proyecto.

Two-way data binding: Es la forma de comunicación entre la plantilla y el controlador de un componente Angular de manera que cualquier cambio en uno de ellos se refleja en el otro.

5. Bibliografía

[1] *Angular* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://angular.io/

[2] *Node.js* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://nodejs.org/en/

[3] *Npm* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://www.npmjs.com/

[4] *Spring* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://spring.io/

[5] *Adoptium* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://adoptium.net/es/

[6] *Visual Studio Code* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://code.visualstudio.com/

[7] *Jetbrains IngelliJ Idea* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://www.jetbrains.com/idea/

[8] *Apache Maven* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://maven.apache.org/index.html

[9] *PostgreSQL* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://www.postgresql.org/

[10] *Docker* [en línea] [06 de marzo de 2023]. Disponible en: https://www.docker.com/

[11] *Microsoft Azure* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://azure.microsoft.com/es-es/

[12] *Git* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://git-scm.com/

[13] *GitHub* [en línea] [consulta: 06 de marzo de 2023]. Disponible en: https://github.com/

[14] *GitLab* [en línea] [06 de marzo de 2023]. Disponible en: https://gitlab.com/

[15] *Postman* [en línea] [06 de marzo de 2023]. Disponible en: https://www.postman.com/

[16] RODRÍGUEZ CARRO, Eduardo. *Plataforma web de e-learning para colegios* [en línea]. Trabajo final de grado: UOC, 2021 [consulta: 06 de marzo de 2023]. Disponible en: https://openaccess.uoc.edu/bitstream/10609/126507/11/erodriguezcarroTFG0121memoria.pdf

[17] COSME DE LA ROSA, Webster. *Adaptador de foros web* [en línea]. Trabajo final de grado: UOC, 2021 [consulta: 06 de marzo de 2023]. Disponible en: https://openaccess.uoc.edu/bitstream/10609/126566/7/wcosmedlrTFG0121memoria.pdf

[18] RAMÓN RODRÍGUEZ, José y MARINÉ JOVÉ, Pere. *Gestión de proyectos: Iniciación del proyecto y trabajos previos* [en línea]. Barcelona: UOC [consulta: 06 de marzo de 2023]. Disponible en: https://materials.campus.uoc.edu/daisy/Materials/PID_00247938/pdf/PID_00247938.pdf

[19] RAMÓN RODRÍGUEZ, José y MARINÉ JOVÉ, Pere. *Gestión de proyectos: Planificación del proyecto* [en línea]. Barcelona: UOC

[consulta: 06 de marzo de 2023]. Disponible en: https://materials.campus.uoc.edu/daisy/Materials/PID_00247943/pdf/PID_00247943.pdf

[20] BURGUÉS ILLA, Xavier y CUARTERO OLIVERA, Josep. *Diseño de bases de datos: Diseño lógico de bases de datos* [en línea]. Barcelona: FUOC, 2020 [consulta: 29 de marzo de 2023]. Disponible en: https://materials.campus.uoc.edu/daisy/Materials/PID_00270596/pdf/PID_00270596.pdf

[21] PRADEL MIQUEL, Jordi y RAYA MARTOS José. *Ingeniería de requisitos. Documentación de requisitos* [en línea]. Barcelona: FUOC [consulta 31 de marzo de 2023]. Disponible en: https://materials.campus.uoc.edu/daisy/Materials/PID_00191257/pdf/PID_00191264.pdf

[22] ULUCA, Doguhan. *Angular for enterprise-ready web applications*. Birmingham: Packt, 2020. ISBN 9781838648800.

[23] RICHARDSON, Chris. *Microservices Patterns*. New York: Manning, 2018. ISBN 9781617294549.

[24] SESHADRI, Shyam. *Angular: Up and Running*. Sebastopol: O'Reilly, 2018. ISBN 9781491999837.

[25] SPILCA, Laurentiu. *Spring Start Here*. Shelter Island: Manning, 2021. ISBN 9781617298691

[26] COHN, Mike. *User stories applied for Agile software development*. Boston: Addison-Wesley, 2004. ISBN 0321205685

[27] *IBM. Topics: API* [en línea] [consulta: 12 de marzo de 2023]. Disponible en: https://www.ibm.com/topics/api

[28] *IBM. Topics: REST-API* [en línea] [consulta: 12 de marzo de 2023]. Disponible en: https://www.ibm.com/topics/rest-apis

[29] *Developer Mozilla. Glossary. SPA* [en línea] [consulta: 12 de marzo de 2023]. Disponible en: https://developer.mozilla.org/en-US/docs/Glossary/SPA

[30] *Spring. Spring Data JPA – Reference Documentation* [en línea] [consulta: 15 de mayo de 2023]. Disponible en: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[31] *Wikipedia. JSON.* [en línea] [consulta 2 de mayo de 2023]. Disponible en: https://es.wikipedia.org/wiki/JSON

[32] *Jakarta EE Platform.* [en línea] [consulta: 20 de mayo de 2023]. Disponible en: https://jakarta.ee/specifications/platform/10/jakarta-platform-spec-10.0.html#jakarta-persistence-api

[33] *Hibernate.* [en línea] [consulta el 15 de mayo de 2023]. Disponible en: https://hibernate.org/

[34] *Spring. Spring Security.* [en línea] [consulta: 15 de mayo de 2023]. Disponible en: https://docs.spring.io/spring-security/reference/index.html

[35] *JWT.* [en línea] [consulta el 16 de mayo de 2023]. Disponible en: https://jwt.io/

[36] *Wikipedia. JSON Web Token.* [en línea] [consulta el 16 de mayo de 2023]. Disponible en: https://es.wikipedia.org/wiki/JSON_Web_Token

[37] *Amigoscode (YouTube). Spring Boot 3 + Spring Security 6 + JWT.* [en línea] [consulta el 10 de mayo de 2023]. Disponible en: https://www.youtube.com/watch?v=KxqlJblhzfI

[38] *Jwtk.jjwt.* [en línea] [consulta el 3 de mayo de 2023]. Disponible en: https://github.com/jwtk/jjwt
[39] *Bouali Ali (YouTube). Spring Boot 3 & Spring Security 6 – Roles and permissions.* [en línea] [consulta el 15 de mayo de 2023]. Disponible en: https://www.youtube.com/watch?v=mq5oUXcAXL4
[40] *RxJS.* [en línea] [consulta el 10 de mayo de 2023]. Disponible en: https://rxjs.dev/
[41] Schwarzmuller, Maximillian. Angular, the complete guide. En O'Reilly Media. [en línea] [consulta: 20 de abril de 2023]. Disponible en: https://learning.oreilly.com/videos/angular-the/9781788998437/
[42] *Wikipedia. Data Transfer Object.* [en línea] [consulta: 18 de junio de 2023]. Disponible en: https://en.wikipedia.org/wiki/Data_transfer_object
[43] *Wikipedia. Open Source.* [en línea] [consulta: 18 de junio de 2023]. Disponible en: https://en.wikipedia.org/wiki/Open-source_software
[44] *Wikipedia. OpenJDK.* [en línea] [consulta: 18 de junio de 2023]. Disponible en: *https://en.wikipedia.org/wiki/OpenJDK*
[45] Rubin, Kenneth S. *Essential Scrum: A practical guide to the most popular Agile process*. New Jersey: Pearson Education, 2012. ISBN 9780137043293
[46] *TypeScript*. [en línea] [consulta el 10 de marzo de 2023]. Disponible en: https://www.typescriptlang.org/
[47] *Spring. Spring Web MVC.* [en línea] [consulta: 15 de mayo de 2023]. Disponible en: https://docs.spring.io/spring-framework/reference/web/webmvc.html
[48] PRADEL i MIQUEL, Jordi y RAYA MARTOS, José Antonio. *Análisis y diseño con patrones. Catálogo de patrones.* Barcelona: FUOC, 2020.
[49] *Jasmine*. [en línea] [consulta el 23 de junio de 2023]. Disponible en: https://jasmine.github.io/
[50] *Junit.* [en línea] [consulta el 5 de mayo de 2023]. Disponible en: https://junit.org/junit5/
[51] *AssertJ*. [en línea] [consulta el 5 de mayo de 2023]. Disponible en: https://assertj.github.io/doc/
[52] *Mockito.* [en línea] [consulta el 5 de mayo de 2023]. Disponible en: https://site.mockito.org/
[53] *SonarLint* [en línea] [consulta el 10 de junio de 2023]. Disponible en: https://www.sonarsource.com/products/sonarlint/
[54] *Keycloak.* [en línea] [consulta el 24 de junio de 2023]. Disponible en: https://www.keycloak.org/
[55] *OAuth2.* [en línea] [consulta el 24 de junio de 2023]. Disponible en: https://oauth.net/getting-started/