

---

## Proyecto de desarrollo de aplicaciones multiplataforma

---

# DESARROLLO DE UNA APLICACIÓN PARA LA GESTIÓN Y REALIZACIÓN DE RESERVAS DE LOS ESPACIOS COMUNES DEL CENTRO ESCOLAR

CICLO FORMATIVO DE GRADO SUPERIOR  
**Desarrollo de Aplicaciones Multiplataforma (IFCS02)**

**Curso 2022-23**



Autor/a/es:

**Mireya Sánchez Pinzón**  
**Alejandro Sánchez Monzón**  
**Rubén García-Redondo Marín**

Tutor/a:

**JAVIER PALACIOS GONZÁLEZ**

Departamento de Informática y Comunicaciones  
**I.E.S. Luis Vives**

## **ABSTRACT**

---

*In response to the needs expressed by the school and in order to improve its experience with the organisation of common spaces, our group considered it useful to develop a digital resource to facilitate the management of common spaces. Gradually, the school is increasingly able to maintain different types of spaces, which improve the students' experience, and therefore their academic development.*

## **RESUMEN**

---

*En respuesta a las necesidades expresadas por el centro y para mejorar su experiencia con la organización de los espacios comunes, nuestro grupo consideró útil desarrollar un recurso digital para facilitar su gestión. Poco a poco, la escuela es cada vez más capaz de mantener diferentes tipos de espacios, lo que mejora la experiencia de los estudiantes y, por tanto, su desarrollo académico.*

# ÍNDICE

ABSTRACT .....	2
RESUMEN.....	2
1 INTRODUCCIÓN .....	5
1.1 OBJETIVO.....	5
1.2 ALCANCE.....	5
1.3 JUSTIFICACIÓN .....	6
2 IMPLEMENTACIÓN.....	6
2.1 ANÁLISIS DE LA APLICACIÓN .....	6
2.1.1 REQUISITOS FUNCIONALES: .....	6
2.1.2 PLATAFORMAS (RF) .....	7
2.1.3 INTERFAZ DE USUARIO (RF) .....	8
2.1.4 ESCALABILIDAD (RF).....	9
2.1.5 BENEFICIOS (RF).....	10
2.1.6 REQUISITOS NO FUNCIONALES: .....	10
2.1.7 PLATAFORMAS (RNF).....	11
2.1.8 NUESTRA PROGRAMACIÓN BASE (RNF) .....	12
2.1.9 BASES DE DATOS (RNF).....	14
2.1.10 DESPLIEGUE (RNF) .....	15
2.1.11 ESCALABILIDAD (RNF).....	16
2.1.12 SEGURIDAD (RNF).....	16
2.1.13 TESTING (RNF) .....	17
2.1.14 INYECCIÓN DE DEPENDENCIAS (RNF).....	18
2.1.15 RETROFIT Y KTORFIT .....	19
2.1.16 CACHÉ.....	20
2.1.17 CLIENTE DE PRUEBAS DE RUTAS: POSTMAN.....	21
2.1.18 CREACIÓN DE PROTOTIPOS Y MOCKUPS.....	22
Ejemplo de navegabilidad de la aplicación en el prototipo gracias a Figma: .....	22
2.2 DISEÑO .....	22
2.2.1 PROTOTIPO .....	22
2.2.2 DIAGRAMA DE CLASES.....	26
2.2.3 DIAGRAMA ENTIDAD-RELACIÓN .....	29
2.2.4 DIAGRAMA CASOS DE USO .....	31
2.2.5 ARQUITECTURA DEL PROYECTO .....	32
2.3 IMPLEMENTACIÓN .....	39
2.3.1 API RESTFUL.....	40
2.3.2 REPOSITORIOS .....	41
2.3.3 SERVICIOS .....	42

2.3.4	CONTROLADORES .....	45
2.3.5	PATRONES DE DISEÑO .....	46
2.3.6	SEGURIDAD EN EL SERVIDOR.....	50
2.3.7	FASE DE TESTING Y ANÁLISIS DE PRUEBAS.....	51
2.3.8	PROVIDERS.....	53
2.3.9	PANTALLAS .....	54
2.4	IMPLANTACIÓN, EMPAQUETADO Y DESPLIEGUE .....	55
2.5	DOCUMENTACIÓN .....	58
2.5.1	SWAGGER .....	58
2.5.2	KDOC CON DOKKA .....	60
2.5.3	README.....	60
3	RESULTADOS Y DISCUSIÓN .....	62
4	TRABAJO FUTURO .....	63
4.1	Segunda parte .....	63
4.2	Mejora y desarrollo continuo.....	64
4.3	¿Qué se puede mejorar?.....	64
5	CONCLUSIONES.....	64
6	BIBLIOGRAFÍA .....	66
7	ANEXOS.....	68
I.	ANEXO 1: LINK DIRECTO AL PROTOTIPO DE LA APLICACIÓN (GITHUB).....	68
II.	ANEXO 2: LINK DIRECTO AL ANTEPROYECTO (GITHUB).....	68

# 1 INTRODUCCIÓN

---

El objetivo del proyecto es desarrollar una aplicación para la gestión y realización de reservas de los espacios comunes del centro escolar, para mejorar la experiencia y organización del centro educativo.

La idea surge de tener una herramienta digital que permita optimizar la gestión de los recursos del centro y facilitar la realización de reservas de salas, aulas, pistas polideportivas, entre otros espacios comunes. Con la implementación de esta aplicación, se espera brindar una solución eficaz y eficiente para la gestión de los espacios, así como mejorar la experiencia de los usuarios del centro en cuanto a la organización y utilización de estos recursos.

El proyecto consistirá en el desarrollo de una aplicación móvil para la reserva de espacios, una aplicación para la gestión de los recursos (“BackOffice”) y un Backend para gestionar los datos y proporcionar el servicio a las aplicaciones cliente. Además, se implementará un sistema de acceso seguro a la plataforma y se dejan varios apartados, dentro de la aplicación, estructurados, para que los compañeros que continúen desarrollando la misma puedan hacerlo sin problemas. Entre estas funciones futuras encontramos: un sistema de notificaciones genérico que pueda ser enfocado tanto al alumno como al equipo de secretaría/jefatura, un sistema de reservas para diferentes servicios que otorgan diferentes ciclos educativos, etc.

Por otro lado, dada la diversidad de sistemas operativos existentes en el mercado de teléfonos móviles (Android y iOS). Se ha desarrollado la aplicación utilizando tecnologías multiplataforma para brindar nuestros servicios al mayor número de usuarios posibles.

En definitiva, es un proyecto ambicioso que pretende consolidar un lenguaje de programación multiplataforma basado en los conocimientos adquiridos en este curso, conocer el funcionamiento de una herramienta de gestión en un entorno real y ayudar al centro educativo en las necesidades que estos manifiestan, mejorando y agilizando su organización.

## 1.1 OBJETIVO

- ✓ Consolidar un lenguaje de programación multiplataforma en base a los conocimientos adquiridos este curso.
- ✓ Conocer el funcionamiento de una herramienta de gestión en un entorno real.
- ✓ Ayudar al centro educativo en las necesidades que estos manifiestan, mejorando y agilizando la organización de este.

## 1.2 ALCANCE

El proyecto consistirá en la realización de una aplicación para la gestión de reservas de espacios comunes del centro (salas, aulas, pistas polideportivas...). La aplicación permitirá gestionar los recursos (espacios), asignar perfiles y permisos a los usuarios y gestionar las reservas.

- ✓ Aplicación móvil para la reserva de espacios.
- ✓ Aplicación para la gestión de los recursos (back-office).
- ✓ Backend para gestionar los datos y proporcionar el servicio a las aplicaciones cliente.
- ✓ Gestionar el acceso seguro a la plataforma.
- ✓ Posibilidad de desarrollar la aplicación para un entorno multiplataforma (web o iOS).

---

## 1.3 JUSTIFICACIÓN

Ante la necesidad del centro educativo y para mejorar su experiencia en torno a la organización de los espacios comunes, nuestro grupo consideró útil desarrollar un recurso digital para facilitar dicha gestión.

# 2 IMPLEMENTACIÓN

---

## 2.1 ANÁLISIS DE LA APLICACIÓN

La aplicación propuesta en este proyecto es una herramienta digital para mejorar la experiencia y organización del centro educativo en la gestión de sus espacios comunes.

Por eso se ha hecho fundamental realizar una entrevista al cliente para obtener los requisitos que debe cumplir la aplicación. Lo que se obtuvo de la entrevista en cuestión fueron los requisitos funcionales de los que, a continuación, se detalla un análisis de sus principales aspectos:

Es primordial señalar que la aplicación será utilizada por diferentes perfiles de usuarios, entre ellos encontraremos alumnos, que simplemente tendrán la posibilidad de crear, modificar o anular las reservas que ellos mismos hagan de los espacios a los que estén autorizados, profesores, que tendrán la posibilidad de realizar las mismas acciones que los alumnos (aunque estarán autorizados a reservar más espacios que los alumnos), y por último el administrador, que tendrá la posibilidad de gestionar todas las reservas de los usuarios, verificar reservas, gestionar los espacios (editarlos, crearlos y eliminarlos) y crear, editar o eliminar usuarios:

### 2.1.1 REQUISITOS FUNCIONALES:

#### ✓ Reserva de espacios:

- **Alumnos:**

- Creación de reservas: Los alumnos tendrán la posibilidad de crear reservas de los espacios a los que estén autorizados, seleccionando la fecha, la hora y el espacio que desean reservar junto a un breve comentario, si lo precisan.
- Modificación de reservas: Los alumnos podrán modificar las reservas creadas, cambiando la fecha, hora o espacio reservado, si se hace antes de la fecha de inicio de la reserva, el tiempo para reservar el espacio es válido, y no haya otra reserva para ese espacio a esa hora, pudiendo cambiar los comentarios/observaciones de esta.
- Anulación de reservas: Los alumnos podrán anular las reservas creadas, siempre que se realice antes de la fecha de inicio de la reserva.

- **Profesores:**

- Creación de reservas: Los profesores tendrán las mismas posibilidades que los alumnos para crear reservas, pero estarán autorizados a reservar más espacios que ellos (todos los espacios que permitan la reserva de profesores y no de alumnos).
- Modificación de reservas: Los profesores tendrán las mismas posibilidades que los alumnos para modificar reservas.
- Anulación de reservas: Los profesores tendrán las mismas posibilidades que los alumnos para anular reservas.

- **Administrador:**
  - Modificación de reservas: El administrador podrá modificar o anular cualquier reserva creada por los usuarios (alumno o profesor), si se detecta un error o una inconsistencia en la reserva.
- ✓ **Gestión de espacios y reservas:**
  - **Administrador:**
    - Gestión de espacios: El administrador tendrá la posibilidad de gestionar todos los espacios del centro educativo, creando nuevos espacios, modificando los existentes y asignando características y limitaciones a cada uno de ellos.
    - Verificación de reservas: El administrador podrá verificar las reservas realizadas por los usuarios, asegurándose de que se cumplen las normas y limitaciones asignadas a cada espacio.
    - Modificación de reservas: El administrador podrá modificar o anular cualquier reserva creada por los usuarios si se detecta un error o una inconsistencia en la reserva.
    - Creación de reservas: Si es necesario, el administrador podrá crear nuevas reservas en nombre de los usuarios o a su propio nombre, si se solicita la reserva de manera explícita o por motivos justificados.
- ✓ **Asignación de perfiles:**
  - **Administrador:**
    - Creación de perfiles: El administrador tendrá la posibilidad de crear perfiles de usuario, asignando a cada perfil los permisos necesarios para acceder a las diferentes funcionalidades de la aplicación.
    - Modificación de perfiles: El administrador podrá modificar los perfiles de usuario existentes, asignando o retirando permisos a cada perfil según sea necesario.
    - Eliminación de perfiles: El administrador podrá eliminar perfiles de usuario si se han creado por error o ya no son necesarios.
    - Asignación de espacios y limitaciones: El administrador podrá asignar a los usuarios los espacios a los que tienen acceso y las limitaciones correspondientes, asegurándose de que se cumplan las normas y restricciones establecidas para cada espacio.
- ✓ **Los créditos como limitación de reservas y su implementación.**
  - Un usuario que tenga permisos de alumno o profesor tendrá la cantidad de 20 créditos mensuales para poder realizar sus reservas, de forma que, si ese usuario no tiene más créditos, no podrá realizar nuevas reservas hasta el siguiente mes. Cada espacio tiene un valor asignado (modificable por los administradores), lo que hace que sea fácilmente regulable su uso. Un administrador puede vetar a un usuario de créditos (privándole de su capacidad de reservar), o sancionar a los mismos restándoles la cantidad que poseen.

### **2.1.2 PLATAFORMAS (RF)**

La aplicación móvil se ha desarrollado teniendo en mente los perfiles de alumno y profesor, por lo que se deben considerar limitaciones y características específicas de las plataformas móviles.

En primer lugar, hay que considerar que las aplicaciones móviles deben ser fáciles de usar y diseñadas para la interacción táctil en pantallas pequeñas. Las interfaces de usuario deben ser simples y claras, y el flujo de navegación debe ser intuitivo y fácil de entender para los usuarios.

Además, las aplicaciones móviles deben tener un rendimiento rápido y fluido, ya que los usuarios esperan una experiencia de uso sin interrupciones en sus dispositivos móviles. Por tanto, el diseño de la aplicación debe optimizarse cuidadosamente para minimizar la carga en los recursos del dispositivo y mejorar el tiempo de respuesta.

En cuanto al perfil de administrador, se desarrollará un BackOffice ya que, a diferencia de las aplicaciones móviles, en las aplicaciones web se pueden aprovechar una pantalla más grande para mostrar más información en una sola vista. Además, nos aprovecharemos de que permiten un mayor control de la interfaz de usuario y la interacción a través del uso de teclados y ratones.

Pero las aplicaciones web también presentan desafíos en cuanto a compatibilidad con navegadores y sistemas operativos. Por lo tanto, se debe garantizar que la aplicación sea compatible con una amplia variedad de dispositivos y navegadores.

Además, la seguridad es una consideración importante al desarrollar una aplicación web. La plataforma debe ser segura para evitar la exposición de datos privados de los usuarios, la inyección de código malicioso y otros ataques.

En resumen, al desarrollar una aplicación móvil y una plataforma web para la gestión de reservas de espacios comunes, se deben considerar cuidadosamente las limitaciones y características específicas de cada plataforma para garantizar una experiencia de usuario de alta calidad y la seguridad de los datos.



### 2.1.3 INTERFAZ DE USUARIO (RF)

Para lograr una interfaz de usuario intuitiva y fácil de usar, es necesario considerar algunos aspectos clave en el diseño de la aplicación para la gestión de reservas de espacios comunes del centro.

A continuación, se describen algunos de estos aspectos:

- ✓ **Diseño visual atractivo:** La interfaz de usuario debe ser visualmente atractiva y coherente con la imagen del centro educativo. Seleccionaremos una paleta de colores adecuada para mejorar significativamente la apariencia de la aplicación.
- ✓ **Estructura clara y organizada:** La interfaz de usuario debe ser clara y fácil de entender. Esto lo lograremos mediante el uso de una estructura organizada y jerarquizada, donde los elementos importantes de la aplicación estarán ubicados en lugares visibles y fáciles de acceder. La aplicación también tendrá una navegación sencilla e intuitiva para que el usuario pueda desplazarse por las diferentes secciones de manera fácil.



- ✓ **Uso de iconos y etiquetas claras:** El uso de iconos y etiquetas claras ayuda a los usuarios a entender el propósito de cada función de la aplicación. Es por ello por lo que la selección de iconos y etiquetas que sean fáciles de entender y que no generen confusión en el usuario serán una tarea tan importante.
- ✓ **Mensajes de retroalimentación:** La aplicación proporcionará mensajes claros y concisos para informar al usuario sobre las acciones que se han llevado a cabo. Por ejemplo, cuando se realiza una reserva, la aplicación mostrará un mensaje confirmando que la reserva se ha realizado correctamente.
- ✓ **Facilidad de uso:** La interfaz de usuario debe ser fácil de usar para que los usuarios puedan realizar sus tareas de manera rápida y eficiente. Por eso enfatizaremos minimizar los pasos necesarios para realizar una tarea y evitar el uso de jerga técnica que confunda al usuario.

De esta manera, se logra una experiencia satisfactoria para el usuario y se facilitará la gestión de los espacios comunes del centro educativo.



#### **2.1.4 ESCALABILIDAD (RF)**

La escalabilidad de la aplicación es fundamental para garantizar que la solución pueda crecer y adaptarse a las necesidades del centro educativo a medida que cambian con el tiempo. En este caso, la aplicación debe ser capaz de gestionar un creciente número de usuarios y espacios reservados sin afectar el rendimiento y la velocidad de la aplicación.

Para lograr una alta escalabilidad, es necesario considerar el diseño de la arquitectura de la aplicación. La aplicación debe diseñarse para que los diferentes componentes puedan escalar vertical u horizontalmente, según sea necesario.

Además, es importante considerar la capacidad de la aplicación para integrarse con otros subsistemas y tecnologías para permitir una mejor gestión de datos y recursos. La integración con otras aplicaciones y servicios puede ser necesaria para mejorar la funcionalidad y la eficiencia de la aplicación.

En resumen, la escalabilidad de la aplicación es esencial para garantizar que pueda crecer y adaptarse a medida que cambian las necesidades del centro educativo. La arquitectura de la aplicación será diseñada para soportar un mayor número de usuarios y espacios, y la capacidad de integración con otras tecnologías y subsistemas también contribuirán a la escalabilidad de la aplicación.



### **2.1.5 BENEFICIOS (RF)**

La implementación de la aplicación traerá beneficios como la optimización de la gestión de los espacios, la mejora en la organización y utilización de los recursos del centro, una mayor eficiencia en la realización de reservas, y una mejora en la experiencia de los usuarios del centro.

En conclusión, aplicar para la gestión y realización de reservas de los espacios comunes del centro escolar es una herramienta digital que, al implementarse adecuadamente, puede mejorar la organización y uso de los recursos del centro y la experiencia de sus usuarios. Con su amplia gama de funcionalidades, acceso seguro y escalabilidad, la aplicación se presenta como una solución eficiente y eficaz para la gestión de los espacios comunes del centro educativo.

### **2.1.6 REQUISITOS NO FUNCIONALES:**

#### **✓ Reserva de espacios:**

##### **• Alumnos:**

- Creación de reservas: recibe los datos de Flutter, esta hace una petición a la API y de aquí se almacenan los datos en la base de datos (todo esto siguiendo las limitaciones estipuladas).
- Modificación de reservas: mismo proceso que el anterior, pero buscando previamente la reserva a eliminar.
- Anulación de reservas: Busca la reserva a eliminar por su identificador y la borra de la base de datos.

##### **• Profesores:**

- Creación de reservas: recibe los datos de Flutter, esta hace una petición a la API y de aquí se almacenan los datos en la base de datos (todo esto siguiendo las limitaciones estipuladas).
- Modificación de reservas: mismo proceso que el anterior, pero buscando previamente la reserva a eliminar.
- Anulación de reservas: Busca la reserva a eliminar por su identificador y la borra de la base de datos.

- **Administrador:**
  - Modificación de reservas: puede modificar cualquier reserva ya que no tiene limitaciones, el proceso es de igual forma que en los anteriores.
- ✓ **Gestión de espacios y reservas:**
  - **Administrador:**
    - Gestión de espacios: puede realizar tareas de actualización, adición, eliminación y comprobación del espacio, haciendo peticiones a la API que realizan cambios en la base de datos.
    - Verificación de reservas: puede editar cualquier reserva modificando el atributo estado de esta, lo que afecta en la visualización del usuario dentro de la aplicación.
    - Modificación de reservas: funciona de igual forma que para el punto de modificación anterior, pero en este caso con más campos modificables gracias a los roles que tiene.
    - Creación de reservas: debe añadir todos los campos solicitados, el servidor comprueba la entrada de datos corrigiendo los que falten o añadiéndolos por defecto, posteriormente hace la llamada a la API.
- ✓ **Asignación de perfiles:**
  - **Administrador:**
    - Creación de perfiles: puede crear usuarios que luego tendrán capacidades a la hora de utilizar la aplicación.
    - Modificación de perfiles: tiene la posibilidad de editar campos básicos como su rol, sus créditos, pero no es capaz de visualizar ni su contraseña ni editar datos como nombre o correo.
    - Eliminación de perfiles: pueden eliminar un perfil en cualquier momento, desapareciendo este de la base de datos. Es tan simple como una petición a la API tras pulsar un botón en la aplicación.
    - Asignación de espacios y limitaciones: pueden acceder a la sección de espacios para añadir uno nuevo y asignarle una serie de limitaciones en tiempo de reserva, precio, etc. Esto se traduce en una dificultad para el usuario a la hora de emplear las funcionalidades de la aplicación.

### 2.1.7 PLATAFORMAS (RNF)

Ante las cuestiones sobre las plataformas en el apartado de requisitos funcionales, se ha analizado las diferentes opciones posibles para implementar en nuestro proyecto.

Uno de los problemas a plantear en estos proyectos es el alcance, hay muchos dispositivos en el mercado que se utilizan diariamente y muchos con sistemas operativos diferentes, lo que provoca que su desarrollo sea independiente.

Ante este problema también se plantea la necesidad de abarcar un panorama relacionado con el desarrollo web (para los apartados administrativos). Por todo esto, el camino elegido ha sido la programación multiplataforma.

Frameworks para desarrollar aplicaciones multiplataforma existen muchas, entre las más conocidas y nuestras opciones estaban: Compose Multiplatform, Kotlin Multiplatform, Flutter, React Native o desarrollo independiente de las aplicaciones (Android, Swift...).

Tras un largo debate, llegamos a la conclusión de que, por contacto, conocimientos base y comodidad para el equipo a la hora de desarrollar, la mejor opción para nosotros era Flutter. Todas las otras opciones eran igual de válidas, incluso algunas ganarían al framework elegido en situaciones donde se tiene especial en cuenta el rendimiento, optimización, carga y levantamiento de máquinas para compilar las aplicaciones y procesarlas. A pesar de esto, hemos decidido centrarnos en Flutter para esta parte frontend.

Flutter es un SDK pensado para desarrollar aplicaciones de alto rendimiento y fidelidad a diversos sistemas operativos. Se organiza en capas, lo que facilita su personalización, y utiliza como lenguaje principal Dart. Dart es un lenguaje familiar Java y JavaScript, este incluye en sus opciones facilidades para manejar tipos de datos, utilización de widgets y elementos gráficos de una forma sencilla, trabajo con estilos generales y personalizados y uso de elementos de animación, entre otros. Durante su desarrollo, ejecuta diversas máquinas que permiten ver el avance de tu aplicación multiplataforma en cada equipo de destino.



Flutter cuenta también con una amplia documentación tanto de páginas oficiales como de páginas destinadas a la educación, lo que hace su aprendizaje aún más sencillo. A su vez, existe un portal donde los usuarios publican de forma oficial librerías y paquetes útiles para implementar en las aplicaciones que desarrollamos.



### **2.1.8 NUESTRA PROGRAMACIÓN BASE (RNF)**

Para que nuestra aplicación pueda acceder a los datos que precisa y rendir eficazmente ante la entrada de usuarios y de tráfico interno, se ha creado una estructura basada en microservicios que basa la implementación de diversas API REST.

Cada API REST se ocupa de gestionar un modelo de datos en concreto, y con ello, las funcionalidades base asociadas. La API de Usuarios se ocupa principalmente de su autenticación y de la creación y de edición de usuarios que consumirán nuestra aplicación.

La API de espacios, ligada a la de reservas, trata internamente la reserva, creación y recibimiento de nuevos espacios según unos requisitos establecidos, y la posibilidad de administrar y realizar acciones sobre estos espacios para dar servicio a los usuarios.

Existe otra API, la llamada comúnmente como Gateway, que se ocupa de organizar el tráfico de datos entre las APIs, y orquesta el funcionamiento de estas para el buen funcionamiento de nuestra aplicación. A su vez, es la que se ocupa de la lógica de negocio, y de cumplir con los requisitos funcionales establecidos.

Para poder llevar a cabo esta implementación, se plantearon dos posibles soluciones, una de ellas era el uso de frameworks como Ktor, para crear APIs (pensado también teniendo en cuenta que nuestro lenguaje para el Backend principalmente sería Kotlin), por otro lado, se pensó también en Spring Boot, una tecnología muy potente capaz de crear y gestionar proyectos de gran escala y escalabilidad.

Ante un análisis de ambas opciones, se llegó a la conclusión de que, puesto que teníamos varias APIs en la aplicación, no era necesario decantarnos por una tecnología, y habiendo trabajado con ambas, se podían implementar ambas. Por esto, se ha decidido utilizar Spring Boot para la API de Usuarios, pensando también en las opciones de seguridad que ofrece este framework, sobre todo queriendo destinarlo a funciones de autenticación.



Para las APIs de espacios y reservas, nos hemos decantado por Ktor, una tecnología que hemos trabajado bastante y de la cual poseemos unos conocimientos base que nos ayudan a su implementación.



En relación con esa API Gateway, el framework elegido también fue Ktor, como hemos explicado antes, su perfil de desarrollo nos encajaba perfectamente en la creación de este servicio, el cual actuaría como director de orquesta y se encargaría de organizar todo el núcleo y la base de nuestra aplicación. Ktor es junto a Kotlin una pareja de tecnologías que combinan muy bien sus recursos y que, durante el tiempo de desarrollo, han demostrado una muy buena capacidad que se traduce en eficiencia y potencial.

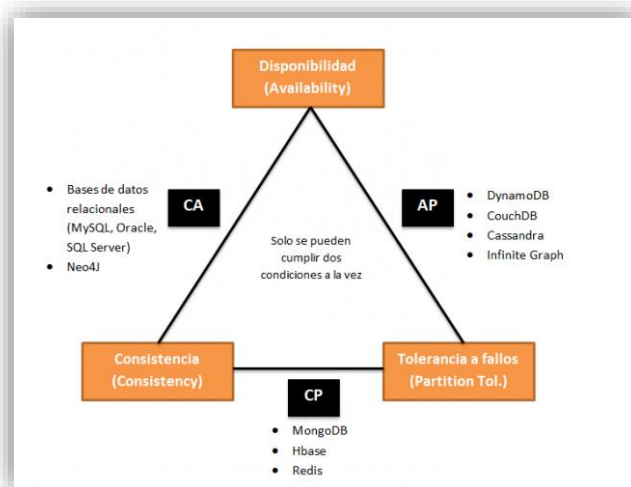


- **Kotlin ante cualquier otro lenguaje de programación:** Kotlin es un lenguaje de programación moderno que destaca por su potencia a la hora de combinar la programación orientada a objetos con características más funcionales. Kotlin tiene una integración muy buena con Java (uso de la misma Java Virtual Machine), facilitando el uso de librerías enfocadas a este último, y a la integración de proyectos que pueden compartir ambos. Hay que tener en cuenta que Ktor es un framework con una notoria sinergia compartida con Kotlin, esto hace que la elección de ambas tecnologías venga dada de la mano. A esto se suma que Spring, pensado para Java, cumple los requisitos de compatibilidad mencionados, la unión de las tres produce un empleo de las tecnologías funcional y competente.

Consideramos que esta flexibilidad que otorga Kotlin no es tan amplia como la que nos hubiese podido ofrecer otro lenguaje como C#, el propio Java directamente, o cualquier otro lenguaje de programación. Todos los lenguajes son útiles y capaces de desarrollar nuestro proyecto, pero no todos se ajustan de la misma forma bajo nuestro criterio, y para nosotros este es el punto que ha marcado la diferencia.

### 2.1.9 BASES DE DATOS (RNF)

Se ha decidido utilizar un único modelo de bases de datos para el proyecto, esta decisión se debe a nuestro interés por mantener una integridad en el desarrollo. Consideramos que un tipo concreto de base de datos en todas nuestras APIs, permite una mejora en la conectividad entre ellas, la consistencia, la escalabilidad, el mantenimiento y facilidades en su configuración.



- **Consistencia:** Al utilizar la misma base de datos para todos los microservicios, se garantiza una mayor coherencia en la forma en que se almacenan y gestionan los datos. Esto facilita la colaboración entre los equipos de desarrollo y simplifica la creación de aplicaciones y servicios que utilizan datos comunes.
- **Facilidad de configuración:** Si todos los microservicios utilizan la misma base de datos, la configuración y el mantenimiento se simplifican. Esto puede reducir la complejidad y el tiempo necesario para implementar y gestionar la infraestructura de la base de datos.

- **Escalabilidad:** Las bases de datos no relacionales como MongoDB son conocidas por su escalabilidad horizontal y capacidad para manejar grandes volúmenes de datos y estructuras flexibles. Esto es especialmente útil en el contexto de microservicios, donde cada servicio puede crecer y escalar independientemente.
- **Flexibilidad:** Las bases de datos no relacionales permiten almacenar datos sin una estructura clara, útil cuando los datos no pueden organizarse fácilmente en tablas como en las bases de datos relacionales. Esto facilita el almacenamiento y la gestión de datos en aplicaciones y servicios con requisitos de datos variados y cambiantes.

Como hemos destacado en el apartado de escalabilidad, en primera instancia no teníamos claro un modelo de bases de datos claro o un tipo definido, simplemente considerábamos que los requisitos de nuestra aplicación nos permitían poder emplear cualquier tipo y este podría acoplarse eficazmente en nuestro proyecto. A pesar de esto, decidimos que una de las soluciones que más nos llamaba la atención era una base de datos no relacional. Consideramos que de una base de datos no relacional podemos destacar fácilmente su dinamismo y flexibilidad, pudiendo adaptarse más fácilmente a los contextos que una base de datos relacional. Por otro lado, consideramos que su procesamiento de datos y tolerancia a fallos también es mayor, gracias a que tiene una distribución mayor en su arquitectura.

También podríamos destacar su bajo costo de mantenimiento en comparación con las bases de datos relacionales, unido a su agilidad y novedad, ya que, actualmente, muchas aplicaciones con grandes flujos de tráfico de datos e interacciones como LinkedIn, Orange o Telefónica tiene proyectos en sus manos donde trabajan con estas.

Una de las bases de datos no relacionales que dominan el mercado es MongoDB, con su alta capacidad de rendimiento, buena implementación y gran distribución, llegamos a la conclusión de que, en una labor de crear un proyecto moderno, fácilmente mantenible y escalable, MongoDB nos podía otorgar los recursos necesarios para dicha tarea.



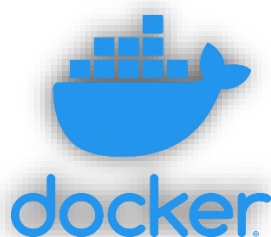
#### **2.1.10 DESPLIEGUE (RNF)**

No puede albergar en un servidor local una aplicación destinada desde su nacimiento al uso público de la misma, es imposible. Por eso, con base en los conocimientos adquiridos en este curso y aplicados en muchas asignaturas, la respuesta a este dilema nos fue sencilla. Desde el primer momento nos decantamos por Docker como recurso para crear una estructura organizada de despliegue para su posterior distribución. Desde el primer momento el objetivo ha sido claro, buscar una organización basada en una estructura propia (on-premises), donde el instituto fuese capaz de gestionar y mantener todos los servicios que conforman este proyecto, los servidores a los que recurre y mantienen sus plataformas, y todos los requisitos que se precisan.

Para la base de datos se creó un contenedor con una imagen de MongoDB con los datos de nuestra aplicación fácilmente accesibles. A su vez, a este contenedor se conectan todos nuestros microservicios, los cuales también se encuentran en contenedores independientes, todos conectados entre ellos almacenados en un stack.

Para tener desplegado nuestra aplicación durante todo el día, hemos utilizado un servidor del instituto creado expresamente para esta función, donde se ha publicado nuestro stack. En este despliegue solo se expone el puerto de nuestra API General (puerto 1212), siendo todos los demás inaccesibles desde el exterior. De esta forma, la aplicación es manejable a todas horas desde cualquier plataforma disponible.

En relación con la plataforma web, la forma de desplegar esta ha sido publicar en un servidor web el cliente final, para que sea accesible siempre. Como se ha mencionado anteriormente, este servidor también pertenece al centro, continuando con esa filosofía de estructura y desarrollo propios.



#### **2.1.11 ESCALABILIDAD (RNF)**

En conclusión, consideramos que hemos elegido, de un sinfín de tecnologías más que aptas, un grupo de estas que cumplen con los requisitos establecidos y que, a pesar de que otras tecnologías pueden tener mayores ventajas en algunos aspectos (aunque también mayores desventajas), el equilibrio entre comodidad en el desarrollo, aprendizaje y eficacia ayuda a que la aplicación sea una aplicación fácil de desarrollar y fácil de mantener.

El uso de tecnologías antiguas o estándares nuevos también puede reducir la escalabilidad del proyecto, ya que la falta de conocimiento general en el sector o de documentación dificulta su mantenimiento.

Estos factores determinan la vida de nuestra aplicación a largo plazo, por lo que una buena elección de tecnologías evita problemas a futuro para nuestro equipo y futuros desarrolladores (si existe).

#### **2.1.12 SEGURIDAD (RNF)**

Dada la necesidad de aplicarnos de poder acceder a ella solo de forma segura, se decidió emplear un recurso conocido como el JSON Web Token (JWT).

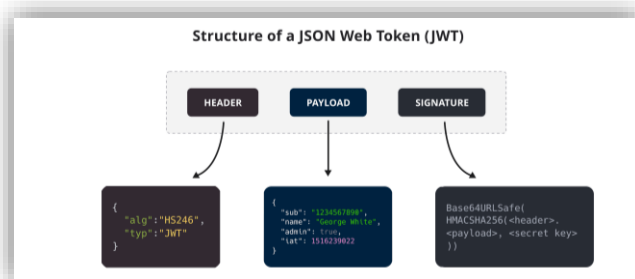




JWT es un estándar para transmitir información segura en internet mediante archivos en formato JSON, un tipo de archivo de texto plano con el que se pueden crear parámetros y asignar un valor. (“¿Qué es JSON Web Token? | KeepCoding Bootcamps”)

JWT está compuesto por varias partes:

- El header: es la parte del token que posee la información sobre el algoritmo de cifrado elegido para proteger nuestros datos y el tipo de token.
- Los claims: contienen la información que transmitirá nuestro token, estos datos pueden ser valores relevantes para emplear los mismos posteriormente en nuestra aplicación, o datos necesarios para autenticar el usuario.
- La firma se usa para verificar la identidad del token, sirve para asegurar que los datos que transporta no se manipularon durante su transmisión, siendo una garantía de que el contenido transportado llegó de forma segura.



JWT es una tecnología empleada en gran parte del mercado, existen muchos proveedores de tokens que fomentan el empleo seguro de proyectos y considerábamos que, dadas nuestras necesidades y requisitos, el uso de un token mediante JWT podía limitar eficientemente las posibilidades de ruptura en relación con la seguridad de nuestro proyecto.

### 2.1.13 TESTING (RNF)

Desde ya el año pasado, hemos aprendido que las pruebas y saber garantizar que un código funciona bien es una de las partes más importantes de un proyecto de programación. Esta aplicación reúne muchos servicios, cada uno con una programación detrás densa y compleja, que debe probarse antes de usarlo.

Para poder hacer pruebas de nuestro código y garantizar que las funcionalidades cumplen con los resultados esperados, hemos recurrido a tecnologías de testeo como son JUNIT 5 y Mockito:

- JUNIT5: es un framework que facilita al usuario la realización eficaz y prospera de pruebas en su código, comparando resultados esperados con los resultados reales que proveen nuestras funciones. JUNIT recurre a la realización de test unitarios para hacer posibles estas pruebas.

Un test unitario es una comprobación aislada de una parte concreta de nuestro código, en concreto de una función, lo que hace que su resultado no dependa de otros elementos de nuestro código. De esta forma podemos comprobar el nivel de lógica y el comportamiento de nuestro proyecto.



- Mockito: es un framework que permite la realización de pruebas en el código mediante la simulación de este.

Un test simulado o “mockeado” consiste en crear clones de los objetos que hay en nuestro código sobre los cuales se realizan las pruebas, de esta forma, no actuamos directamente sobre nuestro servicio, sino que trabajamos en un entorno completamente independiente, aislado e irreal.



Explicado este análisis, consideramos fácil de entender que, en una aplicación organizada por servicios, los cuales tienen como pilar el uso de API REST, era necesario poder simular el comportamiento de estas rutas. Por otro lado, para las pruebas de repositorios, servicios, útiles y funciones aisladas, un test unitario nos da la garantía de que el resultado de esa prueba será veraz y reflejo del comportamiento de nuestra aplicación.

Hay otras formas de realizar pruebas en código de programación, entre ellas, conocemos E2E Test, test funcionales, test de integración, test de regresión, etc. Pero, según nuestras necesidades y requisitos dados, veíamos más que suficiente el uso de test simulados y unitarios para garantizar el funcionamiento de nuestra aplicación.

#### **2.1.14 INYECCIÓN DE DEPENDENCIAS (RNF)**

La inyección de dependencias es una técnica basada en el principio SOLID de Inversión de Dependencias el cual explica que los módulos de alto nivel no deben depender de módulos de bajo nivel de forma directa, sino que ambos deben depender de abstracciones. En otras palabras, los objetos no crean sus propias dependencias directamente, sino que se proporcionan desde una fuente externa.

En nuestro caso, al usar un framework basado en Spring en uno de nuestros microservicios, no es necesario utilizar una librería externa, ya que Spring tiene un sistema de inyección de dependencias propio que hace esto posible.

En caso de los microservicios que utilizan el framework de Ktor, la cosa se complica algo más, ya que, al no existir una forma directa de hacerlo, hay que recurrir a una librería aparte. Ante este dilema, vimos la solución bastante clara, ya que no veíamos un mejor candidato para realizar esta tarea en un sistema basado en Kotlin que Koin.



Koin proporciona una solución simple y declarativa para gestionar las dependencias de nuestro proyecto. Siendo compatible con una gran cantidad de aplicaciones y frameworks, puedes definir fácilmente módulos

que representen componentes y sus dependencias. Los módulos se pueden configurar utilizando una sintaxis clara y concisa, lo que simplifica la definición de las dependencias y su alcance.

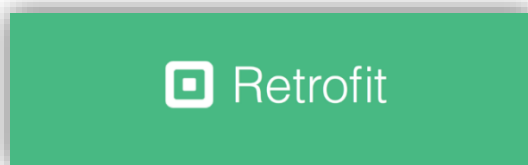
Una de las características distintivas de Koin es su enfoque basado en funciones y DSL, esto permite una configuración flexible y legible, lo que facilita la comprensión y el mantenimiento del código.

### 2.1.15 RETROFIT Y KTORFIT

Dada la estructura de nuestro proyecto y la forma de comunicar las diferentes partes de este, sabiendo que íbamos a seguir una forma de desarrollo basada en APIs, no podía faltar una tecnología capaz de consumir dichas APIs para llevar a cabo las labores requeridas en nuestro proyecto.

Durante el curso, hemos trabajado con dos tipos distintos de formas de consumir APIs, ambas ligadas a dos tecnologías diferentes: Retrofit y KtorFit:

- **Retrofit:** es un cliente que permite realizar peticiones a servidores de tipo GET, POST, UPDATE, DELETE... y también gestionar diferentes tipos de parámetros para desarrollar nuestras peticiones/consultas. Por otro lado, Retrofit también es muy útil cuando ese tipo de peticiones tienen que ver con ficheros (para imágenes), secciones multiparte, etc... Es comúnmente utilizada en el mercado para el desarrollo en Android, pero, como hemos visto durante el curso, se puede utilizar en muchos otros entornos.



- **KtorFit:** es un cliente inspirado en Retrofit y enfocado en sistemas desarrollados con Kotlin, tiene una forma parecida de funcionar que el anterior mencionado, es muy fácil de instalar en nuestro proyecto, y tiene una gran compatibilidad con el framework de Ktor.



Conociendo ya ambas tecnologías, el grupo estudió durante un tiempo cuál de ellas era la que más se ajustaba a nuestro entorno de trabajo, a lo que buscábamos, y a la forma de desarrollar la aplicación. Rápidamente nos dimos cuenta de que, recordando que nuestra estructura y arquitectura se basaba en servicios independientes, y, por ende, la consumición de cada una de las APIs podía ser también independiente, no teníamos la necesidad de elegir una de ellas.

Es por esto por lo que decidimos implementar en nuestro proyecto ambas tecnologías, centrando el uso de KtorFit para la consumición de APIs que iban a devolver y enviar datos modelados de carácter simple, y Retrofit para realizar peticiones algo más complejas, como podría ser el empleo de imágenes, etc.

### 2.1.16 CACHE

Otra cosa aprendida en este curso es que nuestra aplicación debe responder por sí misma. Ante cualquier problema, del tipo que sea, debemos buscar siempre el desarrollo de una aplicación a prueba de fallos, donde la experiencia del usuario no se ve gravemente afectada por dichos sucesos.

Dentro de este paradigma, entra en juego un factor conocido como las caches. Una caché es un tipo de almacenamiento empleado en las aplicaciones para dar respuestas a solicitudes de una forma más rápida. Reduciendo así los tiempos de espera que el usuario experimenta. Explicado de forma simple, la caché

almacena temporalmente unos datos que esta selecciona (antes programada por nosotros), para que la próxima vez que el usuario desee acceder a dichos datos, no sea necesaria la realización de una solicitud.

Bajo esta reflexión, vimos útil la implementación de dicha tecnología en nuestra aplicación, para así sumarle un punto extra a las funcionalidades de nuestro proyecto y, sabiendo que este está destinado a un usuario final, que su experiencia sea lo más agradable posible.

Dentro del catálogo de librerías y paquetes que nos da la oportunidad de poder desarrollar dicha implementación, existen varias opciones, algunas de ellas son:

- **Caffeine:** es una librería de alto rendimiento útil para Java y Kotlin. Se pueden establecer diversas configuraciones como tamaño de la caché, expiración, etc. Proporciona una API sencilla mediante la cual se puede crear dicha caché.
- **Exposed Caching:** es una extensión de Exposed para Kotlin que permite capacidades de caché para capas relacionadas con el acceso a los datos. Mejora el rendimiento de la aplicación haciendo más eficaces las consultas de datos.
- **Cache4K:** es una librería pensada para Kotlin que ofrece un almacenamiento en memoria simple con soporte para tiempos de expiración y funcionalidades basadas en el tamaño.
- **Spring Cache:** es una funcionalidad interna de Spring que permite fácilmente cachear métodos utilizando anotaciones. También proporciona un amplio soporte a bibliotecas externas como algunas mencionadas anteriormente, y otras como Guava, Ehcache y otras.

Tras este análisis, y tras un tiempo de hablar y decidir cual se ajustaba más a nuestro proyecto, llegamos a varias conclusiones. La primera de ellas es que veíamos necesario el implementar una cache por los motivos mencionados anteriormente. La segunda fue que, dentro de esa implementación, no veíamos necesario tener una funcionalidad de caché en todos los microservicios. Vimos especialmente afines a la cache los servicios de reservas y de espacios, entre otras cosas, porque era el recurso con el que los usuarios iban a tratar de forma directa, y suponiendo una afluencia de peticiones mayor que con el microservicio de usuarios (tanto por parte de los alumnos y profesores como los administradores), tuvimos clara la decisión. El microservicio de usuarios es un servicio que se queda al alcance de los administradores, y estos, por el momento, no debería tener la necesidad constante de acceder a los datos de estos, solo para ocasiones concretas por motivos concretos.

Es por esto por lo que, al menos como base, su implementación ha quedado a un lado, aunque no es descartable que se implementase en un futuro, si el rendimiento de la aplicación lo requiriese.

Sabiendo entonces que nos centraríamos en el microservicio de reservas y en el espacio para implementar la cache, ambos basados en el Framework de Ktor y con Kotlin como lenguaje de programación, finalmente nos decidimos por Cache4K, una librería ya trabajada en clase, fácil de utilizar para nosotros, conocida por todos, y altamente compatible con nuestro proyecto.



### **2.1.17 CLIENTE DE PRUEBAS DE RUTAS: POSTMAN**

Cuando se está desarrollando una API, sea el framework que sea el que estemos utilizando, se nos ha inculcado que es una buena práctica tener un entorno de pruebas donde poder comprobar si las salidas son las esperadas, los errores son los que buscamos, y en caso de error, saber fácilmente como corregirlos. Es muy tedioso y complicado desarrollar una API si el único reflejo que tenemos para saber si algo funciona es nuestra propia máquina, ya sea una consola, o la misma base de datos.

Existen clientes tanto web como de escritorio, que nos dan la oportunidad de poder hacer peticiones a rutas tanto públicas como de nuestro propio equipo para poder comprobar la funcionalidad de este, una de ellas es Postman.

Postman es una herramienta muy fácil de utilizar, simplemente necesitas conocer la url de tu API, y los campos que quieres proporcionar para cada consulta, seguido del formato, una vez hecho esto, simplemente envías las consultas y recibes la salida como si fuese un entorno real.

Hemos estado trabajando con este cliente durante gran parte del curso, y creemos que cumple con todos los requisitos necesarios para desarrollar nuestra aplicación.



### 2.1.18 CREACIÓN DE PROTOTIPOS Y MOCKUPS

Es primordial al crear la interfaz de usuario de cualquier aplicación o página web tener claro la organización de los elementos y su estilo. Por esto mismo, es de especial utilidad utilizar herramientas como Figma para poder probar y diseñar cada prototipo que se considere una buena opción. Figma permite recurrir al diseño gráfico para maquetar el prototipo de una aplicación una cantidad bastante grande de iconos, opciones de personalización, fuentes y estilos. A su vez, también permite crear presentaciones profesionales recurriendo a hilos de navegación que conectan los fragmentos entre ellos. Esto es muy útil a la hora de presentar nuestro prototipo, ya que hace que este se acerque el máximo posible a la apariencia de una aplicación ya maquetada.



Ejemplo de navegabilidad de la aplicación en el prototipo gracias a Figma:



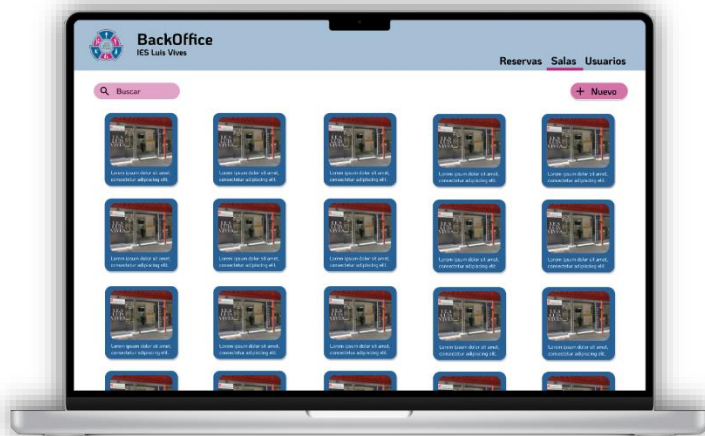
## 2.2 DISEÑO

### 2.2.1 PROTOTIPO

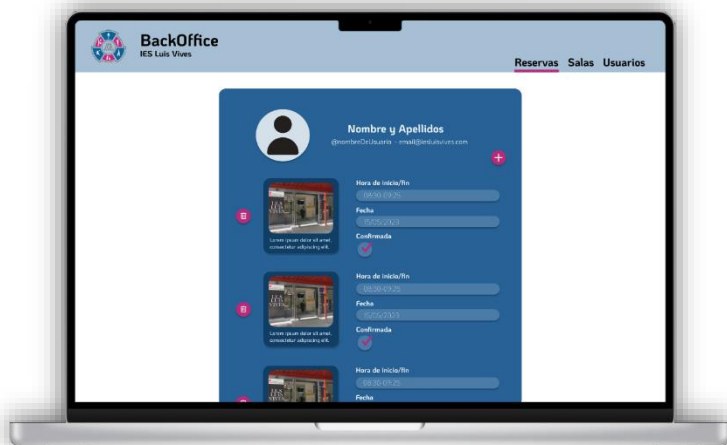
Para la creación de la interfaz, nos hemos basado en los colores del logo del centro. Una imagen de este donde predominan claramente los colores rosados y azulados, intercalando entre diferentes tonos tanto claros como oscuros. Fue bajo esta idea que fuimos conscientes de que nuestra aplicación debía regirse por esos colores en todo momento.

Tras hacer las entrevistas con el cliente y conocer sus necesidades e intereses, supimos identificar las dos partes que debía tener nuestra aplicación: una interfaz que siguiese los estándares de una web, y una interfaz de usuario adaptada para móviles.

Para la parte de la página web decidimos centrarnos en una distribución por pestañas, mediante la que pudiésemos navegar mediante una barra de navegación superior, y que en cada pestaña pudiésemos ver con claridad los diferentes elementos precisos, acompañados de botones y elementos que daban vida a sus funcionalidades.



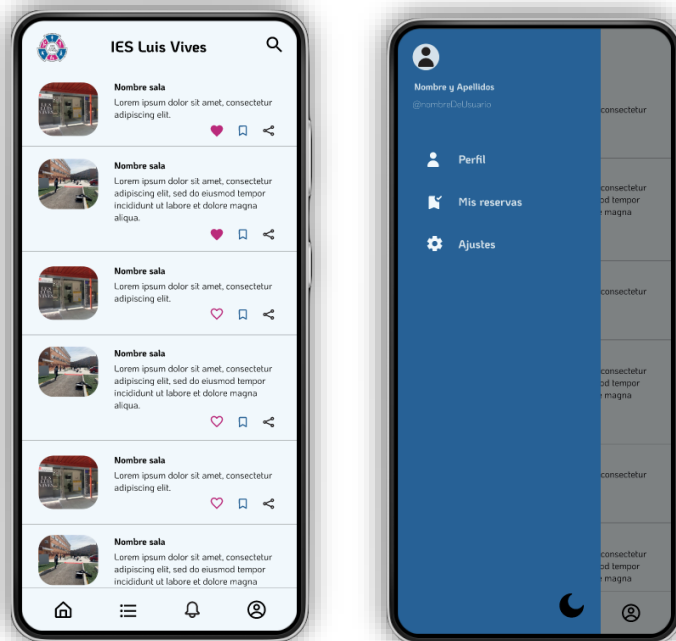
Dentro de esta vista tendríamos acceso a la gestión de dichos recursos, viéndose la pestaña de dicha gestión de la siguiente manera:



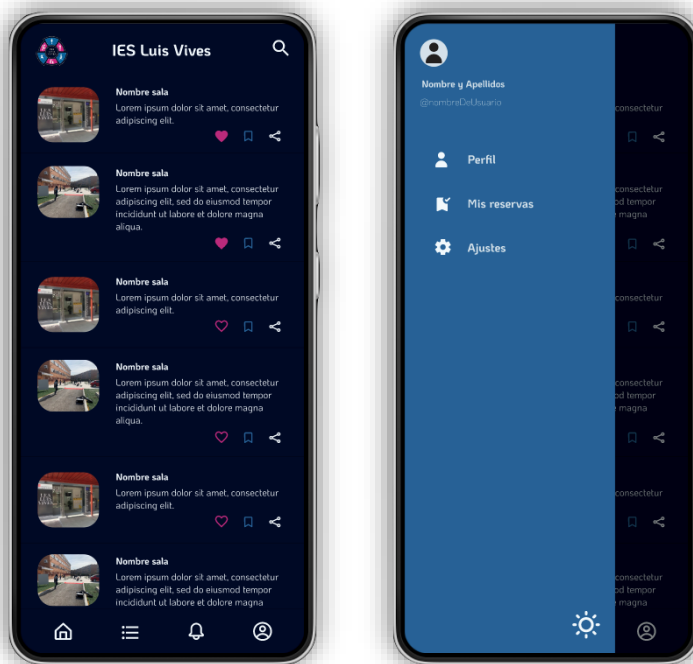


Por otro lado, para la parte de la aplicación móvil, como hemos mencionado anteriormente, decidimos buscar una estructura de las vistas basadas en lo que ya conocemos y en los elementos a los que se recurre actualmente en el mercado. Utilizamos un menú de navegación inferior donde podíamos ir cambiando entre las diferentes pestañas y, a su vez, interactuar directamente con las mismas.

De forma complementaria a este sistema de navegación, también contamos con un menú lateral para poder ver aún más opciones relacionadas con el perfil del usuario.

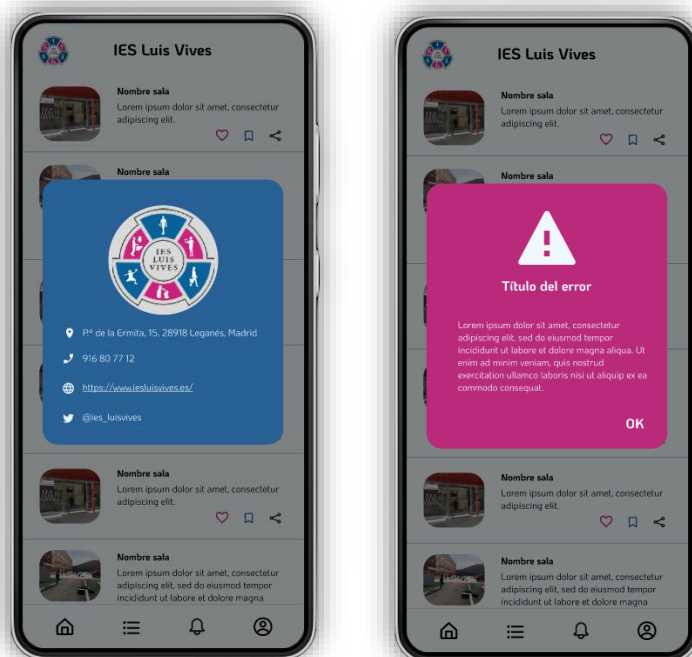


Tanto la versión web como la versión móvil cuentan con la posibilidad de poder cambiar el tema de colores a tonos más oscuros, siguiendo las preferencias del usuario:

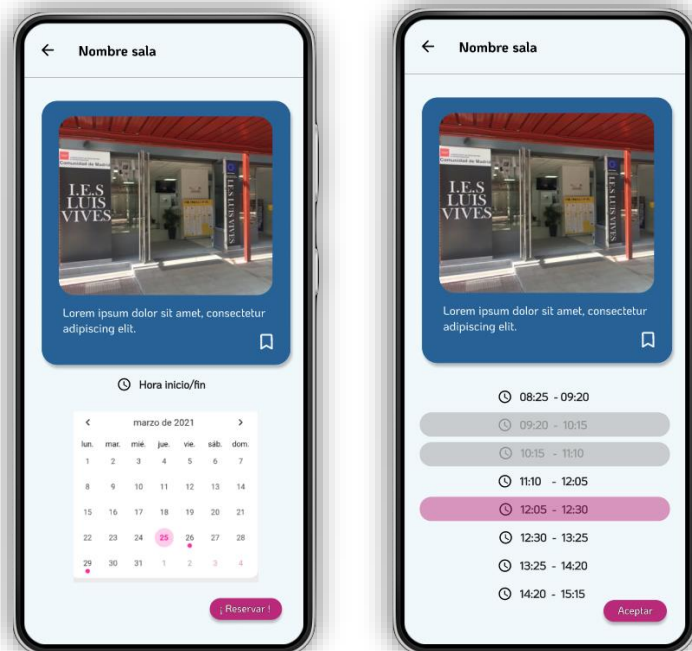




Además de esto, contamos con un sistema de diálogos basados en colores para informar al usuario de errores o advertencias:



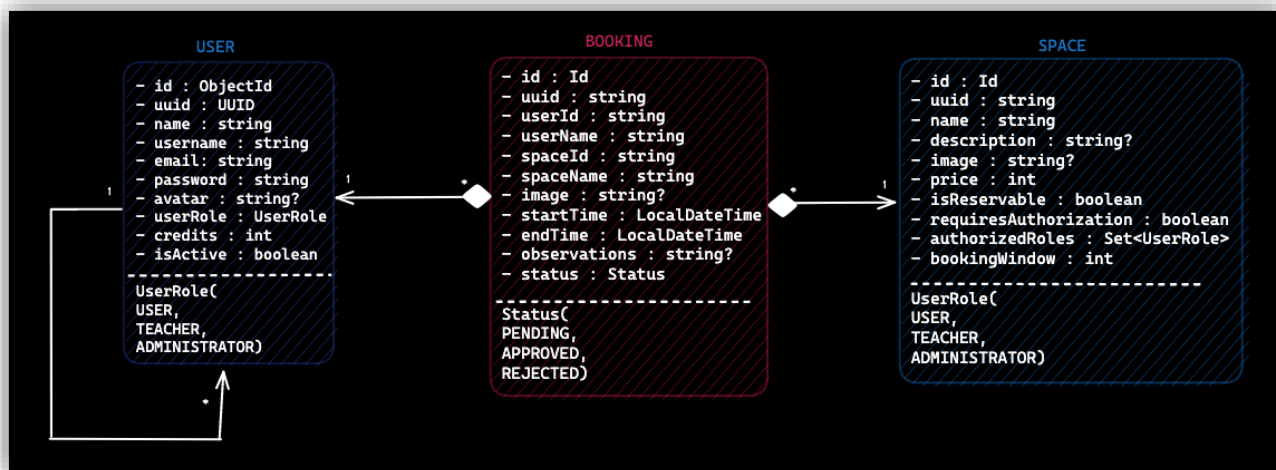
Como se puede observar, se han utilizado una gran cantidad de iconos que hacen la experiencia del usuario más intuitiva, fácil de entender y de manejar.



### 2.2.2 DIAGRAMA DE CLASES

Uno de los primeros pasos tras tener la entrevista con el cliente y recopilar todos los requisitos de la aplicación, fue hacernos un esquema consensuado sobre cómo sería el modelo de clases de nuestra aplicación.

Es por esto por lo que tras poner en común nuestras opiniones y las del cliente, se llegó a la conclusión de implementar este diagrama de clases:



En este diagrama de clases podemos ver claramente la existencia de 3 modelos de datos, los cuales son explicados uno a uno a continuación:

- USER:** el usuario puede tener tres roles distintos (USER, TEACHER y ADMINISTRATOR) los cuales determinarán sus limitaciones dentro de la aplicación. Un usuario con rol de administrador puede editar a otros usuarios, teniendo una clara distinción de permisos respecto a los profesores y usuarios normales. El modelo de usuarios tiene un id que es el usado para referirse al mismo de forma interna por la aplicación. Para hacerlo de forma externa, recurrimos a los identificadores únicos universales. Esta acción mejora la seguridad del acceso a datos de nuestra aplicación. El usuario tiene otros datos más genéricos, como puede ser el nombre, contraseña, email... pero también tiene datos que se almacenan como "string" (los avatares) donde su valor es realmente el identificador de la imagen que tiene asociada, haciendo muy fácil su emparejamiento.



- **BOOKING:** el modelo de reservas tiene tres estados definidos (PENDING, APPROVED y REJECTED), que definen, como su nombre indica, la situación de la reserva en ese preciso momento, siendo muy útil para las diferentes funcionalidades y gestiones dentro de nuestra aplicación. Las reservas cuentan también con el atributo de id e identificador único universal, pensado con el mismo propósito y cumpliendo la función de la misma forma. Más allá de dichos datos, presenta atributos que hacen referencia a otros modelos como es el identificador del espacio, su nombre, el identificador y el nombre del usuario, observaciones, y datos de especial importancia como son la fecha de inicio de la reserva y la de final, declarados como LocalDateTime, donde es fácilmente visualizable la fecha en día y la hora.

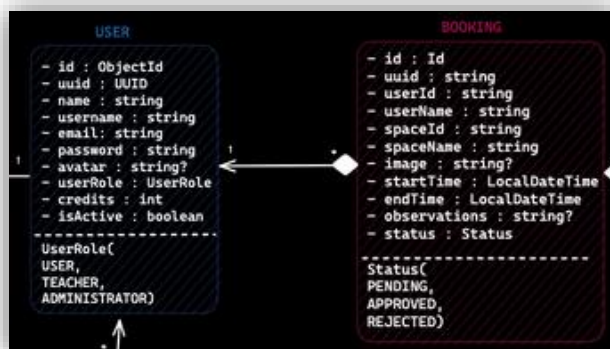
```
BOOKING
- id : Id
- uuid : string
- userId : string
- userName : string
- spaceId : string
- spaceName : string
- image : string?
- startTime : LocalDateTime
- endTime : LocalDateTime
- observations : string?
- status : Status
-----
Status(
PENDING,
APPROVED,
REJECTED)
```

- **SPACE:** los espacios usarán el atributo relacionado con los roles del usuario, porque cada espacio tiene asociada una lista con los roles que tienen permitido reservarlo. También cumple con la idea de tener un id y un identificador universal, siguiendo la misma lógica. Por otro lado, tiene atributos genéricos como pueden ser descripción, precio, ventana de reserva (los días de antelación con los que se puede reservar), un "boolean" para declarar si dicho espacio es reservable o no (similar a si está o no disponible), afectando de esta forma a la capacidad de los usuarios de visualizar dicho espacio. En este caso y al igual que en los anteriores, los atributos que hacen referencia a imágenes son "string" a los que se les asignará como valor el identificador de la imagen a la que se empareja.

```
SPACE
- id : Id
- uuid : string
- name : string
- description : string?
- image : string?
- price : int
- isReservable : boolean
- requiresAuthorization : boolean
- authorizedRoles : Set<UserRole>
- bookingWindow : int
-----
UserRole(
USER,
TEACHER,
ADMINISTRATOR)
```

Sobre cómo se relacionan los modelos, podemos diferenciar tres tipos de relaciones distintas las cuales se explican a continuación:

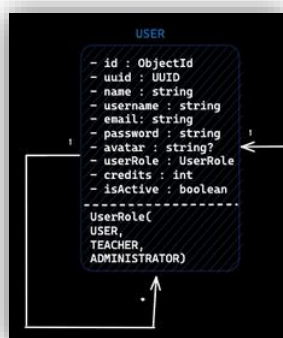
- **Relación de composición entre usuarios y reservas:** para poder realizar una reserva, es necesaria la existencia de un usuario, es de ahí de donde nace la relación de composición. A su vez, un usuario puede hacer muchas reservas siempre que se cumplan sus limitaciones, pero una reserva solo estará asociada a un usuario, pudiendo ver datos de este con identificador universal o nombre.



- **Relación de composición entre espacios y reservas:** al igual que es necesario un usuario para realizar una reserva, también es necesario que exista un espacio del cual se producirá la reserva, es de aquí de donde nace esta segunda relación de composición. Una reserva solo tendrá asociada un espacio, del cual podremos saber datos como el nombre o el identificador universal, pero un espacio puede tener múltiples reservas en relación, siempre respetando limitaciones de tiempo, entre otras...



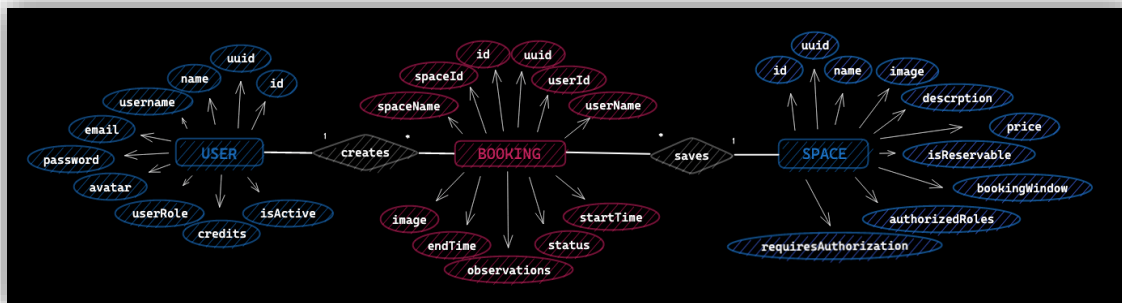
- **Relación de reflexión de usuarios:** en los usuarios, existe una relación la cual permite al usuario que tenga rol de administrador, gestionar otros usuarios, ya sea editarlos, añadir nuevos, etc. Es por esto por lo que se crea una reflexión sobre sí mismo visible en el diagrama de clases.



### 2.2.3 DIAGRAMA ENTIDAD-RELACIÓN

Junto al diagrama de clases, también se estudió y desarrolló un diagrama entidad-relación que representa para estructurar la base de datos.

A continuación, se resume detenidamente el contenido de dicho esquema:



#### USUARIOS

ATRIBUTO	TIPO
<b>ID</b>	OBJECTID
<b>UUID</b>	UUID
<b>NAME</b>	STRING
<b>USERNAME</b>	STRING
<b>EMAIL</b>	STRING
<b>PASSWORD</b>	STRING
<b>AVATAR</b>	STRING
<b>USERROLE</b>	STRING
<b>CREDITS</b>	INT
<b>ISACTIVE</b>	BOOLEAN

#### RESERVAS

ATRIBUTO	TIPO
<b>ID</b>	OBJECTID
<b>UUID</b>	STRING
<b>USERID</b>	STRING
<b>USERNAME</b>	STRING
<b>SPACEID</b>	STRING
<b>SPACENAME</b>	STRING
<b>IMAGE</b>	STRING
<b>STARTTIME</b>	ISODATE ()
<b>ENDTIME</b>	ISODATE ()
<b>OBSERVATIONS</b>	STRING
<b>STATUS</b>	STRING

## ESPACIOS

ATRIBUTO	TIPO
ID	OBJECTID
UUID	STRING
NAME	STRING
IMAGE	STRING
DESCRIPTION	STRING
PRICE	INT
ISRESERVABLE	BOOLEAN
BOOKINGWINDOW	INT
AUTHORIZEDROLES	ARRAY
REQUIRESAUTHORIZATION	BOOLEAN

Dado el diagrama entidad-relación, también podemos explicar detenidamente las siguientes relaciones:

- **La relación de usuario crea reservas:** un usuario podrá crear muchas si cumplen con los requisitos, y una reserva la hará solo un usuario.



- **Relación de la reserva guarda un espacio:** una reserva podrá hacer referencia a un único espacio, nunca a más de uno, mientras que un espacio puede tener una referencia en varias reservas (por ejemplo, reservas del mismo espacio a diferentes horas).

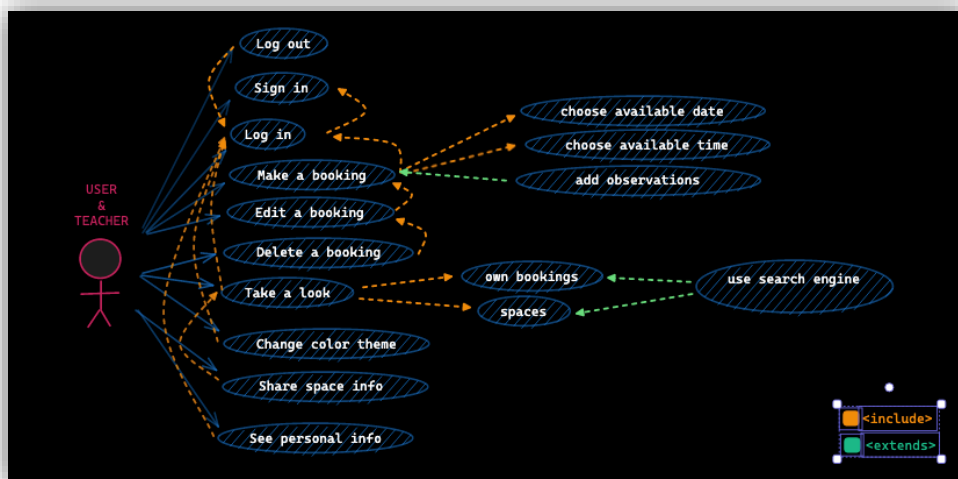




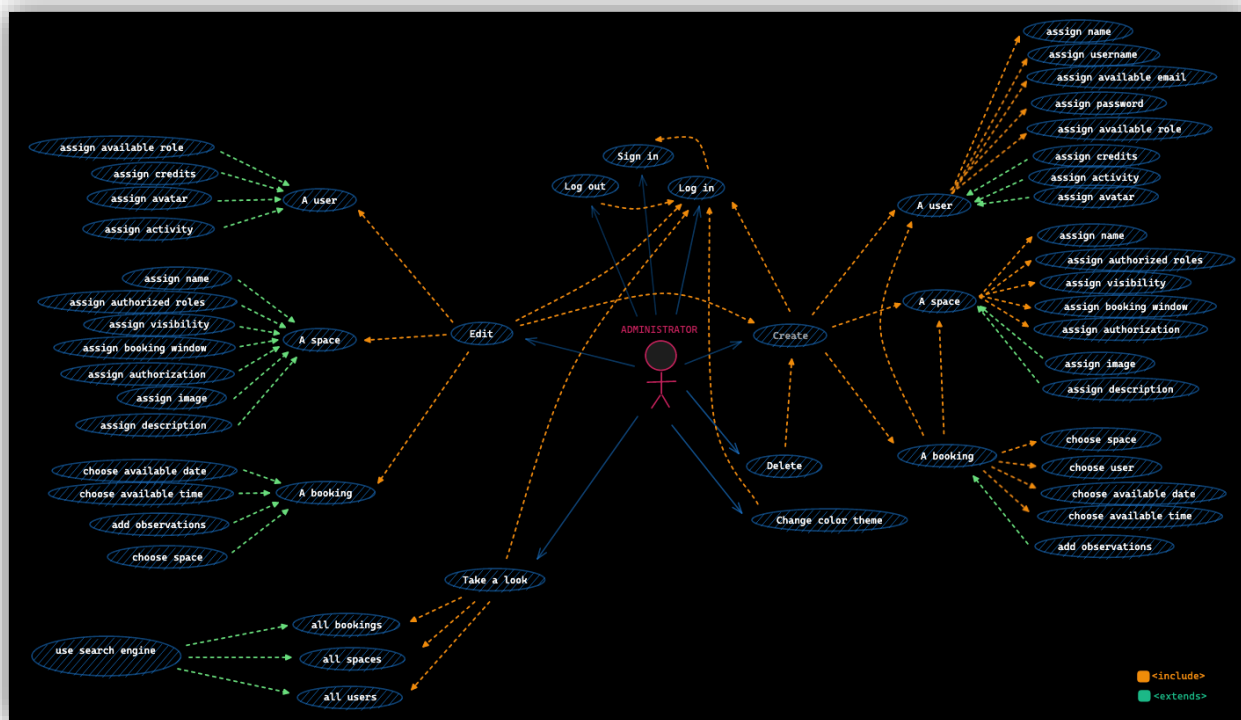
## 2.2.4 DIAGRAMA CASOS DE USO

En el primer análisis, explican los requisitos del proyecto y el estudio de estos según las necesidades del cliente, detallados con precisión, a continuación, se muestra un esquema visual que muestra los casos de uso asociados al usuario para cada role, agrupados en dos: casos de usuario administrador y casos de usuario alumno o profesor.

### CASOS DE USO PARA USUARIO ALUMNO-PROFESOR



### CASOS DE USO PARA USUARIO ADMINISTRADOR

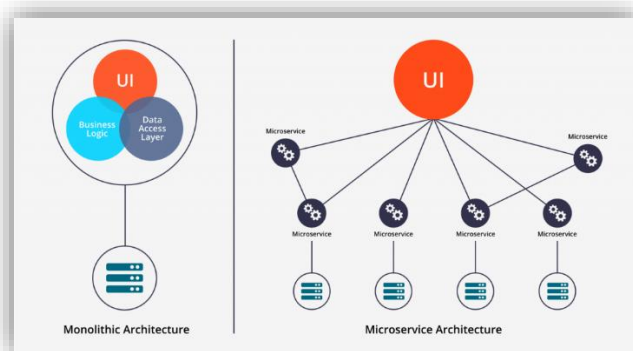


## 2.2.5 ARQUITECTURA DEL PROYECTO

### MICROSERVICIOS

Como se menciona un poco por encima anteriormente en el análisis de las tecnologías, en este proyecto se podían diferenciar claramente varios tipos de servicios, los cuales son Usuarios, Espacios y Reservas. Ante esta situación existen dos formas de implementar dicho planteamiento: uno en el que incluir todos los servicios, algo que complicaría las tareas de mantenimiento y mejoras de la aplicación, haciendo que todo dependiese de un solo conjunto, o seguir una arquitectura basada en microservicios como hemos visto en clase.

Las arquitecturas basadas en microservicios es un método de desarrollo de software por el cual podemos dividir los servicios en sistemas independientes haciendo más fácil su desarrollo, mantenimiento o implementación. Podríamos tener servicios donde se utilizan bases de datos diferentes, lenguajes de programación diferentes e incluso frameworks diferentes que funcionen independientemente. Posterior a esto, se puede desarrollar un servicio general que sirve para conectar todos los microservicios y completar la relación entre estos.

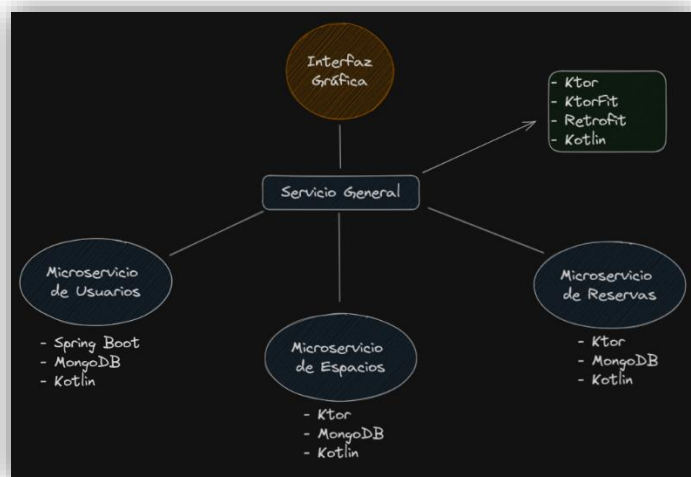


Teniendo claro este esquema, podemos llevar ahora el modelo a nuestro proyecto actual.

En nuestro caso concreto, tenemos un microservicio de usuarios, el cual está desarrollado en Spring Boot y tiene una base de datos MongoDB. El microservicio de reservas está desarrollado con Ktor, y utiliza una base de datos de MongoDB. El microservicio de espacios también ha sido desarrollado con Ktor, y a su vez, utiliza una base de datos de MongoDB.

Todos los microservicios han sido desarrollados utilizando Kotlin como lenguaje de programación.

Este esquema muestra de una forma más gráfica nuestro modelo final:





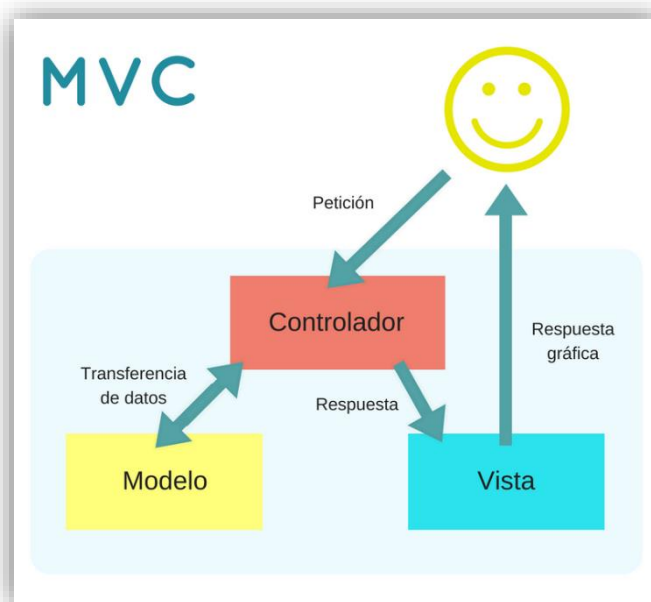
Como se puede observar en el esquema realizado, nuestro cliente se conecta de forma directa con el servicio general, conocido en nuestro proyecto como “Api Gateway”. Desde este servicio, nos conectamos a los demás microservicios de la aplicación, y vamos haciendo llamadas a unos u otros dependiendo de nuestras necesidades.

Una organización y estructura así permiten añadir más microservicios distintos a los presentes para cualquier otra funcionalidad que se quiera implementar, siendo así una herramienta muy útil si pensamos en la escalabilidad y en el desarrollo a futuro.

### **MODELO - VISTA – CONTROLADOR**

Dentro de los microservicios hemos seguido un modelo enfocado en la arquitectura MVC (Modelo Vista Controlador) (Modelo – Vista - Controlador), esta arquitectura consiste en una diferenciación de las funciones de nuestra aplicación, en este caso, nuestro microservicio. Organizamos nuestro código en componentes que cumplen sus funciones de forma independiente:

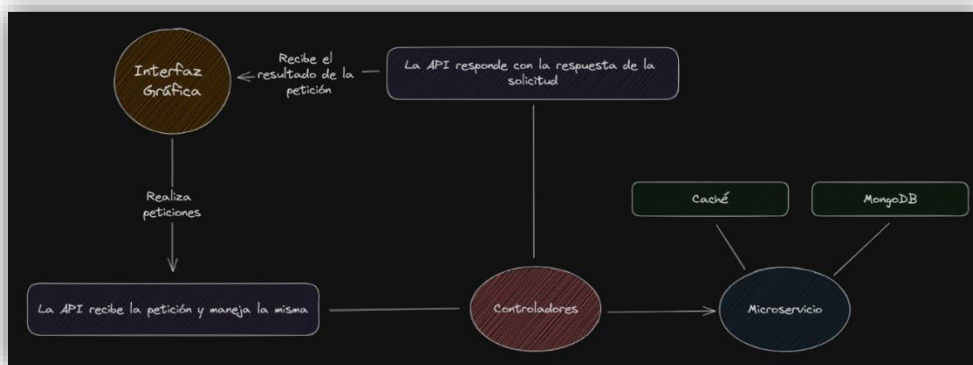
- **Modelos:** son los encargados de representar los datos empleados en la aplicación, se encargan de almacenar y recuperar los datos de nuestro recurso para posteriormente implementar la lógica de estos.
- **Controladores:** son los encargados de aplicar la lógica de los modelos y manejar las interacciones con la vista. Trabajan con los modelos para responder e interaccionar con los datos que el usuario necesita, y son el nexo entre nuestra aplicación y el producto final (el usuario).
- **Vistas:** componentes que el usuario percibe mientras usa nuestra aplicación. Son los encargados de informar a los controladores de las interacciones que este tiene con la aplicación, y posteriormente, recibir la información que mostrará al usuario.



Explicada dicha arquitectura, podemos entonces trasladar la teoría a nuestro proyecto: los controladores de cada microservicio son los encargados de manejar la herramienta base de nuestro sistema, las APIs REST. Nuestros controladores se ocupan de declarar las rutas que empleará nuestra aplicación y son los encargados de gestionar la información que aplican los modelos, para posteriormente, tras realizar según que funciones, trasladárselas al usuario.

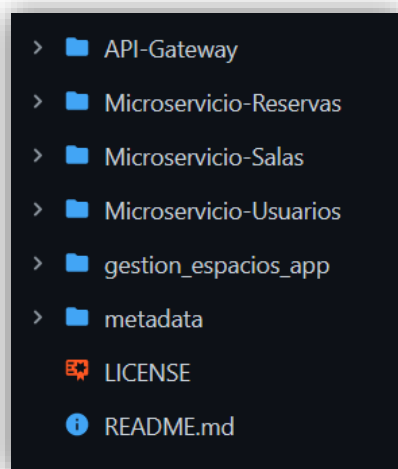
En relación con las vistas, en nuestro proyecto tenemos una vista clara que comparten todos los microservicios y sistemas, la aplicación móvil o web del usuario final. Esta vista es la encargada de enviar las peticiones al servicio general dependiendo de las interacciones del usuario. Una vez están hayan sido completadas, de nuevo recibe los datos tratados y se los muestra para que pueda visualizarlos.

En este esquema mostramos gráficamente cómo se realiza dicha comunicación:



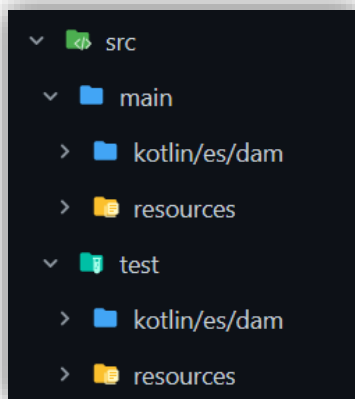
## ESTRUCTURA DE CARPETAS

El proyecto está conformado por un total de 5 proyectos distintos que están relacionados entre sí, entre los cuales encontramos el desarrollo de las cuatro APIs y el desarrollo del cliente:

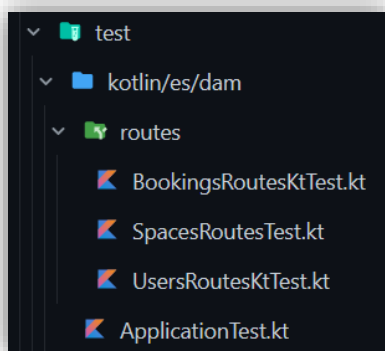


A su vez, contamos con una carpeta para almacenar documentos varios y esquemas, y archivos independientes.

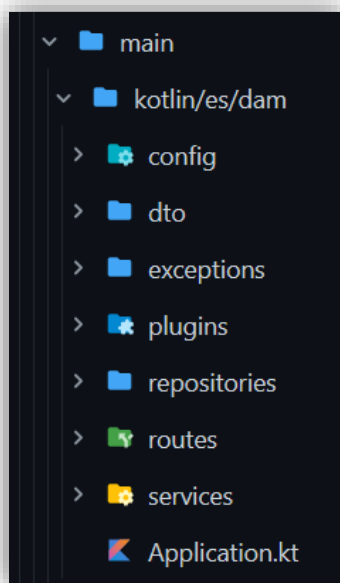
- **API Gateway:** dentro de esta carpeta tenemos el proyecto sobre el que desarrolla la API que se ocupa de interconectar todos los demás microservicios. Dentro de su carpeta src/ cuenta con una división entre los test y el desarrollo del proyecto:



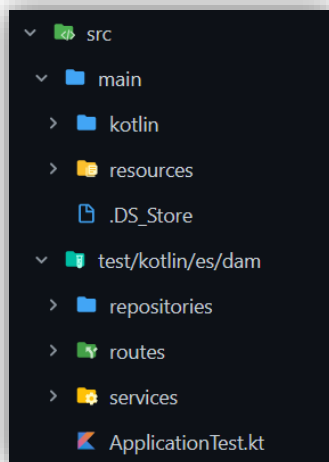
En el apartado de test se encuentran todas las pruebas de las rutas:



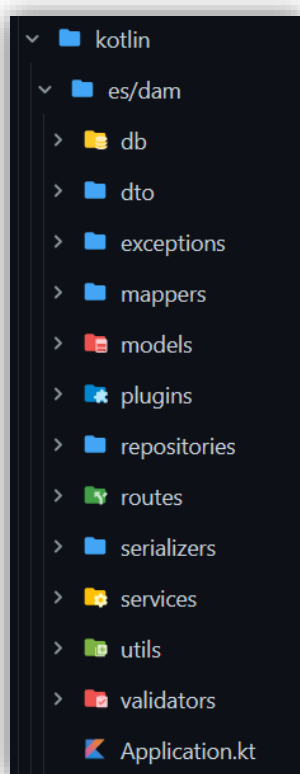
Dentro de la carpeta main/ tenemos una división entre los diferentes paquetes, de los cuales destacamos clases de configuración, plugin, repositorios, excepciones, rutas, servicios y objetos de transferencia de datos:



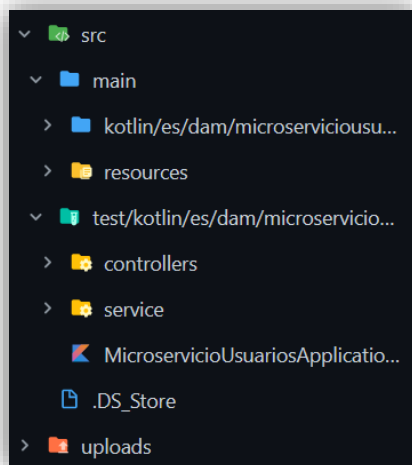
- **Microservicio-Reservas:** es el proyecto que se encarga de desarrollar la API destinada a la gestión de las reservas de la aplicación. En la carpeta src/ del proyecto encontramos de nuevo un apartado para hacer pruebas, en este caso, se clasifican como pruebas destinadas a las rutas, destinadas a los servicios y destinadas a los repositorios.



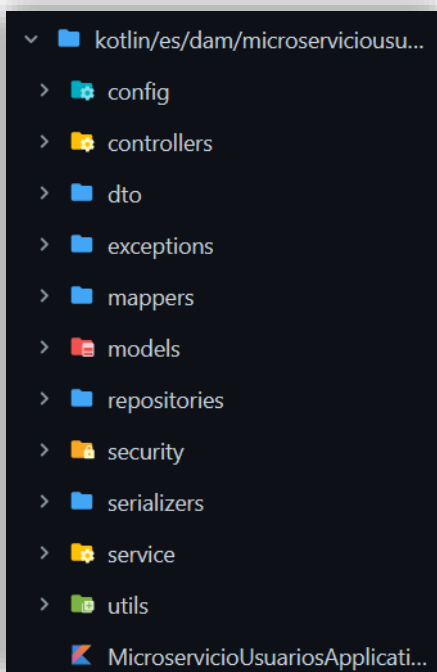
Por otro lado, dentro de la carpeta de Kotlin, encontramos los paquetes de funcionalidad de nuestra aplicación, entre las que diferenciamos clases de configuración de la base de datos, excepciones, mapeadores, repositorios y servicios, utilidades, serializadores, validadores, modelos de datos, objetos de transferencia de datos...



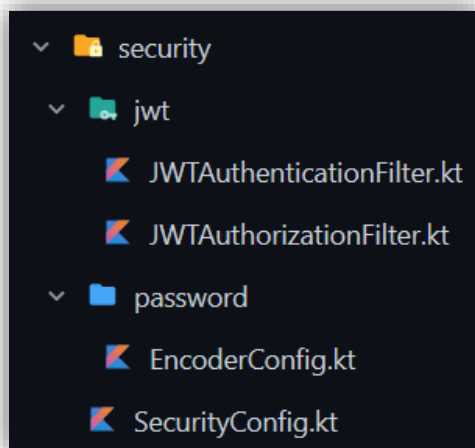
- **Microservicio-Salas:** Debido a que su implementación es la misma que el proyecto de desarrollo de la API de reservas, su estructura es exactamente igual al anterior explicado.
- **Microservicio-Usuarios:** En este microservicio tenemos una diferenciación respecto al resto de proyectos, y es que en este utilizamos Spring Boot como framework, esto hace que la división de las carpetas, y sobre todo su contenido, se vea ciertamente distinto.  
Dentro de la carpeta src/ tenemos una estructura similar a las anteriores, en este caso los controladores son las definiciones de las rutas, por lo que son los que se testean junto a los servicios, que llaman a los repositorios.



Dentro de la carpeta de Kotlin, tenemos algunas diferencias, dentro de dicha carpeta encontramos las clases de configuración, los controladores (en otros proyectos conocidos como rutas), mapeadores, excepciones, objetos de transferencia de datos, repositorios y servicios, serializadores, utilizados y un apartado de seguridad muy importante dentro de Spring Security.

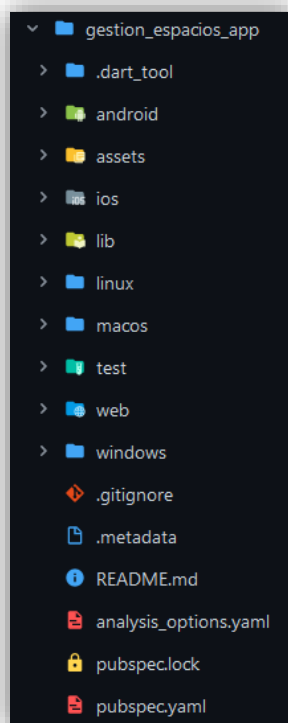


Dentro de Spring Security, recurrimos a varios paquetes donde definimos las opciones de seguridad, configuramos la autenticación mediante JWT, y la codificación de contraseñas mediante BCrypt.

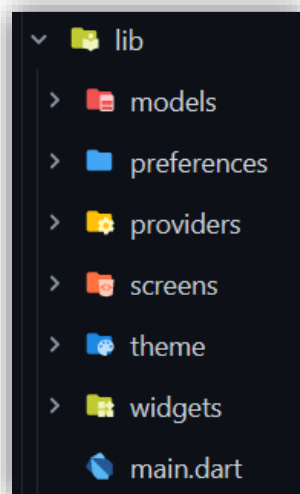


- **gestion\_espacios\_app**: este proyecto es el que alberga el desarrollo de la aplicación multiplataforma que el cliente usará para interactuar con el núcleo de la aplicación. En este caso, ya no hablamos de una estructura de Backend como en las anteriores, sino que nos movemos al campo de Flutter, con su estructura de carpetas propias.

En Flutter, encontramos una diferenciación en las carpetas, dentro del proyecto se puede acceder a los apartados de cada plataforma, donde podemos modificar ficheros de configuración, etc. En la carpeta raíz también tenemos archivos que nos permiten hacer cambios generales en la configuración.



La carpeta en la que se desarrolla el código y se declara la forma y funcionamiento de las vistas se conoce como `lib/`, esta carpeta tiene dentro una serie de paquetes que se encargan de definir los modelos de datos, configuraciones de preferencia para el usuario dentro de la aplicación, proveedores de servicios (son los encargados de consumir las APIs desarrolladas en los microservicios), las diferentes escenas que tiene la aplicación, el esquema de colores y temas de la aplicación, y los widgets personalizados de la aplicación, fragmentos de código que se incorporan en las escenas y que tienen funcionalidades y apariencia aislada.



El fichero principal que contiene la función que se ocupa de ejecutar todo, y, por ende, las escenas, es el archivo nombrado como “main”.

## 2.3 IMPLEMENTACIÓN

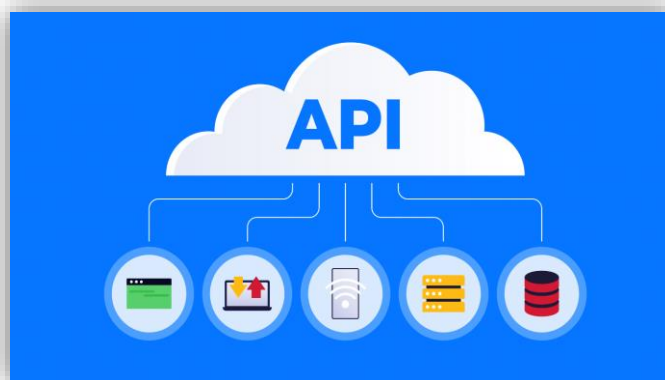
Llegamos a la fase del proyecto que más tiempo ha llevado. La implementación estaba planificada en nuestro esquema inicial del anteproyecto como una de las partes más extensas en tiempo, ocupando esta más del 50% del tiempo dedicado por cada uno de nosotros a la aplicación.

La implementación de este proyecto empieza siguiendo una arquitectura base como las mencionadas anteriormente, microservicios que albergan APIs que sirven para poder gestionar los datos que mueven los modelos. Pero una de las primeras preguntas que nos podemos hacer es cómo funciona una API RESTFUL.

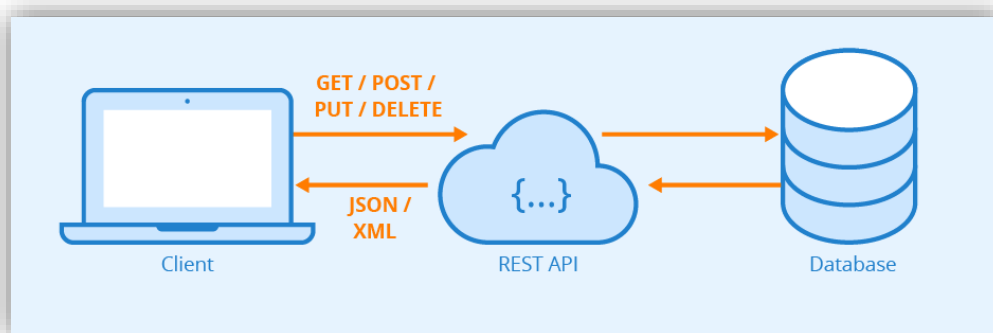
### 2.3.1 API RESTFUL

Una API RESTFUL, también conocida como una API Rest, es un conjunto de reglas que definen la conectividad de una aplicación, esta, a diferencia de las demás APIs existentes, cumple con el patrón de REST.

- **API:** una interfaz que permite comunicarse entre aplicaciones de software, las APIs son sistemas que los programadores desarrollan para cumplir funciones específicas dentro de su aplicación, que tienen un alto nivel de asociación.



Lo que podemos observar es que existe una diferencia entre una API y los patrones que a esta pueden acompañar. Un patrón REST es un mecanismo de la API que permite a esta tener una interfaz uniforme, completa eficazmente un desacoplamiento entre el servidor y el cliente, no requieren estado ni ninguna sesión del lado del servidor, pero a cambio, debemos declarar toda la información que procesará la solicitud. Permiten el almacenamiento en caché, una arquitectura en capas, y muchos otros beneficios.



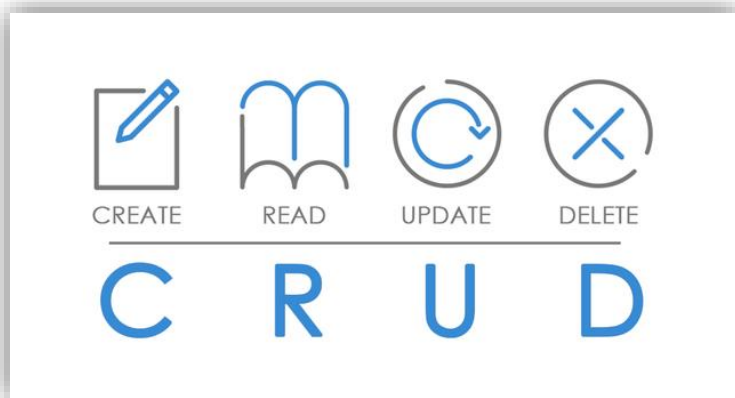
Para poder poner en uso estas API Rest, y antes de utilizar cualquier framework que las implemente, es necesario desarrollar una lógica de aplicación y comportamiento base. Sin este, no habría función alguna para implementar.

Como se ha explicado en otros puntos, nuestra forma de trabajar se basa en un controlador que gestiona la lógica de los modelos y se las hace llegar al cliente, pero ¿qué ocurre entre el controlador y el modelo?



### 2.3.2 REPOSITORIOS

Un repositorio es una interfaz dentro del modelo MVC (en nuestro caso) que se encuentra más cercano a la base de datos. Es la parte de la aplicación encargada de realizar las operaciones a las que se llama en los diferentes servicios, con acción directa en nuestra base de datos, hay muchas metodologías distintas para realizarlas, pero la que seguimos se conoce como CRUD (CREATE-READ-UPDATE-DELETE).



En un repositorio que implementa una interfaz CRUD, declaramos métodos que siguen la estructura de crear, leer, actualizar o borrar, donde cada método tiene una función muy concreta, y todo lo que se salga de esa tarea, se atribuye al nacimiento de un nuevo método.

```
interface IUsersRepository {  
    suspend fun findAll(token: String): UserDataDTO  
    suspend fun findById(token: String, id: String): UserResponseDTO  
    suspend fun findMe(token: String, id: String): UserResponseDTO  
    suspend fun isActive(username: String): Boolean  
    suspend fun update(token: String, id: String, entity: UserUpdateDTO): UserResponseDTO  
    suspend fun updateCredits(token: String, id: String, creditsAmount: Int): UserResponseDTO  
    suspend fun updateCreditsMe(token: String, id: String, creditsAmount: Int): UserResponseDTO  
    suspend fun updateActive(token: String, id: String, active: Boolean): UserResponseDTO  
    suspend fun delete(token: String, id: String)  
    suspend fun me(token: String, entity: UserUpdateDTO): UserResponseDTO  
    suspend fun login(entity: UserLoginDTO): UserTokenDTO  
    suspend fun register(entity: UserRegisterDTO): UserTokenDTO  
    suspend fun downloadFile(uuid: String): File  
    suspend fun uploadFile(token: String, file: MultipartBody.Part): Call<UserPhotoDTO>  
}
```

El controlador no llama directamente a los repositorios, es una mala práctica que haya una relación tan estrecha entre estos, no es recomendable y puede provocar problemas a futuro, pero existe una solución sencilla basada en la estructura y en la delegación de tareas. Esta delegación viene otorgada a los servicios, encargados de recibir la llamada de los controladores, tratar todo lo necesario, y comunicarse con los repositorios.

### 2.3.3 SERVICIOS

Un servicio en nuestro contexto de aplicación son esos paquetes que engloban las clases encargadas de tratar los datos, manipularlos, realizar operaciones lógicas o seguir pasos previos antes de llegar a la capa más cercana a la base de datos. Los servicios reciben los datos directamente del controlador, depurados si es posible y con un nivel de contraste, pero no están listos para introducirse en la base de datos, aquí ocurre la lógica de las funcionalidades de la aplicación.

Otro papel fundamental de los servicios es lanzar excepciones en caso de errores que los controladores reciban, indicándoles así el tipo de respuesta que deben dar al usuario al consumir la API.

```
override suspend fun updateSpace(space: Space, id: String): Space {
    val spaceOriginal = repo.findById(UUID.fromString(id))
    val spaceUpdated = spaceOriginal.copy(
        name = space.name,
        description = space.description,
        image = space.image,
        isReservable = space.isReservable,
        price = space.price,
        requiresAuthorization = space.requiresAuthorization,
        authorizedRoles = space.authorizedRoles,
        bookingWindow = space.bookingWindow
    )
    return repo.update(spaceUpdated)
}
```

En algunos ejemplos como el de continuación, apreciamos un servicio que recibe directamente un objeto por parámetros, luego se trata haciendo todo tipo de comprobaciones, y tras terminar sus tareas, llama a los repositorios necesarios para almacenar correctamente estos nuevos datos (o simplemente gestionarlos).

Existen otros tipos de servicio que no solo se ocupan del almacén de datos simple, sino que entran en juego otros conceptos como es el caché, explicado previamente, o el storage, para almacenar las imágenes a las que recurren nuestros usuarios en el transcurso del uso de la aplicación.

No vamos a entrar de nuevo a explicar cómo funciona la cache, simplemente dejaremos un ejemplo visual de como accede la aplicación a la misma, mejorando significativamente el consumo de tiempo de las solicitudes, pero si en cómo funciona por dentro el servicio de storage.

```
@Single
@Named("SpaceCacheImpl")
class SpaceCacheImpl : SpaceCache {
    override val hasRefreshAllCacheJob: Boolean = true
    override val refreshTime = 60 * 60 * 1000L

    override val cache = Cache.Builder()
        .expireAfterAccess(60.minutes)
        .build<UUID, Space>()
}
```

```
override suspend fun findAll(): List<Space> {  
    return if (!cache.hasRefreshAllCacheJob || cache.cache.asMap().isEmpty()) {  
        repository.findAll()  
    } else {  
        cache.cache.asMap().values.toList()  
    }  
}
```

En este ejemplo, podemos observar el funcionamiento de la cache, al recibir una llamada al repositorio, inicialmente se comprueba si existen los datos en la cache, si existen, presentamos esos datos, si no, realizamos las solicitudes necesarias, esta dinámica mejora sin duda el tiempo de respuesta del api, llegando a tiempos de menos de 10 ms.

- **Storage:** el servicio de storage implementa una interfaz que centra sus métodos en la gestión de archivos, en este caso, imágenes, para el posterior uso de estas en la aplicación final.

```
interface StorageService {  
    fun getConfig(): StorageConfig  
    fun initStorageDirectory()  
    suspend fun saveFile(fileName: String, fileBytes: ByteArray): Map<String, String>  
    suspend fun saveFile(fileName: String, fileBytes: ByteReadChannel): Map<String, String>  
    suspend fun getFile(fileName: String): File  
    suspend fun deleteFile(fileName: String)  
}
```

El servicio de storage trabaja con paquetes y librerías de archivos para poder almacenar en nuestro proyecto una secuencia de imágenes a las que luego poder otorgarles un fácil acceso.

```
override suspend fun getFile(fileName: String): File = withContext(Dispatchers.IO) {  
    var resourceStream = getResourceAsStream("uploads/$fileName")  
    if (resourceStream == null) {  
        resourceStream = getResourceAsStream("placeholder.png")  
        val imagePlaceholder: BufferedImage = ImageIO.read(resourceStream)  
        val outputFile = Files.createTempFile("temp", "").toFile()  
        ImageIO.write(imagePlaceholder, "", outputFile)  
        return@withContext outputFile  
    } else {  
        val imagePlaceholder: BufferedImage = ImageIO.read(resourceStream)  
        val outputFile = Files.createTempFile("temp", "").toFile()  
        ImageIO.write(imagePlaceholder, "", outputFile)  
        return@withContext outputFile  
    }  
}  
  
override suspend fun deleteFile(fileName: String): Unit = withContext(Dispatchers.IO) {  
    val file = File("${uploadsPath}/${fileName}")  
    if (!file.exists()) {  
        throw StorageException.FileNotFound("No se ha encontrado el fichero: $fileName")  
    } else {  
        file.delete()  
    }  
}
```

En esta imagen vemos el código de lo que son dos funciones básicas dentro de nuestro servicio, una para eliminar imágenes y otra para eliminarlas, recibiendo cómo referencia el identificador de dicha imagen.

*Antes de hablar de controladores entran en juego en todo momento en nuestra aplicación paquetes de clases independientes para configuración, utilidades, recursos, etc. Que sirven para mejorar el funcionamiento de esta. En esta explicación no vamos a definir cómo funciona cada uno de ellos, ni daremos una explicación, pero, en resumen, sirven para cumplir las siguientes tareas:*

- *Serialización de clases y tipos de datos para la manipulación de estos.*
- *Ficheros de configuración que definen el empleo de parámetros, estándares y funcionamiento de elementos dentro de nuestra aplicación.*
- *Paquete de plugin, donde instalamos, declaramos y definimos todos los componentes que conforman nuestro proyecto.*
- *Paquete de excepciones donde se declara el tipo de estas y sus diferentes clases asociadas (explicado en controladores).*
- *Paquete de objetos de transferencia de datos, de los que hablamos anteriormente, mencionando su utilidad a la hora de mapear los datos de una forma más sencilla para su transporte, lectura y procesado.*

Un ejemplo visual de configuración muy importante dentro de nuestro proyecto es el uso de Spring Schedule, del framework de Spring Boot, esta configuración permite resetear los créditos de un usuario a principios de cada mes, manteniendo así la microeconomía interna de la aplicación.

```
@Configuration
@EnableScheduling
class ScheduleConfiguration {
}
```

```
@Component
class ScheduleService constructor(
    @Autowired
    val userService: UserService){

    @Scheduled(cron = "0 0 1 1 * ?")
    fun poner20creditosAllUsers() {
        println("${LocalDateTime.now()} INFO --- [Schedule credits function] ScheduleService : Poniendo 20 creditos a todos los usuarios...")
        userService.poner20creditosAllUsers();
        println("${LocalDateTime.now()} INFO --- [Schedule credits function] ScheduleService : Operacion completada")
    }
}
```

El siguiente paso una vez tenemos definidos nuestros servicios es avanzar al siguiente paso, como materializar esos servicios en gestiones iniciadas por peticiones por parte de la API. Este dilema tiene una solución sencilla siguiendo nuestra arquitectura, dependiendo del framework el paquete es llamado rutas, es llamado controladores, pero, para nosotros, no deja de ser un grupo de clases que agrupan las funciones de control de nuestra aplicación.

### 2.3.4 CONTROLADORES

Los controladores, como su nombre describen, son agrupaciones de clases en las que juntamos todas las funciones de gestión, administración y control de la aplicación, es el punto de partida inicial entre la conexión con el usuario (mediante la interfaz) y el resto de la aplicación.

En los controladores, definimos las rutas (recomendable usando una ruta base de la que bebe la de cada método) y dentro se verifican las limitaciones necesarias para llamar a las capas internas, estando presente la importancia de escuchar errores para mostrar el código pertinente e informar al usuario.

```
post {
    try {
        val originalToken = call.principal<JWTPrincipal>()!!
        val token = tokenService.generateToken(originalToken)
        val userRole = originalToken.payload.getClaim("role").toString()
        val subject = originalToken.payload.subject

        val entity = call.receive<BookingCreateDTO>()
        val user = userRepository.findById("Bearer $token", subject)
        val space = spaceRepository.findById("Bearer $token", entity.spaceId)

        if(!userRole.contains("ADMINISTRATOR")){
            require(subject == entity.userId){ "No se puede realizar la reserva a nombre de otra persona" }
            require(user.credits >= space.price) { "No tienes créditos suficientes para realizar la reserva" }
            userRepository.updateCreditsMe("Bearer $token", subject, space.price)

            require(LocalDate.parse(entity.startTime).isAfter( LocalDate.now()))
            ("No se ha podido guardar la reserva fecha introducida es anterior a la actual.")
            require(ChronoUnit.DAYS.between(LocalDate.now(), LocalDate.parse(entity.startTime.split("T")[0])) <= space.bookingWindow)
            ("No se puede reservar con tanta anterioridad.")
            require(bookingsRepository.findByTime("Bearer $token", entity.spaceId, entity.startTime.split("T")[0])
                .data
                .filter{it -> it.startTime.split("T")[1].split(":")[0] == entity.startTime.split("T")[1].split(":")[0]}
                .isEmpty()
            )
            ("Franja horaria no disponible.")
        }
    }
}
```

En esta imagen, podemos observar con detenimiento la implementación de diferentes requerimientos basados en los requisitos del cliente, que controlan todo lo necesario para garantizar la seguridad lógica de las operaciones y funciones. Si estos requerimientos son aceptados, el controlador admite que el servicio continúe con sus labores en las capas inferiores.

```
} catch (e: BookingNotFoundException) {
    println("Error: ${e.message}")
    call.respond(HttpStatusCode.NotFound, "${e.message}")
} catch (e: BookingBadRequestException) {
    println("Error: ${e.message}")
    call.respond(HttpStatusCode.BadRequest, "${e.message}")
} catch (e: BookingInternalErrorException) {
    println("Error: ${e.message}")
    call.respond(HttpStatusCode.InternalServerError, "${e.message}")
} catch (e: IllegalArgumentException) {
    println("Error: ${e.message}")
    call.respond(HttpStatusCode.BadRequest, "${e.message}")
} catch (e: SpaceNotFoundException) {
    println("Error: ${e.message}")
    call.respond(HttpStatusCode.NotFound, "${e.message}")
} catch (e: UserNotFoundException) {
    println("Error: ${e.message}")
    call.respond(HttpStatusCode.NotFound, "${e.message}")
}
}
```

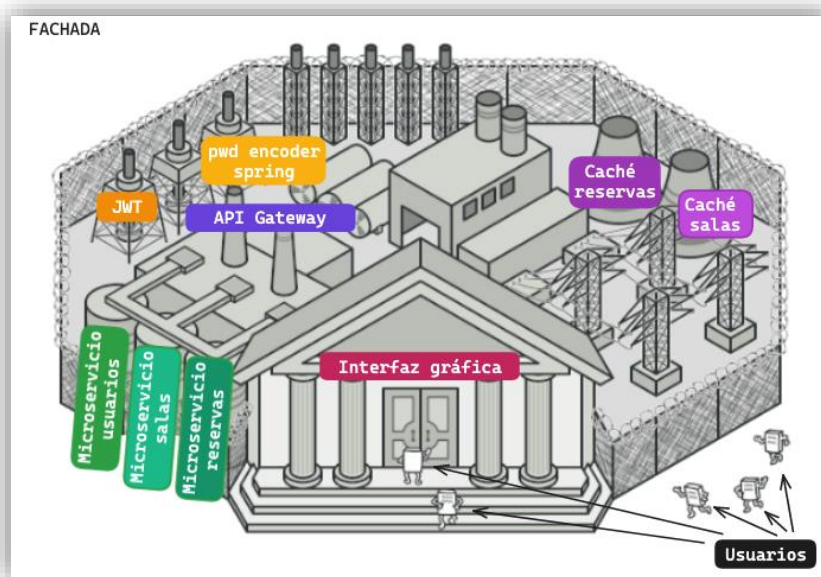
Cuando un controlador reconoce un error, emite una respuesta con el código determinado y el mensaje de error. Es muy importante tener presentes todos los tipos de código de respuesta, ya que son el pilar fundamental de la comunicación con la API.



Como aparte a este seguimiento, nos parece interesante mencionar el uso de diferentes patrones de diseño estudiados durante el curso y destacables en nuestra aplicación, hemos hecho una selección de los que nos han parecido más interesantes.

### 2.3.5 PATRONES DE DISEÑO

#### FACHADA



Surge ante un problema bien definido por el portal web de **Refactoring Guru**:

*“Imagina que debes lograr que tu código trabaje con un amplio grupo de objetos que pertenecen a una sofisticada biblioteca o framework. Normalmente, debes inicializar todos esos objetos, llevar un registro de las dependencias, ejecutar los métodos en el orden correcto y así sucesivamente.” (“Facade - refactoring.guru”)*

*Como resultado, la lógica de negocio de tus clases se vería estrechamente acoplada a los detalles de implementación de las clases de terceros, haciéndola difícil de comprender y mantener.” (“Facade - refactoring.guru”)*

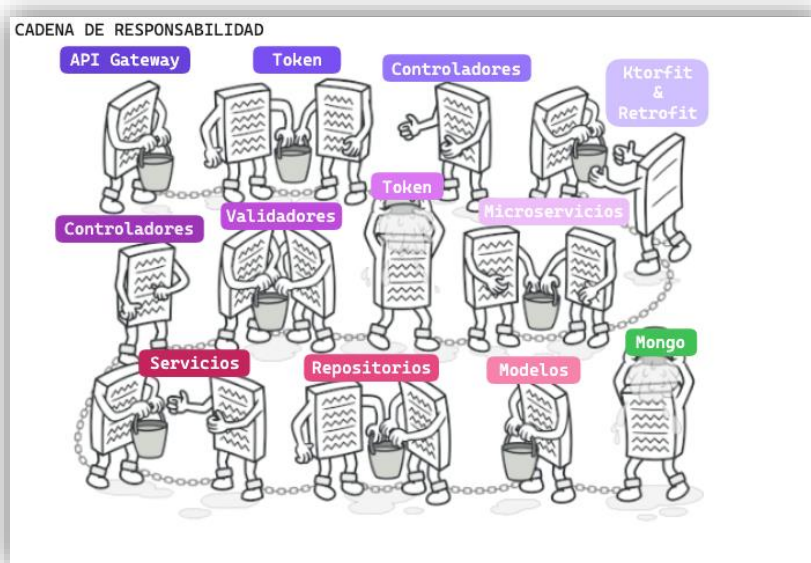
La solución, una **fachada**:

*“Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles. Una fachada puede proporcionar una funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente importantes para los clientes. (“Resumen - dds 1er parcial - PRIMER PARCIAL - Studocu”)*

*Tener una fachada es útil cuando integras tu aplicación con una biblioteca sofisticada con decenas de funciones, de la que solo necesitas una pequeña parte. (“Facade - Refactoring and Design Patterns”)*

*Por ejemplo, una aplicación que sube breves vídeos divertidos de gatos a las redes sociales podría potencialmente utilizar una biblioteca de conversión de vídeo profesional. (“Facade - Refactoring and Design Patterns”) Sin embargo, lo único que necesita en realidad es una clase con el método simple codificar (nombreDelArchivo, formato). Una vez que crees dicha clase y la conectes con la biblioteca de conversión de vídeo, tendrás tu primera fachada.”*

#### CADENA DE RESPONSABILIDADES



Ante el problema que encontramos:

*“Imagina que estás trabajando en un sistema de pedidos online. Quieres restringir el acceso al sistema de forma que únicamente los usuarios autenticados puedan generar pedidos. Además, los usuarios que tengan permisos administrativos deben tener pleno acceso a todos los pedidos. (“Chain of Responsibility - refactoring.guru”)*

*Tras planificar un poco, te das cuenta de que estas comprobaciones deben realizarse secuencialmente. La aplicación puede intentar autenticar a un usuario en el sistema cuando reciba una solicitud que contenga las credenciales del usuario. Sin embargo, si esas*



---

*credenciales no son correctas y la autenticación falla, no hay razón para proceder con otras comprobaciones. ("Chain of Responsibility - refactoring.guru")*

[...]

*El código de las comprobaciones, que ya se veía desordenado, se vuelve más y más abotargado cada vez que añades una nueva función. En ocasiones, un cambio en una comprobación afecta a las demás. Y lo peor de todo es que, cuando intentas reutilizar las comprobaciones para proteger otros componentes del sistema, tienes que duplicar parte del código, ya que esos componentes necesitan parte de las comprobaciones, pero no todas ellas. ("Chain of Responsibility - refactoring.guru")*

*El sistema se vuelve muy difícil de comprender y costoso de mantener. Luchas con el código durante un tiempo hasta que un día decides refactorizarlo todo." ("Chain of Responsibility - refactoring.guru")*

La solución a la que nos aferramos, un sistema **organizado y secuencial**:

*"Al igual que muchos otros patrones de diseño de comportamiento, el Cadena de Responsabilidades se basa en transformar comportamientos particulares en objetos autónomos llamados manejadores. En nuestro caso, cada comprobación debe ponerse dentro de su propia clase con un único método que realice la comprobación. La solicitud, junto con su información, se pasa a este método como argumento. ("Chain of Responsibility - refactoring.guru")*

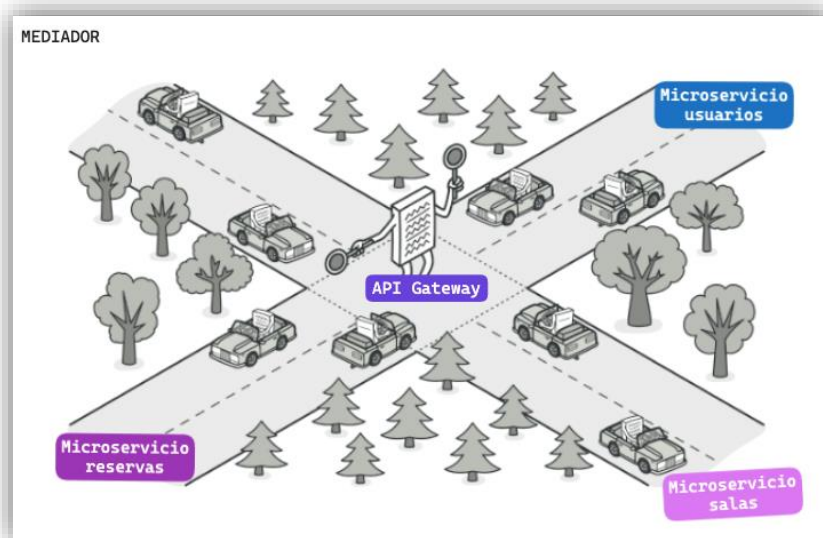
*El patrón sugiere que vincules esos manejadores en una cadena. Cada manejador vinculado tiene un campo para almacenar una referencia al siguiente manejador de la cadena. Además de procesar una solicitud, los manejadores la pasan a lo largo de la cadena. La solicitud viaja por la cadena hasta que todos los manejadores han tenido la oportunidad de procesarla. ("Chain of Responsibility - refactoring.guru")*

*"Y ésta es la mejor parte: un manejador puede decidir no pasar la solicitud más allá por la cadena y detener con ello el procesamiento." ("Chain of Responsibility - refactoring.guru")*

*En nuestro ejemplo de los sistemas de pedidos, un manejador realiza el procesamiento y después decide si pasa la solicitud al siguiente eslabón de la cadena. Asumiendo que la solicitud contiene la información correcta, todos los manejadores pueden ejecutar su comportamiento principal, ya sean comprobaciones de autenticación o almacenamiento en la memoria caché." ("Chain of Responsibility - refactoring.guru")*



## MEDIADOR



El problema que se plantea:

*“Digamos que tienes un diálogo para crear y editar perfiles de cliente. Consiste en varios controles de formulario, como campos de texto, casillas, botones, etc. (“Mediator - Refactoring and Design Patterns”)*

*Relaciones caóticas entre elementos de la interfaz de usuario*

*Algunos de los elementos del formulario pueden interactuar con otros. Por ejemplo, al seleccionar la casilla “tengo un perro” puede aparecer un campo de texto oculto para introducir el nombre del perro. Otro ejemplo es el botón de envío que tiene que validar los valores de todos los campos antes de guardar la información. (“Mediator - Refactoring and Design Patterns”)*

*Los elementos de la UI son interdependientes*

*Al implementar esta lógica directamente dentro del código de los elementos del formulario, haces que las clases de estos elementos sean mucho más difíciles de reutilizar en otros formularios de la aplicación. Por ejemplo, no podrás utilizar la clase de la casilla dentro de otro formulario porque está acoplada al campo de texto del perro. O bien podrás utilizar todas las clases implicadas en representar el formulario de perfil, o no podrás usar ninguna en absoluto.” (“Mediator - Refactoring and Design Patterns”)*

La solución que se propone, un sistema capaz de arreglar los problemas de **comunicación** y **garantizar** la **integridad** de los mensajes:

*“El patrón Mediator sugiere que detengas toda comunicación directa entre los componentes que quieres hacer independientes entre sí. En lugar de ello, estos componentes deberán colaborar indirectamente, invocando un objeto mediador especial que redirija las llamadas a los componentes adecuados. (“Mediator - Refactoring and Design Patterns”) Los componentes dependen solo de una clase mediadora, en vez de estar acoplados a decenas de colegas.*

*En nuestro ejemplo del formulario de edición de perfiles, la propia clase de diálogo puede actuar como mediadora. Lo más probable es que la clase de diálogo conozca ya todas sus subelementos, por lo que ni siquiera será necesario que introduzcas nuevas dependencias en esta clase.” (“Mediator - Refactoring and Design Patterns”)*

### 2.3.6 SEGURIDAD EN EL SERVIDOR

Para garantizar la integridad de los datos de la mejor forma posible bajo nuestro alcance, se ha empleado el uso de pares de clave mediante el concepto de SSL para asegurar que las conexiones de la API General son firmes e íntegras, pero, con este análisis surgen varias preguntas:

- **¿Es necesario SSL en cada microservicio?**

En nuestro caso, SSL no es necesario en nuestros microservicios porque solo se expone hacia afuera la API General, por eso, entre contenedores, no hay influencia que pueda afectar a la integridad de los datos, pero a todas las capas exteriorizadas, es altamente recomendado encriptar sus datos.

- **¿Qué ocurre con los clientes distribuidos?**

Para el cliente web y la ruta del servidor, si consideramos recomendable utilizar un método de encriptación de datos siguiendo la metodología de SSL para garantizar la integridad de estos. Por alcance, hemos visto prioritario centrarnos en la fortificación de la seguridad del servidor y la API internamente, pero si quieren ir más allá, hay muchas páginas y tecnologías que podría blindar la aplicación en todos sus puntos.

Este análisis no sirve de nada si no conocemos a fondo el funcionamiento de SSL. La tecnología SSL/TSL y HTTPS son conceptos que relacionan entre sí la seguridad de la comunicación en línea y la protección de los datos transmitidos a través de Internet. SSL (Secure Sockets Layer) y su sucesor TLS (Transport Layer Security) son protocolos criptográficos que proporcionan un canal seguro de comunicación entre un cliente y un servidor. HTTPS (Hypertext Transfer Protocol Secure) es la implementación de estos protocolos en el protocolo HTTP, utilizado para la transferencia de datos en la web. (“Definición de HTTPS (HTTP Secure) - Alegsa.com.ar”)



Para poder llevar a cabo una conexión segura, recurrimos a la generación de claves y llaveros para ello. Nuestra herramienta favorita es **keytool**, un manejador de certificados que permite crear claves que se almacenan en llaveros y que se declaran en nuestro fichero de configuración para asegurar que, de punto inicial a punto final, la transmisión de los mensajes de nuestra aplicación será seguro y garantizará todos los principios de integridad requeridos.



### 2.3.7 FASE DE TESTING Y ANÁLISIS DE PRUEBAS

En el desarrollo de todos los microservicios, se han implementado paralelamente las tecnologías de test mencionadas en el análisis.

A continuación, mostramos algunos ejemplos de código donde se muestra la implementación de los test unitarios:

```
@Test
fun save() = runTest {
    repository.delete(UUID.fromString(space.uuid))
    val response = repository.save(space)

    assertEquals(space, response)
    assertEquals(space.id, response?.id)
    assertEquals(space.uuid, response?.uuid)
    assertEquals(space.name, response?.name)
    assertEquals(space.image, response?.image)
    assertEquals(space.price, response?.price)
    assertEquals(space.isReservable, response?.isReservable)
    assertEquals(space.requiresAuthorization, response?.requiresAuthorization)
    assertEquals(space.authorizedRoles, response?.authorizedRoles)
    assertEquals(space.bookingWindow, response?.bookingWindow)
}
```

A su vez, mostramos un ejemplo de la implementación de test simulados (Mockito) para comprobar el uso de algunas rutas:





```
@Test
fun findById() = runTest {
    coEvery { spaceRepository.findById(UUID.fromString(space.uuid)) } returns space

    val result = spaceService.getSpaceById(space.uuid)

    assertEquals(space, result)
    assertEquals(space.id, result.id)
    assertEquals(space.uuid, result.uuid)
    assertEquals(space.name, result.name)
    assertEquals(space.image, result.image)
    assertEquals(space.price, result.price)
    assertEquals(space.isReservable, result.isReservable)
    assertEquals(space.requiresAuthorization, result.requiresAuthorization)
    assertEquals(space.authorizedRoles, result.authorizedRoles)
    assertEquals(space.bookingWindow, result.bookingWindow)
}
```

El resultado de estos códigos es un total de más de 100 pruebas pasadas correctamente en un tiempo medio de 5 a 6 segundos.

Tests in 'Microservicio-Reservas.test': 54 total, 54 passed		8.50 s
		<a href="#">Collapse</a>   <a href="#">Expand</a>
	BookingCachedRepositoryTest	1.34 s
	BookingRepositoryImplTest	3.25 s
	BookingRoutesTest	3.40 s
	BookingServiceImplTest	513 ms
Generated by IntelliJ IDEA on 6/10/23, 9:24 PM		

Tests in 'Microservicio-Salas.test': 41 total, 41 passed		7.62 s
		<a href="#">Collapse</a>   <a href="#">Expand</a>
	SpaceCachedRepositoryTest	1.50 s
	SpaceRepositoryImplTest	3.24 s
	SpaceRoutesTest	2.38 s
	SpaceServiceImplTest	501 ms
Generated by IntelliJ IDEA on 6/10/23, 9:22 PM		

Tests in 'Microservicio-Usuarios.test': 39 total, 39 passed		2.68 s
		<a href="#">Collapse</a>   <a href="#">Expand</a>
MicroservicioUsuariosApplicationTests		741 ms
UsersControllerTest		1.29 s
UserServiceTest		652 ms

Generated by IntelliJ IDEA on 6/10/23, 9:17 PM

Llegamos al final de la implementación de nuestro servidor, mediante estas explicaciones, hemos conocido como trabaja por detrás el servidor cuando el usuario interactúa, pero, también es necesario conocer que es lo que hace que el usuario pueda realizar cualquier tipo de interacción con nuestra aplicación.

Entramos en el mundo de Flutter, el mundo del desarrollo consumidor, donde ya no hablamos de implementaciones, sino de recibir respuestas, y mostrarlas de una forma agradable.

Para poder hacer un desarrollo técnicamente fuerte de nuestra aplicación, es importante conocer los conceptos que hacen que esto sea posible en el cliente.

A la hora de trabajar con Flutter, delegamos las funciones de consumición de las APIs a una serie de clases conocidas como Providers.

### 2.3.8 PROVIDERS

Los Providers son clases que agrupan, respetando los principios de responsabilidad los métodos que utilizan librerías de Flutter para consumir la API, posteriormente tratan sus datos agrupándolos en variables previamente definidas, y confeccionan un entorno cómodo para ser posteriormente utilizado por el resto de las partes de nuestra aplicación.

```
class AuthProvider with ChangeNotifier {
  String _token = '';
  String _userId = '';

  Usuario _usuario = Usuario(
    uuid: '',
    name: '',
    username: '',
    email: '',
    password: '',
    avatar: '',
    userRole: [],
    credits: 0,
    isActive: false,
  );

  Usuario get usuario => _usuario;
  String get token => _token;
  String get userId => _userId;

  String baseUrl = 'http://app.iesluisvives.org:1212';
}
```

En esta imagen podemos observar la definición de variables que luego utilizarán el resto de los elementos de nuestra aplicación.

```
Future<Usuario?> login(String username, String password) async {  
    final response = await http.post(  
        Uri.parse('$baseUri/users/login'),  
        headers: {'Content-Type': 'application/json'},  
        body: jsonEncode(  
            {  
                'username': username,  
                'password': password,  
            },  
        ),  
    );  
  
    if (response.statusCode == 200) {  
        final data = jsonDecode(response.body);  
        Usuario usuario = Usuario(  
            uuid: data['user']['uuid'],  
            name: data['user']['name'],  
            username: data['user']['username'],  
            email: data['user']['email'],  
            password: data['user']['password'],  
            avatar: data['user']['avatar'],  
            userRole: List<String>.from(data['user']['userRole']),  
            credits: data['user']['credits'],  
            isActive: data['user']['isActive'],  
        );  
  
        _usuario = usuario;  
        _token = data['token'];  
        _userId = data['user']['uid'];  
        notifyListeners();  
  
        return usuario;  
    } else {  
        notifyListeners();  
        throw Exception(response.body);  
    }  
}
```

En este fragmento de código, observamos el funcionamiento genérico de una función que llama a la API, recibe y procesa sus datos, y en caso positivo, devuelve el dato precise, en caso negativo, lanza una excepción que se gestionará después.

### 2.3.9 PANTALLAS

Dentro de cada pantalla de la aplicación, si es precisa su utilización, recibimos los datos de la API gracias a los Providers, los cuales nos brindan el acceso a sus definiciones, para poder mostrar las mismas al cliente.

```
class _EspaciosScreenState extends State<EspaciosScreen> {  
    final TextEditingController _searchController = TextEditingController();  
    List<Espacio> espaciosFiltrados = [];  
    bool _showSpinner = true;  
  
    @override  
    void initState() {  
        super.initState();  
        final espaciosProvider =  
            Provider.of<EspaciosProvider>(context, listen: false);  
  
        espaciosFiltrados = espaciosProvider.espaciosReservables;  
  
        espaciosProvider  
            .fetchEspaciosByReservable(true)  
            .then((value) => setState(() {  
                espaciosFiltrados = espaciosProvider.espaciosReservables;  
            }));  
  
        Timer(const Duration(seconds: 3), () {  
            setState(() {  
                _showSpinner = false;  
            });  
        });  
    }  
}
```

## 2.4 IMPLANTACIÓN, EMPAQUETADO Y DESPLIEGUE

Desde el primer momento se nos dio las pautas de seguir una metodología que tuviese como objetivo final un sistema basado en elementos propios por eso, en la medida de lo posible, se procura que todos los servicios aplicados estuviesen desplegados de igual forma en servidores que el instituto nos ha ofrecido para esta tarea.

Para dicha función, uno de nuestros pilares a la hora de poder desplegar y hacer pública la aplicación ha sido el uso de Docker.

Antes de explicar cómo ha sido el proceso de desarrollo de cada fase, vemos útil hacer un repaso de los diferentes conceptos que conforman el empaquetado y despliegue de una aplicación como la nuestra.

### DOCKERFILE

Un Docker File es un archivo o documento de texto que incluye unas instrucciones que se necesitan ejecutar para cumplir con el proceso de creación de una nueva imagen.

### CONTENEDOR

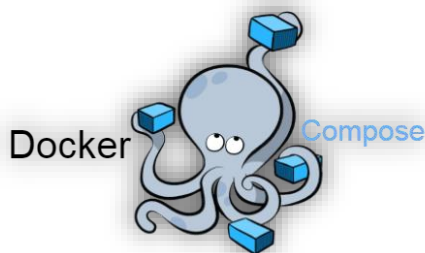
Un contenedor de Docker es un ejecutable ligero e independiente que alberga todo el código, librerías, herramientas y útiles necesarios para hacer funcionar una aplicación, estos tienen su base en las imágenes, elementos explicados en el siguiente punto.

### IMAGEN

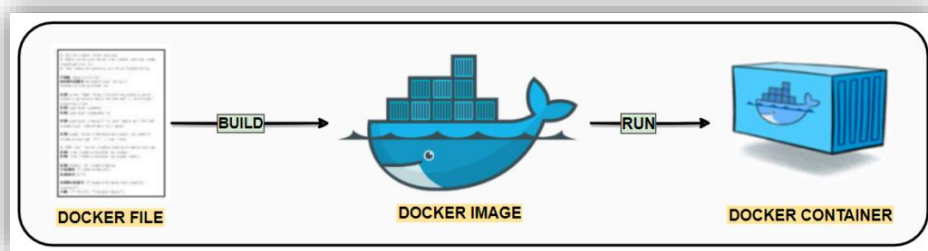
Una imagen de Docker es plantilla de lectura que sirve para definir un contenedor, en ella se incluye el código a ejecutar, incluye todas las definiciones necesarias para que nuestro código funcione.

### DOCKER COMPOSE

Es un archivo de extensión .yaml que sirve para definir y ejecutar contenedores de Docker de forma ordenada y estructurada. Sirve para configurar los servicios de nuestra aplicación y levantarla con un simple comando.



Finalmente, teniendo claros estos conceptos, explicar la forma de trabajar es muy sencilla, para cada servicio tenemos asociada una imagen con la ayuda de un Docker File que configura el mismo y define todos los elementos necesarios, una vez hecho esto, con ayuda de Docker Hub, podemos tener publicadas en internet nuestras imágenes, las cuales descargamos y utilizamos en nuestro Docker Compose para orquestar contenedores que hacen que nuestra aplicación se ejecute.



Para poder resumir de forma correcta todos los pasos a seguir para la implantación de los servicios que conforman la aplicación, paso a paso, vamos a ir deteniéndonos en cada fase para explicar todo lo necesario:

- **Conectividad entre microservicios y base de datos:** creamos un Docker File para cada uno de los microservicios, en este Docker File creamos un archivo JAR con nuestra aplicación, copiamos el directorio y, a su vez, declaramos el puerto interno que usara cada microservicio para comunicarse. Así se vería uno de estos Docker File:

```
FROM gradle:7-jdk17 AS build
COPY --chown=gradle:gradle . /home/gradle/src
WORKDIR /home/gradle/src
RUN gradle buildFatJar --no-daemon

FROM openjdk:17-jdk-slim-buster
EXPOSE 8181
RUN mkdir /app
COPY --from=build /home/gradle/src/build/libs/Microservicio-Reservas-all.jar /app/microservicios-reservas.jar
ENTRYPOINT ["java","-jar","/app/microservicios-reservas.jar"]
```

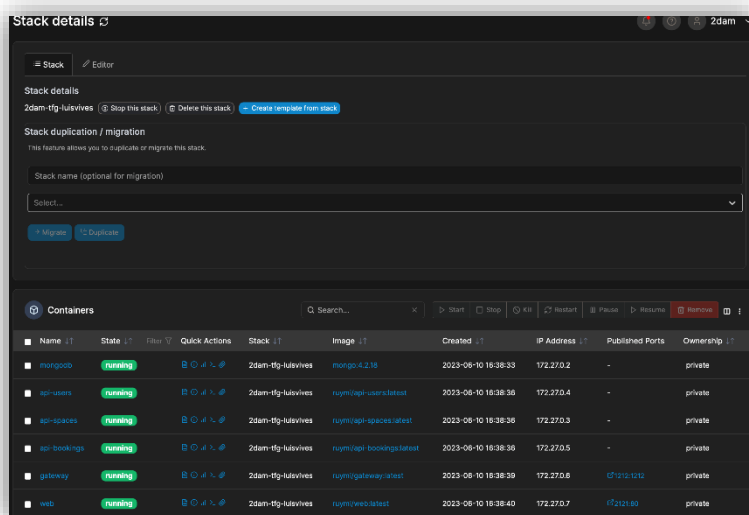
Posterior a esto, lo que hacemos es crear un docker-compose que obtiene todas las imágenes resultantes y, una a una, y de forma ordenada, va ejecutando cada contenedor, teniendo prioridad la base de datos, y, por último, la página web.

```
version: "3.9"
services:
  mongodb:
    image: mongo:4.2.18
    container_name: mongodb
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: admin
      MONGODB_DATABASE: reservas-luisvives
      MONGODB_USER: admin
      MONGODB_PASSWORD: admin
      MONGODB_HOST_NAME: localhost
      MONGODB_PORT: 27017
  api-users:
    build:
      context: ../../Microservicio-Usuarios
      dockerfile: ./docker/Dockerfile
    image: ruymi/api-users:latest
    container_name: api-users
    depends_on:
      - mongodb
  api-spaces:
    build:
      context: ../../Microservicio-Salas
      dockerfile: ./docker/Dockerfile
    image: ruymi/api-spaces:latest
```

*Imagen no completa...*



- **Despliegue de nuestro servidor al propio del instituto:** una vez implementada la capacidad de utilizar un docker-compose para ejecutar todos los contenedores (todos los servicios de la aplicación), el siguiente paso es acceder al portainer proporcionado por el instituto. En este portainer nosotros subimos el docker-compose, que sirve como stack de contenedores los cuales, a partir de este momento, comienzan a funcionar de manera constante sin depender de ninguno de nuestros ordenadores (sino del servidor del instituto).



- **Despliegue del cliente web al servidor propio del instituto:** de forma paralela a la conectividad de los microservicios, también se incluye en ese docker-compose mencionado una nueva imagen que se crea de la misma forma que las demás, pero que tiene una funcionalidad muy distinta, en este caso la función que realiza es la de publicar el cliente web de la aplicación, para que sea accesible fácilmente a través de cualquier navegador, podemos comprobar esto mismo accediendo a la siguiente url.

<http://app.iesluisvives.org:2121>

- **Planteamiento a futuro, despliegue en tienda de aplicaciones:** como planteamiento a futuro, se plantea la necesidad de estudiar cuales son los requisitos de los principales proveedores de aplicaciones móviles, como App Store o Play Store, para que nuestra aplicación se distribuya mediante este medio y pueda ser accesible para todos los alumnos y miembros del centro educativo.

## 2.5 DOCUMENTACIÓN

Para la documentación de la API, se ha utilizado dos tipos de bibliotecas, cada una para una función en concreto.

Para futuros compañeros que puedan heredar el código, tengan labores de desarrollo en él, y necesitan entenderlo para poder aplicarlo, usar un sistema de documentación de la propia API, donde el desarrollador pueda ver las peticiones, los datos requeridos, ejemplos, lo necesario para entender cómo funciona la API sin preguntar a nadie.

Por otro lado, para la documentación pura del código, las cabeceras de los métodos, etc. Se ha pensado en seguir la dinámica de utilizar un sistema que permita acceder fácilmente a la descripción de las funciones, conocer sus parámetros y sus funciones de retorno.

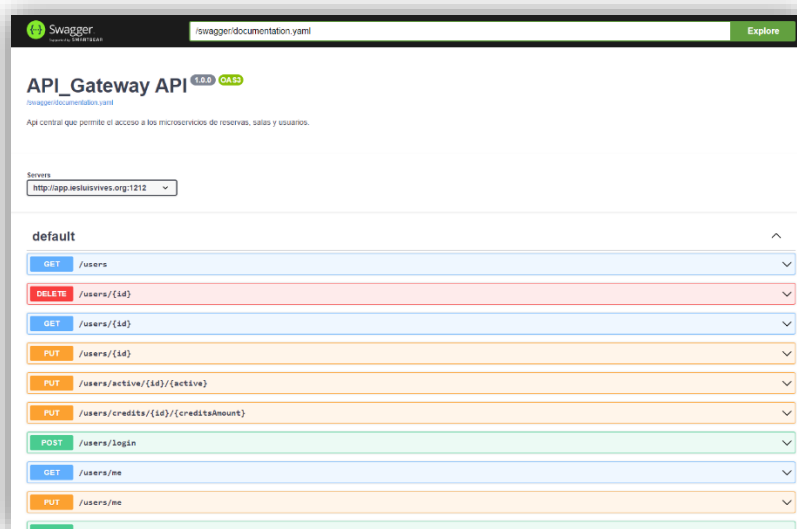
Ante este análisis, se ha decidido utilizar Swagger para la primera tarea, y Java Doc (en este caso KDoc), para la segunda.

### 2.5.1 SWAGGER

Swagger es una librería que habilita en nuestra API una ruta que muestra al usuario los datos de la API desarrollada, recopilando en una única página todas las peticiones, opciones de respuesta con ejemplos, posibles errores, etc. Es una elección muy común en el mercado, funciona muy bien y es fácil de implementar, ya sea mediante su archivo de documentación o con código específico dentro de nuestro proyecto.



Tras implementar Swagger en nuestro proyecto, algunos ejemplos de cómo se ve la descripción e información de las rutas es la siguiente:



Devuelve una lista de todos los usuarios existentes.

Parameters Try it out

No parameters

Responses

Code	Description	Links
200	OK	No links
401	Unauthorized	No links
404	Not Found	No links

Media type:

Controls Accept header:

Example Value | Schema

```

{
  "data": [
    {
      "id": "f6d02ed-fcb3-479e-97ca-33ac1f6a2ff",
      "name": "maria fernanda",
      "username": "mariafernanda",
      "email": "mariafernanda@gmail.com",
      "password": "password12345",
      "avatar": "placeholder.png",
      "role": "user",
      "credits": 20
    }
  ]
}

```

Esta operación no está permitida para los usuarios que no son administradores.

Schemas

- UserResponseDTO >
- UserDataDTO >
- UserUpdateDTO >
- UserLoginDTO >
- UserTokenDTO >
- UserRegisterDTO >
- UserPhotoDTO >
- SpaceDTO >
- SpaceDataDTO >
- SpaceCreateDTO >
- SpaceResponseDTO >
- SpaceUpdateDTO >
- SpacePhotoDTO >
- BookingResponseDTO >

UserResponseDTO >

```

{
  id: string
  name: string
  username: string
  email: string
  password: string
  avatar: string
  role: string
  credits: integer(32)
}

```

Para poder acceder a dicha web con la información, tenemos a nuestra disposición el siguiente link:

<http://app.iesluisvives.org:1212/swagger>

## 2.5.2 KDOC CON DOKKA

Cómo hemos dicho antes, no solo es necesario entender las peticiones, sino también conocer el código, sus puntos fuertes, débiles, y cómo se desarrolla. Es por esto por lo que hemos visto útil utilizar una librería de Java Doc destinada a Kotlin en la que con simples códigos (siguiendo un formato específico), podamos ser capaces de exportar una documentación de nuestros modelos de datos, paquetes y funciones siguiendo un estándar.



Dokka cuenta a su vez con una amplia documentación que el propio equipo de desarrollo de Kotlin ofrece a todo el que quiera aprender a implementarlo en su proyecto, lo que nos ha sido de especial utilidad para nosotros.

## 2.5.3 README

Para hacer más fácil la visualización de información básica acerca del proyecto, se ha incluido en el repositorio principal del proyecto un README.md donde se definen algunos datos de relevancia que podrías interesar a cualquier usuario que de primeras observase el mismo:



## Roles de Usuario

La aplicación cuenta con los siguientes roles de usuario:

- Alumno/a
- Profesor/a
- Administrador/a

## Espacios

Característica	Valor
Nombre	Nombre del espacio
Imagen	Imagen del espacio
Descripción	Descripción del espacio
Precio	Cuánta de créditos que debe pagar el usuario para reservar dicho espacio
Reservable	Sí/No
Ventana temporal (antelación con la que se puede reservar el espacio)	En días
Roles que pueden reservar el espacio	Alumno/a, Profesor/a, Administrador/a
Requiere autorización	Sí/No

## Reservas

Característica	Valor
ID de Usuario	Identificador del usuario que realiza la reserva
Nombre del Usuario	Nombre del usuario que realiza la reserva
ID de Espacio	Identificador del espacio reservado
Nombre del Espacio	Nombre del espacio reservado
Imagen	Imagen del espacio reservado
Fecha de Inicio	Fecha de inicio de la reserva
Fecha de Fin	Fecha de fin de la reserva
Observaciones	Observaciones de la reserva
Estado	Estado de la reserva (pendiente, aceptada, rechazada)

## Usuarios

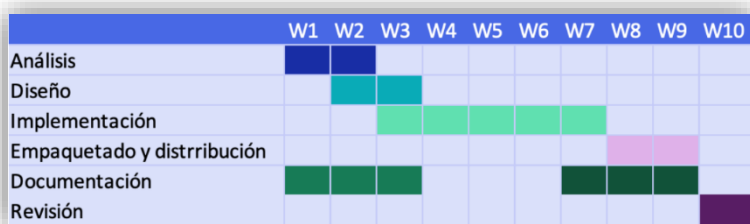
Característica	Valor
Nombre	Nombre del usuario
Nombre de usuario	Nombre de usuario del usuario
Email	Email del usuario
Contraseña	Contraseña del usuario
Avatar	Avatar del usuario
Rol	Rol del usuario (Alumno/a, Profesor/a, Administrador/a)
Créditos	Créditos del usuario disponibles para reservar
Activo	Sí/No



### 3 RESULTADOS Y DISCUSION

Llegados a este punto, y habiendo analizado, descrito y explicado todos los anteriores, llega el momento de estudiar cual ha sido el resultado final y nuestra visión al respecto.

Para ello, nos apoyaremos en el siguiente esquema que se encuentra en el anteproyecto y que nos daba una guía sobre el proceso de trabajo a nivel general:



Siendo sinceros ante el lector de esta memoria, podríamos considerar que, aunque no de forma exacta, se han respetado la gran mayoría de los tiempos estipulados, aunque siempre con altibajos.

- El proceso de análisis duro en tiempo real hasta la primera tutoría fue entonces cuando se acoplaba esa semana con la semana de diseño, tuvimos que decidir cómo enfocar según qué aspectos de la arquitectura y algunas tecnologías.
- El proceso de diseño fue bastante más rápido de lo esperado, debido a que, habiendo tenido un pequeño retraso en la anterior fase, decidimos que la idea que teníamos iba a ser la final, ya que había sido bien estudiada y planteada.
- La implementación empezó siendo inicialmente un proceso sencillo para nosotros, pero solo lo que considerábamos una base. Esta fase rápidamente se tornó a una búsqueda y caza de posibles agujeros y problemas en los que poder aplicar nuestra lógica de negocio y pulir detalles, lo que hace que el tiempo se extienda notoriamente.
- A la vez que íbamos implementando la aplicación, estuvimos tocando un poco de documentación, pero la justa, ya que finalmente nos centramos en dejar la última semana y media para ello.

- De forma paralela al desarrollo de la documentación, se empezó la implementación de un sistema de empaquetado y distribución tedioso, ya que cada cambio realizado debía comprobarse correctamente, probado para todos los casos, etc. Esto se traduce en un gasto de tiempo más notorio que en el planteamiento inicial, pero fácil de llevar.
- Por último, se ha llegado a la fase de revisión, la cual nosotros resumimos coloquialmente como “destrozar nuestra propia aplicación”, teniendo como objetivo seguir buscando fallos maliciosamente, para posteriormente aplicarles una solución, y así fortalecer nuestra aplicación.

Más allá de este análisis, nos gustaría hacer una reflexión sobre las partes que consideramos que han sido más complicadas durante la realización de este proyecto. Dentro de estas complicaciones no solo hablamos del apartado técnico, sino un poco también del temporal:

- Crear una estructura ordenada y concisa para el despliegue de la aplicación.
- Cambiar de fase de la de desarrollo a la de producción, cerrando poco a poco procesos de implementación que considerábamos terminados.
- Discusión en relación con la implementación de funcionalidades más complejas en comparación con otras más básicas.
- Organización y compaginación de nuestro tiempo con el dedicado al desarrollo.

## 4 TRABAJO FUTURO

A continuación, se relata detenidamente nuestra visión en relación con el trabajo futuro respecto al ya realizado en este proyecto, las inquietudes de los compañeros y los intereses del centro.

### 4.1 SEGUNDA PARTE

Este proyecto nació inicialmente para tener dos partes. La primera parte es este proyecto, un sistema de gestión de los espacios comunes del centro con un usuario final claro: todo profesor o alumno que quisiese hacer uso de una sala, pista deportiva, etc.

Como se ha podido ver en los prototipos y producto final, la aplicación no solo se limita a esto, al menos no en apariencia. Existen pantallas y funcionalidades en la aplicación preparadas para ser utilizadas por otros compañeros y poco a poco continuar ampliando la aplicación.

Si hablamos de funcionalidad, nuestros requisitos y los del centro se han cubierto y la aplicación realiza todas las tareas precisadas para esta primera parte, pero según el proceso de desarrollo, entendimos que había muchas partes que podían mejorar e incluso nuevas ideas de implementación que escapaban de nuestro alcance, pero que podrían ser una realidad en el corto plazo. No hablamos solo de concertar servicios, o de utilizar la aplicación como un sistema de notificaciones, funciones que están a la espera de ser implementadas en algún momento, sino de ir mejorando de forma continua el producto ya existente, ya sea en capacidad de personalización, mejoras en la integración con plataformas del centro (incluso unificación de estas), mejoras de funcionalidades básicas, etc.

---

## 4.2 MEJORA Y DESARROLLO CONTINUO

Tras terminar el proyecto, consideramos que la elección de tecnologías fue la adecuada, y pese a posibles complicaciones, o en el punto opuesto, posibles facilidades, creemos que ninguna tecnología se ajusta completamente a este proyecto porque todas son bienvenidas y ninguna se excluye; por eso invitamos a futuros compañeros a ampliar el camino de lo descrito.

## 4.3 ¿QUÉ SE PUEDE MEJORAR?

Conocido el interés de que este proyecto siga adelante, vemos adecuado dejar por escrito una serie de posibles implementaciones futuras:

- Mejoras en la visualización de los listados para el BackOffice (más opciones de filtrado y visualizado).
- Mejora en las capacidades de personalización de la aplicación para el propio usuario.
- Mejoras en la interfaz gráfica con el objetivo de hacer la aplicación más accesible (pensando en aplicaciones adaptadas para el uso de personas con discapacidad).
- Escalabilidad lógica de la aplicación en relación con futuras funciones.
- Adición de un sistema de notificaciones que acompañe a las funciones.
- Mejoras en el acceso seguro a la plataforma con sistemas del propio centro.
- Implementación del concepto de tiempo real si hay muchos usuarios activos de forma simultánea.
- Mejoras en la estructura de widgets de la aplicación de Flutter.
- Uso de clases “Result” para obtener y procesar excepciones.

---

## 5 CONCLUSIONES

Este proyecto nació inicialmente bajo un pensamiento inconcluso sobre el trabajo final.

Inicialmente no estábamos seguros de lo que queríamos intentar desarrollar, pero en cuanto se propuso la posibilidad de crear algo que fuese útil para el centro, todo el equipo creyó que ese era el camino que podíamos seguir.

En primer momento no nos resultó difícil abordar el mismo, teníamos unas buenas pautas, veíamos claras las tecnologías a utilizar, pero poco a poco nos dimos cuenta de que algunas de ellas no iban a ser tan fructíferas como otras, es por esto por lo que tuvimos que estudiar nuevamente nuestras diferentes opciones.

Durante el transcurso de este módulo hemos aprendido tanto tecnologías nuevas como mejorado notoriamente el uso de muchas otras que ya conocíamos.



Hemos vuelto a ver lo útiles que son las arquitecturas basadas en microservicios, las APIs RESTFUL, lo cómodas de implementar y la variedad de nuevas funcionalidades que se les pueden añadir.

Hemos vuelto a encontrarnos con el mundo de Docker y sus complicaciones, solventando las mismas de una forma muy buena, bajo nuestro criterio. No solo hemos aprendido a levantar un contenedor y a ejecutarlo, sino a seguir todo un proceso de desarrollo y despliegue que va desde cero hasta un producto final.

Hemos conocido y nos hemos adentrado en la programación enfocada al frontend, al cliente. Partiendo de un conocimiento casi nulo, a excepción de unas semanas a final de curso. Esto nos ha servido para tener que buscar y aplicar recursos enteramente por nuestros propios medios, utilizar internet, artículos, incluso preguntar a nuestros profesores o compañeros de prácticas. Todo esto para crear un producto final funcional (y ciertamente agradable) que demuestra que cualquier con un poco de conocimiento y las bases asentadas, puede hacer proyectos muy completos.

Como conclusión, nos quedamos pensando que nos hemos demostrado, y el tiempo dirá si mucha gente, que somos capaces de crear y asumir proyectos con complejidad discutible. Que de este proyecto salimos siendo mejores programadores en todos los ámbitos trabajados, y que este grado si nos ha servido para avanzar como personas y como profesionales.

Nos quedamos con esta frase:

*“El arte desafía a la tecnología, y la tecnología inspira al arte.” (“¿Cuál es el impacto cultural de la tecnología?”)*

---

## 6 BIBLIOGRAFÍA

---

*API Documentation & Design Tools for Teams.* (n.d.). Swagger.io. Retrieved June 10, 2023,

<https://swagger.io/>

*Build apps for any screen.* (n.d.). Flutter.dev. Retrieved June 10, 2023,

<https://flutter.dev/>

*Cache4k: In-memory Cache for Kotlin Multiplatform.* (n.d.).

*Dart programming language.* (n.d.). Dart.dev. Retrieved June 10, 2023,

<https://dart.dev/>

Dock, M. (2022, May 10). *Docker: Accelerated, containerized application development.* Docker.

<https://www.docker.com/>

*dokka: API documentation engine for Kotlin.* (n.d.).

*Home.* (n.d.). Home. Retrieved June 10, 2023,

<https://spring.io/>

*JSON web tokens - jwt.io.* (n.d.). Jwt.io; Auth0. Retrieved June 10, 2023,

<https://jwt.io/>

*JUnit 5.* (n.d.). Junit.org. Retrieved June 10, 2023,

<https://junit.org/junit5/>

*Koin - the kotlin dependency injection framework.* (n.d.). Insert-Koin.io. Retrieved June 10, 2023,

<https://insert-koin.io/>

*Kotlin programming language.* (n.d.). Kotlin. Retrieved June 10, 2023,

<https://kotlinlang.org/>

*Ktor: Build asynchronous servers and clients in kotlin.* (n.d.). Ktor Framework. Retrieved June 10, 2023,

<https://ktor.io/>

*Ktorfit.* (n.d.). Github.io. Retrieved June 10, 2023,

<https://foso.github.io/Ktorfit/>

---

*Mockito framework site.* (n.d.). Mockito.org. Retrieved June 10, 2023,

<https://site.mockito.org/>

*MongoDB atlas: Cloud document database.* (n.d.). MongoDB. Retrieved June 10, 2023,

[https://www.mongodb.com/cloud/atlas/lp/try4?utm\\_source=google&utm\\_campaign=search\\_gs\\_pl\\_evergreen\\_atlas\\_core\\_prosp-brand\\_gic-null\\_emea-es\\_ps-all\\_desktop\\_eng\\_lead&utm\\_term=mongodb&utm\\_medium=cpc\\_paid\\_search&utm\\_ad=e&utm\\_ad\\_campaign\\_id=12212624563&adgroup=115749706983&cq\\_cmp=12212624563&gad=1&gclid=CjwKCAjwvpCkBhB4EiwAujULMqa1t\\_RekDoU1FU7SIFak30TZBIDM2vODgm2mNRzdcV6JqfSPJTRPxOCTjiQAvD\\_BwE](https://www.mongodb.com/cloud/atlas/lp/try4?utm_source=google&utm_campaign=search_gs_pl_evergreen_atlas_core_prosp-brand_gic-null_emea-es_ps-all_desktop_eng_lead&utm_term=mongodb&utm_medium=cpc_paid_search&utm_ad=e&utm_ad_campaign_id=12212624563&adgroup=115749706983&cq_cmp=12212624563&gad=1&gclid=CjwKCAjwvpCkBhB4EiwAujULMqa1t_RekDoU1FU7SIFak30TZBIDM2vODgm2mNRzdcV6JqfSPJTRPxOCTjiQAvD_BwE)

*Qué es MVC.* (2014, January 2). Desarrolloweb.com.

<https://desarrolloweb.com/articulos/que-es-mvc.html>

*¿Qué es una API de REST?* (n.d.). Redhat.com. Retrieved June 10, 2023,

<https://www.redhat.com/es/topics/api/what-is-a-rest-api>

*¿Qué son SSL, TLS y HTTPS?* (n.d.). Digicert.com. Retrieved June 10, 2023,

<https://www.digicert.com/es/what-is-ssl-tls-and-https>

*Refactorización y patrones de diseño.* (n.d.). Refactoring.guru. Retrieved June 10, 2023,

<https://refactoring.guru/es>

*Retrofit.* (n.d.). Github.io. Retrieved June 10, 2023,

<https://square.github.io/retrofit/>

*Uso de Docker Compose para implementar varios contenedores.* (n.d.). Microsoft.com. Retrieved June 10, 2023,

<https://learn.microsoft.com/es-es/azure/cognitive-services/containers/docker-compose-recipe>

(N.d.-a). Amazon.com. Retrieved June 10, 2023,

<https://aws.amazon.com/es/microservices/>

(N.d.-b). Baeldung.com. Retrieved June 10, 2023,

<https://www.baeldung.com/spring-scheduled-tasks>

---

## 7 ANEXOS

---

A continuación, se dejan ver los enlaces a varios anexos de interés que acompañan a nuestro proyecto.

- I. **ANEXO 1: LINK DIRECTO AL PROTOTIPO DE LA APLICACIÓN (GITHUB).**  
[https://github.com/RuyMi/tfg-gestion-espacios/blob/main/metadata/Presentacion\\_Tribunal.pdf](https://github.com/RuyMi/tfg-gestion-espacios/blob/main/metadata/Presentacion_Tribunal.pdf)
- II. **ANEXO 2: LINK DIRECTO AL ANTEPROYECTO (GITHUB).**  
<https://github.com/RuyMi/tfg-gestion-espacios/blob/main/metadata/Anteproyecto.pdf>
- III. **ANEXO 3: FICHEROS JSON DE POSTMAN (GITHUB).**  
<https://github.com/RuyMi/tfg-gestion-espacios/blob/main/metadata/postman>
- IV. **ANEXO 4: PDF CON EL PROTOTIPO.**  
[https://github.com/RuyMi/tfg-gestion-espacios/blob/main/metadata/Prototipo\\_Interfaz.pdf](https://github.com/RuyMi/tfg-gestion-espacios/blob/main/metadata/Prototipo_Interfaz.pdf)