



IES JULIO VERNE
BARGAS
Departamento de Informática

Ciclo Formativo de Grado Superior de Desarrollo de Aplicaciones
[Web/Multiplataforma]

APP CLÍNICA VETERINARIA

Autor: DAVID MENÉNDEZ RODRÍGUEZ
Tutor académico: BLANCA ISABEL BRASAL

Mayo, 2025

|

Capítulo 1 Introducción.....	11
1.1. Motivación.....	11
1.2. Abreviaturas y acrónimos.....	12
Capítulo 2 Objetivos.....	12
2.1. Objetivo general.....	13
2.2. Objetivos específicos.....	13
Capítulo 3 Metodología.....	15
3.1. Guía rápida de las metodologías de desarrollo del software.....	15
3.1.1. Proceso de desarrollo de software.....	15
3.1.2. Metodología de Desarrollo software.....	22
3.2. Proceso de <i>testing</i>	24
3.3. Evolución.....	25
3.4. Marco tecnológico.....	26
3.4.1. Herramientas CASE (Computer Aided Software Engineering).....	26
3.4.2. IDE (Integrated Development Environment).....	26
3.4.3. Depuración.....	26
3.4.4. Repositorios y control de versiones.....	27
3.4.5. Documentación.....	27
3.4.6. Gestión y planificación de proyectos.....	27
Capítulo 4 Resultados.....	28
4.1. Iteración 0.....	28

.....	31
4.1.2. Estudio de viabilidad.....	32
4.1.3. Plan de proyecto.....	35
4.2. Iteración 1 - Fase I: Análisis, Diseño Inicial y Prototipado.....	36
4.3. Iteración 2 - Fase II: Diseño de backend básico.....	38
4.4. Iteración 3 - Fase III: Diseño base Frontend.....	47
4.5. Iteración 4 - Fase IV: Implementación de la Autenticación y Gestión de Usuarios.....	56
4.6. Iteración 5 - Fase V: Gestión de Mascotas y Citas.....	59
4.7. Iteración 6 - Fase VI: Desarrollo de la Tienda y Carrito de Compras.....	65
4.8. Iteración 7 - Fase VII: Funcionalidades de Perfil de Usuario.....	69
4.10. Iteración 9 - Fase IX: Últimas Pruebas, Despliegue y Mantenimiento.....	76
Capítulo 5 Conclusiones.....	80
5.1. Revisión de los objetivos.....	80
5.2. Trabajos Futuros.....	83
Capítulo 6: Bibliografía.....	84

Índice de figuras

Figura 1.....	16
Figura 2: Arquitectura Cliente-Servidor.....	16
Figura 3.....	17
Figura 4: Diagrama de clases.....	17
Figura 5: Diagrama casos de uso.....	18
Figura 6: Diagrama casos de uso.....	31
Figura 7: Diagrama de Gantt.....	35
Figura 8: Diagrama de clases.....	36
Figura 9: Arquitectura Cliente-Servidor.....	37
Figura 10: Diagrama casos de uso.....	38
Figura 11: Creacion BBDD.....	39
Figura 12: Conexion BBDD.....	40
Figura 13: GeneralExceptionAdvice.....	41
Figura 14: Servicio.java.....	42
Figura 15: CreateServicioDto.....	42
Figura 16: GetServicioDto.....	44
Figura 17: ServicioMapper.....	45
Figura 18: ServicioRepository.....	45
Figura 19: ServicioService.....	46
Figura 20: Login.....	47
Figura 21: SignUp.....	47
Figura 22: Container Login.....	48
Figura 23: codigo CircleMenuButton.....	49
Figura 24: codigo positionedCircleCustom.....	49
Figura 25:Codigo MenuInicio.....	49
Figura 26: MenuInicio.....	50
Figura 27:Codigo Calendario.....	51
Figura 28: ReservarCita.....	52
Figura 29: Mis Citas.....	53
Figura 30: Mis mascotas.....	54

Figura 31: Registro Mascotas.....	54
Figura 32: Tienda.....	55
Figura 33: Productos.....	55
Figura 34: Configuración Security.....	56
Figura 35: Autorización Básica.....	56
Figura 36: registrar Usuario.....	57
Figura 37: implementación endpoint login.....	58
Figura 38: fetch mascotasUsuario.....	59
Figura 39: endpoints subirImagen.....	60
Figura 40: CrearMascota.....	60
Figura 41: application.properties.....	61
Figura 42: Implementación POST crear cita.....	62
Figura 43: Obtener Servicios.....	63
Figura 44: Horas ocupadas.....	63
Figura 45: implementación mis citas.....	64
Figura 46: código filtros.....	65
Figura 47: Producto Repository.....	66
Figura 48: código añadir carrito.....	67
Figura 49: cesta.....	68
Figura 50: perfil usuario.....	69
Figura 51: Actualizar Usuario.....	70
Figura 52: Cita Repository.....	71
Figura 53: Cerrar sesión.....	72
Figura 54: citas confirmadas.....	73
Figura 55: citas próximas.....	73
Figura 56: navegación rol.....	74
Figura 57: post productos.....	75
Figura 58: basic auth frontend.....	75
Figura 59: Regex.....	76
Figura 60: TextFormField.....	77
Figura 61: validación regex.....	77
Figura 62: validación encriptado.....	78
Figura 63: service usuario.....	78

Índice de tablas

Capítulo 1 Introducción

Este proyecto propone el **desarrollo de una aplicación móvil destinada a una clínica veterinaria**, en este caso una PYME, con el **objetivo de facilitar** principalmente la **gestión de citas, compra de productos especializados y atención al cliente**, mejorando así la experiencia tanto de los clientes como del personal de la clínica, adaptándose a las nuevas demandas del entorno digital.

1.1. Motivación

En los últimos años, la digitalización ha transformado la gestión de servicios en diferentes sectores. Las **PYMES**, como la que motiva este proyecto, suelen enfrentar dificultades en la organización de citas y la gestión de ventas de productos, recurriendo en **muchos casos a métodos manuales que resultan ineficientes y propensos a errores**. Esta realidad genera incomodidades tanto para los profesionales como para los clientes, quienes demandan procesos más ágiles, seguros y personalizados.

En mi caso, me surge la motivación por orientar la aplicación a la gestión de una pequeña clínica veterinaria como la de mi padre, algunos de estos negocios siguen sin digitalizarse usando páginas web desactualizadas, softwares antiguos, gestionando la agenda de manera manual y cogiendo las citas por teléfono, lo que resulta ineficiente, e incluso algunos ni siquiera se encuentran en algún portal de reseñas de alto impacto como Google Maps que les daría muchísima más visibilidad y muchos de estos dueños de negocios, en muchos casos autónomos tampoco cuentan con el conocimiento para digitalizarse por su cuenta necesitando el asesoramiento y la gestión de empresas o personas dedicadas a ello, en este caso nosotros.

Las aplicaciones móviles ha demostrado ser una **solución eficaz para optimizar la administración** de negocios de este tipo, facilitando la **comunicación** directa con los clientes, la **automatización** de tareas y la mejora de la **experiencia de usuario**. En el sector veterinario, las apps permiten gestionar **citas, historiales clínicos y ventas** online, lo que contribuye a **diferenciarse de la competencia**.

Sin embargo, la mayoría de estas soluciones están orientadas a clínicas de mayor tamaño o a plataformas web, dejando un margen de mejora y adaptación para pequeños negocios que buscan herramientas **sencillas, económicas y específicas para sus necesidades**.

En este contexto, el desarrollo de una aplicación móvil para la gestión integral de una clínica veterinaria que incluya la administración de mascotas, la reserva de citas, la venta de productos y atención al cliente se presenta como una respuesta pertinente y actual a las necesidades del sector. El proyecto busca aportar una **solución tecnológica accesible y eficiente**, alineada con las tendencias actuales y con potencial para mejorar la calidad del servicio ofrecido tanto a los profesionales veterinarios como a los propietarios de mascotas.

1.2. Abreviaturas y acrónimos

- **PYME:** Pequeñas y Medianas Empresas.
- **BBDD:** Base de Datos.
- **API:** Application Programming Interface.
- **REST:** Representational State Transfer.

Capítulo 2 Objetivos

Para hacer un planteamiento apropiado de los objetivos se recomienda utilizar la Guía para la elaboración de propuestas de TFG en la que se explica cómo definir correctamente los objetivos de un TFG.

2.1. Objetivo general

El objetivo principal de este Trabajo Fin de Ciclo es **desarrollar una aplicación móvil multiplataforma** destinada a **optimizar la gestión integral de una pequeña clínica veterinaria**, facilitando tanto la administración interna como la experiencia de los clientes. La **problemática** detectada radica en la **falta de herramientas digitales sencillas y adaptadas a las necesidades específicas de las pequeñas empresas (PYMEs)**, donde la gestión de citas, la atención al cliente y la venta de productos suelen realizarse de forma manual o con sistemas poco eficientes.

La solución propuesta consiste en una **aplicación móvil**, desarrollada con **Flutter** y conectada a una **API** basada en **Spring Boot y MySQL**, que permitirá a los usuarios registrar y gestionar sus mascotas, reservar citas para los diferentes servicios ofrecidos (consulta, vacunación, peluquería, etc.), acceder a una tienda online de productos especializados y contactar fácilmente con la clínica a través de WhatsApp o llamada telefónica mediante botones de acceso directo. El entorno de trabajo estará limitado a una pequeña clínica veterinaria con tres empleados, aunque la arquitectura y el **diseño de la aplicación permitirán su adaptación a otras pequeñas empresas o comercios con necesidades similares**.

Entre las principales **limitaciones y condicionantes** del proyecto se encuentran la necesidad de desarrollar una **solución intuitiva y fácil de usar**, tanto para los clientes como para el personal de la clínica, así como la **integración eficiente entre el front y el backend, garantizando la seguridad y privacidad de los datos**. Adicionalmente, se busca que la **aplicación sea escalable y flexible** para poder adaptarse a posibles futuras ampliaciones o personalizaciones del cliente.

Con este proyecto **se pretende** no solo resolver la problemática concreta de la clínica veterinaria en cuestión, sino también **sentar las bases para el desarrollo de próximas soluciones digitales accesibles y eficaces para pequeñas y medianas empresas del sector servicios**.

2.2. Objetivos específicos

Permitir a los clientes gestionar de forma sencilla el alta, edición y eliminación de sus mascotas, así como la solicitud, consulta y cancelación de citas para diferentes servicios de la clínica.

1. **Implementar un sistema de reservas** que muestre la disponibilidad de días y horas para consulta y peluquería, facilitando la organización y evitando solapamientos.
2. Desarrollar una **tienda online** integrada donde los usuarios puedan visualizar, filtrar, seleccionar y comprar productos, gestionando su cesta de manera intuitiva.
3. **Contacto directo con la clínica** (llamada y WhatsApp) y opciones de compartir la aplicación para mejorar la comunicación y la visibilidad del negocio.

4. **Herramientas para la gestión de productos al administrador**(alta, edición y eliminación) y la visualización de la agenda de citas mediante un calendario, permitiendo una organización eficiente de los recursos y servicios.
5. **Garantizar la seguridad y privacidad de los datos**, así como la correcta autenticación de usuarios y la protección de endpoints o información sensible de los usuarios como contraseñas.
6. **Asegurar la escalabilidad y adaptabilidad de la aplicación**, diseñar la solución de forma que pueda ser fácilmente adaptada a otras pequeñas empresas o comercios con necesidades similares, permitiendo futuras ampliaciones y personalizaciones.

Requisitos de la solución

Requisitos funcionales:

- La aplicación debe **permitir el registro y gestión de mascotas** y usuarios(Clientes y Administradores).
- Los usuarios podrán **reservar, y cancelar citas** para sus mascotas sobre los diferentes servicios.
- La **tienda online** debe gestionar el catálogo de productos, carrito, pagos y pedidos.
- Deben existir accesos directos a **WhatsApp y llamada telefónica desde la app**.
- El personal de la clínica debe poder acceder a paneles exclusivos como uno para ver la agenda con las citas de los clientes en los diferentes espacios.

Requisitos no funcionales:

- La aplicación debe ser **multiplataforma** (Android/iOS) por ello el desarrollo con Flutter.
- El backend debe estar implementado con **Spring Boot y MySQL**.
- Se debe garantizar la **seguridad y privacidad de los datos** de los usuarios.
- La **interfaz** debe ser **intuitiva y accesible** para usuarios con diferentes niveles de experiencia tecnológica.
- El sistema debe ser escalable para permitir la **adaptación a otras PYMEs** del sector servicios.

Estos objetivos y requisitos definen el alcance y las características esenciales de la solución propuesta, asegurando que la aplicación responda de manera eficaz a las necesidades identificadas tanto de la clínica veterinaria como de sus clientes.

Capítulo 3 Metodología

En este apartado se indicarán las metodologías empleadas para planificación y desarrollo del TFG, así como explicar de modo claro y conciso cómo se han aplicado dichas metodologías.

3.1. Guía rápida de las metodologías de desarrollo del software

A continuación, se incluye una guía rápida sobre las metodologías empleadas.

3.1.1. Proceso de desarrollo de software

El desarrollo de la aplicación para la clínica veterinaria ha seguido un ciclo de vida del desarrollo de software (SDLC) estructurado en diferentes fases, adecuando la metodología a las necesidades del proyecto y a la disponibilidad de recursos humanos

- **Obtención y análisis de requisitos (*requirements analysis*).**

El cliente en este caso, presenta un **grave problema a la hora de la gestión de citas** ya que **actualmente** lo gestiona de forma **manual**, por lo que los trabajadores, en este caso el veterinario y la peluquera mientras trabajan tienen que estar atendiendo a las llamadas telefónicas y tomando las citas a mano, el **objetivo** es **automatizarlo liberando así una carga importante de trabajo**. También quiero aprovechar para presentarle como podría incorporar una **tienda online** donde poder vender los productos que actualmente vende en la tienda de manera local, pudiendo iniciarse así en el **e-commerce**.

- **Diseño (*SW design*).**

El diseño del sistema define la estructura y el funcionamiento de la aplicación tanto a nivel de arquitectura como de componentes, interfaces y flujos de usuario. Este apartado documenta los principales diagramas y decisiones de diseño que han guiado el desarrollo.

Arquitectura general

La aplicación sigue una arquitectura **cliente-servidor** compuesta por tres capas principales:

Frontend móvil: Desarrollado en Flutter (Dart), responsable de la interfaz de usuario y la experiencia interactiva.

Backend (API REST): Desarrollado en Java con Spring Boot, encargado de la lógica de negocio, gestión de datos y comunicación con la base de datos.

Base de datos: MySQL, utilizada para el almacenamiento persistente de la información (usuarios, mascotas, citas, productos, etc.).

Esta arquitectura permite una clara separación de responsabilidades, facilita el mantenimiento y la escalabilidad, y es fácilmente adaptable a otros entornos o dispositivos.

Ejemplo de diagrama de arquitectura Cliente-Servidor

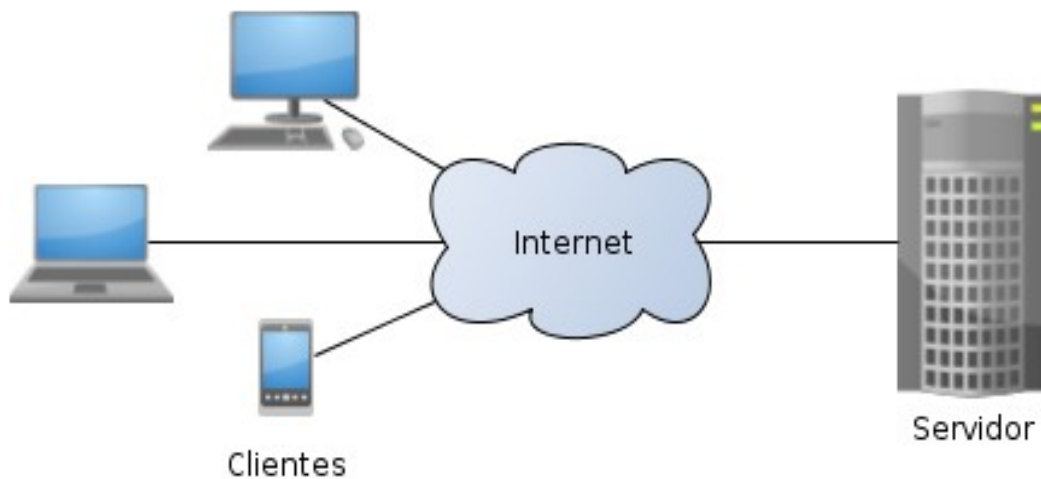


Figura 2: Arquitectura Cliente-Servidor

Figura 1

Diagrama de clases

Se ha elaborado un diagrama de clases para modelar las entidades principales del sistema, como Usuario, Mascota, Cita, Producto, Carrito, Servicio y sus relaciones. Este diagrama ha servido como base para la implementación tanto del backend como de la lógica de negocio, asegurando la coherencia y el correcto mapeo entre objetos y tablas de la base de datos. Se adjunta imagen:

Diagrama de casos de uso

Para visualizar la interacción de los usuarios con la aplicación, se ha incluido un diagrama de casos de uso que muestra las principales funcionalidades accesibles para clientes y administradores: gestión de mascotas, citas, productos, compras y administración de la tienda. Se adjunta imagen:

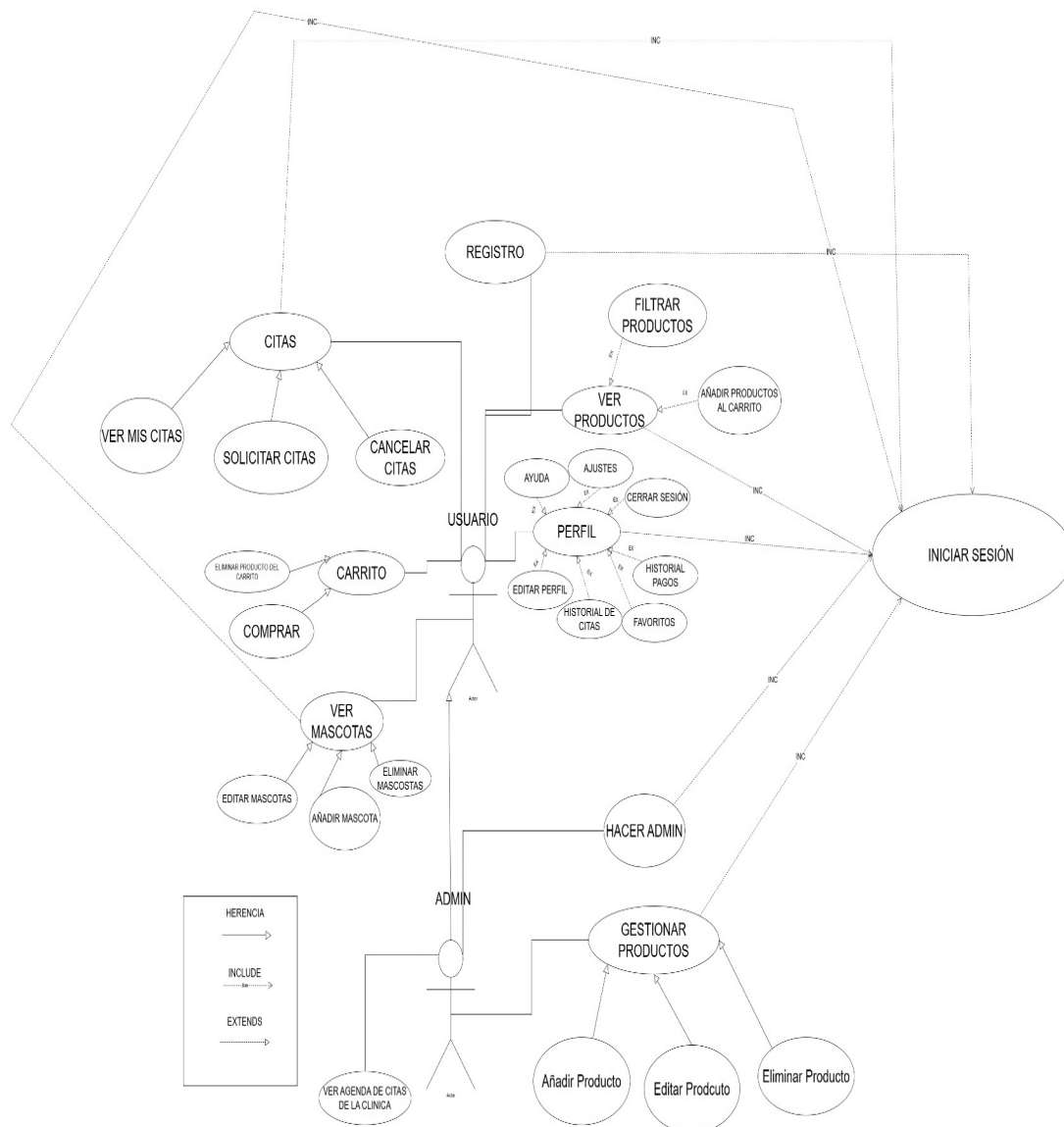


Figura 5: Diagrama casos de uso

Figura 4.3 Diagrama de casos de uso

Esta documentación ha permitido estructurar el desarrollo, facilitar la implementación y asegurar la coherencia y calidad de la aplicación.

- Implementación (*SW construction and coding*).

La aplicación móvil será desarrollada en **Flutter**, que utiliza como lenguaje de programación **Dart**, la **API REST** es desarrollada en **Java** con **SpringBoot** conectada a una **BBDD MySQL** y utilizando el framework **Hibernate**. A continuación haré una descripción sobre las características de estas tecnologías y paquetes utilizados.

Flutter es un framework **multiplataforma** que permite desarrollar aplicaciones nativas para Android, iOS, web y escritorio a partir de un único código fuente. Destacando el “**hot reload**”, que permite visualizar cambios en el código de forma instantánea, y una extensa biblioteca de widgets personalizables.

El **lenguaje Dart**, destaca por su **alto rendimiento** gracias a la compilación Ahead-Of-Time (AOT), su soporte para **null safety**, que ayuda a prevenir errores comunes, y su capacidad para organizar el código de manera clara y **modular**.

En la elaboración del frontend con Flutter se ha hecho uso de las siguientes librerías externas:

Table Calendar: Permite la implementación de un **calendario interactivo** en la aplicación, fundamental para la gestión y visualización de citas disponibles y reservadas.

Image Picker: Facilita la **selección de imágenes desde la galería** del dispositivo o la captura de **fotografías con la cámara**, útil **para añadir imágenes** de mascotas o productos.

http: Proporciona una interfaz sencilla para **realizar peticiones HTTP** y **comunicarse con la API** backend, permitiendo la obtención y envío de datos de forma eficiente.

Shared Preferences: Ofrece un **almacenamiento persistente de datos sencillos** en el dispositivo, ideal para guardar preferencias de usuario, tokens de sesión o configuraciones.

El Backend se ha desarrollado en **Java 17** utilizando **Spring Boot**, un framework que simplifica la creación de aplicaciones empresariales y **APIs REST** gracias a su arquitectura modular, su sistema de auto-configuración y la integración con múltiples tecnologías y librerías. Algunos de los paquetes/librerías utilizados han sido los siguientes:

Springboot Data JPA: Integra **Spring Data JPA con Hibernate**, permitiendo el acceso a la base de datos de forma sencilla y potente.

Springboot Validation: Proporciona **validación automática de datos**, asegurando que la información recibida cumpla con los requisitos definidos.

Springboot Web: Facilita la creación de aplicaciones web y APIs REST, gestionando **peticiones HTTP** y el enrutamiento.

Springboot Dev-Tools: **Mejora la experiencia de desarrollo**, permitiendo reinicios rápidos y herramientas de diagnóstico.

MySQL connector-j: Driver de conexión con la base de datos **MySQL**.

Lombok: Reduce la cantidad de **código repetitivo mediante anotaciones**, facilitando la creación de clases y métodos.

Springboot Test: Permite la realización de **pruebas unitarias e integradas**.

Springboot Security: Añade seguridad a la API, gestionando autenticación y autorización.

También se han utilizado algunos plugins de **Maven** para la compilación del código fuente de Java y el empaquetado y despliegue de Springboot.

- **Pruebas (*testing and verification*).**

Las **pruebas** de diseño y de las distintas funcionalidades se han ido realizando **manualmente durante el desarrollo del proyecto**. En el frontend se ha utilizado la función de **debug** aprovechando el “**Hot Reload**” de Flutter para observar el funcionamiento del código en **tiempo real**. Antes de implementar **cualquier endpoint** a la aplicación **se ha comprobado** su funcionamiento **con Postman**.

- **Despliegue (*deployment*).**

El despliegue de la aplicación desarrollada para la clínica veterinaria implica la puesta en marcha tanto del backend (API) como del cliente móvil. El proceso está pensado para facilitar tanto pruebas locales como el acceso desde distintos dispositivos conectados a la misma red, lo que resulta ideal para entornos pequeños y pruebas en tiempo real.

- **La API** se ejecuta en un dispositivo que actúa como servidor, normalmente un ordenador. Basta con compilar el proyecto y ejecutar el archivo DemoApplication.java, lo que habilita automáticamente los endpoints REST en el puerto configurado (por defecto, el 8080).

- La **base de datos MySQL** debe estar operativa y accesible desde el servidor donde se ejecuta la API.

- Para pruebas locales, tanto la API como la aplicación móvil pueden ejecutarse en el mismo equipo, utilizando emuladores o dispositivos físicos. Además, **se ha habilitará la posibilidad de que cualquier dispositivo conectado a la misma red local que el servidor pueda acceder a la API utilizando la dirección IP del servidor en lugar de localhost**. Esto permite que móviles, tablets u otros ordenadores puedan ejecutar la app y realizar peticiones HTTP a la API sin necesidad de configuraciones adicionales complejas.

Es importante asegurarse de que el firewall del servidor permita conexiones entrantes al puerto donde se ejecuta la API, y que la configuración de CORS en Spring Boot permita peticiones desde los clientes de la red.

Aplicación Móvil(Flutter)

La aplicación móvil puede ejecutarse en **emuladores o dispositivos físicos**. Durante el desarrollo y pruebas, se utiliza la dirección IP local del servidor para que la app pueda comunicarse correctamente con la API.

- **Mantenimiento (*maintenance*).**

El mantenimiento de la aplicación es una fase clave para garantizar su correcto funcionamiento a largo plazo, así como su capacidad de adaptación a nuevas necesidades y requisitos que puedan surgir tras la puesta en producción. Aunque la versión actual de la aplicación cubre las funcionalidades esenciales para la gestión de citas, mascotas y productos, varias características avanzadas permanecen pendientes de implementación y están planificadas para futuras actualizaciones.

Plan de mantenimiento y evolución

Se prevé la **corrección de errores y la mejora continua del software** en función del feedback recibido por parte de los usuarios y del propietario de la clínica(cliente). Esto incluye la **resolución de posibles incidencias detectadas durante el uso** real de la aplicación, así como la **optimización de procesos internos**.

El desarrollo contempla la **incorporación progresiva de funcionalidades actualmente no disponibles**, como la opción de **compartir** la aplicación (que solo tendrá sentido una vez publicada en las tiendas oficiales), el envío de **mensajes de WhatsApp** y **llamadas** directas desde la app, y la integración de **nuevas secciones en el perfil de usuario**.

Entre las **funcionalidades planificadas para futuras versiones** destacan: el **historial de pagos** (que se habilitará tras la implementación de una pasarela de pagos segura, esta decisión sobre los métodos de pago depende mucho de cada cliente), la **gestión de productos favoritos**, el sistema de **notificaciones**, el apartado de **ajustes personalizados** y la sección de **ayuda**, destinada a **mostrar documentación legal** y recomendaciones al consumidor.

El mantenimiento también incluirá la **actualización de dependencias**, la **adaptación a nuevas versiones de los sistemas operativos** y la **mejora de la seguridad**, especialmente en lo relativo a la protección de datos personales y la gestión de autenticación.

Gestión incidencias y actualizaciones

Todas las **incidencias detectadas serán documentadas y priorizadas** para su resolución en futuras versiones, siguiendo un **proceso de revisión y pruebas antes de su despliegue**.

Las **nuevas funcionalidades se integrarán de forma modular**, permitiendo su activación progresiva y la validación con usuarios reales antes de su lanzamiento definitivo.

Se contempla la realización de **pruebas periódicas** y la **monitorización del funcionamiento de la aplicación** para **garantizar su estabilidad y fiabilidad**, así como la capacitación de los usuarios finales en el uso de las nuevas herramientas.

En resumen, la aplicación **está diseñada para ser fácilmente mantenible y escalable, permitiendo** la incorporación de **mejoras y nuevas funcionalidades** según las necesidades de la clínica y el avance tecnológico. El compromiso de mantenimiento asegura que la herramienta evolucione para seguir siendo útil, segura y eficiente en el tiempo.

3.1.2. Metodología de Desarrollo software

Para el desarrollo de la aplicación de gestión para la clínica veterinaria se seguirá la **Metodología en Cascada**. Un enfoque secuencial y estructurado que divide el proceso en fases claramente diferenciadas y ordenadas. Esta metodología resulta especialmente **adecuada** para **proyectos pequeños y con requisitos bien definidos**, ya que **permite avanzar** de manera **ordenada y controlada**, asegurando que cada etapa esté finalizada(o al menos eso piensas en el momento, después surgen cosas) antes de pasar a la siguiente.

Aplicación práctica del modelo en cascada en el proyecto:

1. Análisis y planificación de requisitos iniciales.

Se identifican y definen las entidades principales del sistema (usuarios, mascotas, citas, productos, servicios etc.), apoyándose en la elaboración de un diagrama de clases para visualizar la estructura base de los datos y las relaciones entre ellos.

En esta fase se documentan los requisitos funcionales y no funcionales, estableciendo claramente los objetivos y el alcance del proyecto.

2. Diseño del Sistema.

Una vez definidos los requisitos, se procederá al diseño tanto del backend como del frontend.

En el backend, se crean las bases para las entidades y los endpoints básicos (GET, POST, PATCH/PUT, DELETE) para cada una de ellas.

En el frontend, se realiza un diseño base de la interfaz de usuario, estructurando las pantallas principales y los flujos de navegación.

3. Implementación

Se desarrolla la funcionalidad de manera incremental, pero siempre completando cada módulo o conjunto de funcionalidades antes de pasar al siguiente.

El proceso sigue este orden: primero la autenticación (registro e inicio de sesión), luego la gestión de citas, la tienda y el carrito, las funciones de perfil de usuario y, finalmente, las funcionalidades específicas para administradores (gestión de productos y agenda de citas).

Durante la implementación, se realizaron los ajustes necesarios tanto en el backend como en el frontend, añadiendo endpoints y adaptando la lógica según las necesidades detectadas en cada fase.

4. Pruebas.

Las pruebas se realizan de forma continua tras completar cada funcionalidad, asegurando que cada módulo funcionara correctamente antes de avanzar al siguiente.

Se verifica tanto la integración entre backend y frontend como el correcto funcionamiento de los distintos flujos de usuario.

5. Despliegue

Una vez finalizadas todas las funcionalidades y superadas las pruebas, se procede al despliegue de la aplicación, siguiendo los pasos descritos en el apartado correspondiente.

6. Mantenimiento

Finalmente, se estableció un plan de mantenimiento para la corrección de errores y la incorporación de futuras mejoras o nuevas funcionalidades, según las necesidades de la clínica y el feedback de los usuarios.

Este enfoque lineal y estructurado ha permitido mantener el control sobre el avance del proyecto, minimizando los riesgos y asegurando la calidad del producto final. La metodología en cascada ha resultado especialmente útil en este caso, al tratarse de un proyecto individual, con un alcance bien definido y sin grandes cambios de requisitos durante el desarrollo

3.2. Proceso de *testing*

En este proyecto, el proceso de testing se ha basado principalmente en **pruebas manuales** realizadas **de forma continua** durante el desarrollo de cada funcionalidad. A continuación se detallan algunos tipos de pruebas que se han llevado a cabo.

Pruebas modulares (pruebas unitarias)

Aunque no se han desarrollado pruebas unitarias automatizadas formales, el proceso de desarrollo ha seguido una práctica similar a las pruebas unitarias manuales. Es decir, **cada módulo o funcionalidad se ha probado individualmente** y de forma exhaustiva **antes de avanzar a la siguiente**. Por ejemplo, al implementar el sistema de autenticación (registro y login), se verificó manualmente que cada función y endpoint asociado funcionara correctamente antes de continuar con la gestión de citas o la tienda.

Permitiendo detectar y corregir errores en etapas tempranas

Pruebas de integración

Antes de integrar cualquier **endpoint** en la aplicación móvil, se ha **comprobado su correcto funcionamiento mediante** la herramienta **Postman**. Esto ha permitido validar que la API responde adecuadamente a las solicitudes HTTP (GET, POST, PATCH/PUT, DELETE) y que los datos se gestionan correctamente en la base de datos.

Aunque no se han realizado pruebas de integración automatizadas, **esta validación manual con Postman garantiza que los distintos módulos backend funcionan correctamente** en conjunto y que la comunicación con el frontend es fiable.

Pruebas funcionales manuales

Durante el desarrollo del diseño e implementación de endpoints en la app, se han realizado pruebas funcionales manuales en el frontend utilizando la función de **debug y el “hot reload” de Flutter**. Esto ha facilitado observar en tiempo real el comportamiento de la aplicación y corregir errores de interfaz o lógica antes de avanzar.

Cada funcionalidad implementada (gestión de mascotas, citas, tienda, carrito, perfil) se ha probado manualmente para asegurar que cumple con los requisitos y que la experiencia de usuario es correcta.

3.3.Evolución

El desarrollo del proyecto se dividirá en varias fases, permitiendo avanzar de forma estructurada y asegurando la correcta implementación de cada funcionalidad antes de abordar la siguiente. A continuación, se describen las distintas fases:

Fase I: Análisis, Diseño Inicial y Prototipado

En esta primera fase se recopilarán los requisitos del sistema, se **definen** las **entidades** principales y sus **relaciones** mediante un **diagrama de clases**, también la **interacción** entre los usuarios y la aplicación con un **diagrama de casos de uso** . Además, se estableció la **arquitectura** general de la aplicación, optando por una solución **cliente-servidor** basada en Flutter para el frontend y Spring Boot para el backend, con MySQL como base de datos.

Fase II: Diseño de backend básico

En el backend **se definirán los paquetes** a utilizar en el proyecto, se realizará un **primer diseño de entidades y endpoints básicos** para empezar, se irán modificando a medida que lo requiera el proyecto, en esta fase tendremos definidas todas las entidades y todas contarán con al menos 4 endpoints GET, POST, PATCH/PUT Y DELETE. Las que cuenten con alguna clave ajena del usuario se les hará un get para filtrar por el dni del usuario.

Fase III: Diseño base Frontend

Se realizará un **primer diseño base del frontend** de la aplicación en Flutter sobre el que ir asentando las distintas funcionalidades en las siguientes fases.

Fase IV: Implementación de la Autenticación y Gestión de Usuarios

La cuarta fase se centrará en la **creación de la funcionalidad de registro e inicio de sesión de usuarios**. Se desarrollaron los **endpoints necesarios en el backend para el alta y autenticación de usuarios**, y **se integraron en el frontend** las pantallas y flujos correspondientes, asegurando la correcta validación y gestión de sesiones.

Fase V: Gestión de Mascotas y Citas

En esta fase se implementarán las funcionalidades para que los **usuarios puedan registrar, editar y eliminar mascotas**, así como **solicitar y gestionar citas para consulta o peluquería**. Se desarrollarán los endpoints y formularios necesarios para ello.

Fase VI: Desarrollo de la Tienda y Carrito de Compras

La sexta fase aborda la **creación de la tienda online**, permitiendo a los usuarios **visualizar productos, filtrarlos por diferentes criterios y añadirlos al carrito de compras**. Se implementarán los endpoints necesarios restantes para la gestión de productos y carrito .

Fase VII: Funcionalidades de Perfil de Usuario

En esta etapa se añadirán las **funcionalidades básicas del perfil de usuario**, como la **visualización y edición de datos personales**, el **acceso al historial de citas y las primeras opciones de configuración**. Algunas funcionalidades más avanzadas y adicionales (historial de pagos, favoritos, notificaciones, ajustes y ayuda) se han dejado **planificadas para futuras**

actualizaciones antes del release, a petición del usuario.

Fase VIII: Funcionalidades de los usuarios Administradores

La octava fase estará dedicada a la **implementación de las funciones exclusivas para el administrador**, como la **gestión de productos** (añadir, editar y eliminar) y la **visualización de la agenda** de todas las citas reservadas en peluquería y consulta a través de un calendario. Esto permite una mejor **organización interna y un control eficiente de los servicios ofrecidos por la clínica**.

Fase IX: Últimas Pruebas, Despliegue y Mantenimiento

Finalmente, se realizarán las **últimas pruebas manuales y testeo de todas las funcionalidades implementadas en conjunto**, tanto en el backend (utilizando Postman para los endpoints) como en el frontend (aprovechando el hot reload de Flutter). Tras **validar el correcto funcionamiento del sistema**, se procederá al **despliegue y presentación** al cliente del **primer release** en entorno local y **se establecerá un plan de mantenimiento** para la futura incorporación de nuevas funcionalidades y corrección de posibles incidencias.

Esta división en fases ha permitido avanzar de forma ordenada, asegurando la calidad y la coherencia del desarrollo en cada etapa del proyecto.

3.4. Marco tecnológico

En esta sección se enumeran las tecnologías y herramientas utilizadas en la elaboración del TFG. A continuación, se citan algunos ejemplos.

3.4.1. Herramientas CASE (Computer Aided Software Engineering)

Las herramientas CASE están destinadas a facilitar una o varias de las tareas implicadas en el ciclo de vida del desarrollo de software. Se pueden dividir en la siguientes categorías:

- Draw.io: Herramienta utilizado para el modelado de diagramas: <https://www.drawio.com/>
- Canva: Herramienta para el modelado de diagrama de Gantt: <https://www.canva.com/>

3.4.2. IDE (Integrated Development Environment)

- MySQL Workbench: <https://www.mysql.com/products/workbench/>
- Visual Studio Code: <https://code.visualstudio.com/>

3.4.3. Depuración

- Postman: <https://www.postman.com/>

3.4.4. Repositorios y control de versiones

- Git: <https://git-scm.com/>
- Github: <https://github.com/>

3.4.5. Documentación

- LibreOffice Writer: <https://es.libreoffice.org/>

3.4.6. Gestión y planificación de proyectos

No se ha utilizado ninguna herramienta concreta para gestionar la planificación del proyecto, tampoco ha hecho falta ya que el equipo de trabajo estaba formado por una única persona.

Capítulo 4 Resultados

4.1. Iteración 0

ENTREVISTA CON EL CLIENTE

En la primera entrevista con **se propone al cliente**(mi padre) la idea de poder **desarrollar una aplicación para su negocio**, principalmente el cliente requiere un software para **gestionar las reserva de citas**, supondría una gran descarga de trabajo el poder llegar a **automatizar este proceso**. Desde la parte desarrolladora **se le propone** también la posibilidad de elaborar un apartado para la **tienda online**, esta sugerencia es aceptada, pero no asegura que se vaya a terminar implementando ya que por ahora no estaba realizando venta online en su negocio, pero **se deja abierta la posibilidad de implementarla**.

4.1.1. IEEE 830

4.1.1.1. Introducción

4.1.1.1.1 Propósito

El propósito de este documento es especificar de manera clara y detallada los requisitos funcionales y no funcionales del sistema software para la gestión de una clínica veterinaria. Este documento está dirigido a los desarrolladores, al propietario de la clínica y a cualquier persona interesada en comprender el alcance y las características del sistema. Servirá como base para el diseño, desarrollo, validación y mantenimiento de la aplicación.

4.1.1.1.2 Ámbito del Sistema

El sistema, denominado “App Clínica Veterinaria”, es una **aplicación multiplataforma** destinada a facilitar la gestión integral de una clínica veterinaria. Permitirá a los clientes **gestionar sus mascotas, reservar y cancelar citas para consulta y peluquería, y comprar productos en la tienda online**. El **administrador**, además, **podrá gestionar productos y visualizar la agenda de citas**.

El sistema no incluye la gestión de inventario físico, contabilidad avanzada ni integración con sistemas externos de facturación.

Beneficios esperados:

- Mejora de la **eficiencia en la gestión de citas** y clientes.
- **Facilita la comunicación** entre la clínica y los usuarios.
- Permite la **venta de productos de forma digital**.
- **Escalabilidad** para adaptarse a **otras PYMEs** con necesidades similares.

4.1.1.1.3 Ámbito del Sistema

IEEE 830-1998: Estándar para la especificación de requisitos de software.
(<https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf>)

Documentación oficial de Flutter: <https://flutter.dev/>

Documentación oficial de Spring Boot: <https://spring.io/projects/spring-boot>

Documentación oficial de MySQL: <https://www.mysql.com/>

4.1.1.2. Descripción General

4.1.1.2.1 Perspectiva del Producto

La aplicación de gestión para la clínica veterinaria es un sistema **software independiente, concebido para** ser utilizado tanto por **clientes** (propietarios de mascotas) como por el **personal administrativo de la clínica**. El sistema **no depende de otros productos ni se integra con sistemas externos**, aunque **podría ampliarse en el futuro** para conectarse con soluciones de pago o sistemas de gestión más amplios, como conectarse a otra aplicación de escritorio.

La **arquitectura** del sistema es **cliente-servidor**, compuesta por una **aplicación móvil** multiplataforma desarrollada en Flutter (frontend), una **API REST** implementada con Spring Boot (backend) y una **base de datos MySQL** para el almacenamiento de la información. La **comunicación entre el frontend y el backend** se realiza mediante **peticiones HTTP**.

4.1.1.2.2 Funciones del Producto

El sistema proporciona las siguientes funciones principales, diferenciadas por tipo de usuario:

- Para el **usuario Cliente**:

- Registro e inicio de sesión en la aplicación.
- Gestión de mascotas (alta, edición, eliminación, visualización).
- Solicitud de citas para consulta y peluquería.
- Visualización y cancelación de citas programadas.
- Acceso a la tienda de productos: visualización, filtrado y búsqueda de productos.
- Añadir productos al carrito, modificar cantidades y eliminar productos del carrito.
- Visualización y edición de perfil.
- Acceso al historial de mis citas.
- Comunicación con la clínica mediante llamada o WhatsApp (funcionalidad prevista para futuras versiones).

Para el **administrador**:

- Gestión de productos de la tienda (alta, edición y eliminación).

- Visualización de la agenda de citas en formato calendario.
- Control y gestión de usuarios (opcional, según necesidades futuras).

Para que todo esto resulte más visual se adjunta abajo de nuevo una imagen del diagrama de casos de uso.

4.1.1.2.3 Características de los Usuarios

Usuario Cliente:

Propietarios de mascotas que buscan gestionar sus citas y compras de productos.

Nivel técnico: **usuario medio de aplicaciones móviles**, sin necesidad de conocimientos avanzados de informática.

Administradores:

Personal de la clínica encargado de la **gestión de productos y agenda**.

Nivel técnico: **usuario medio**, con conocimientos básicos de gestión informática.

La **interfaz** está diseñada para ser **intuitiva y accesible**, minimizando la curva de aprendizaje **para todos los perfiles de usuario**.

4.1.1.2.4 Restricciones

La aplicación debe estar **disponible** para dispositivos móviles **Android e iOS**.

El **backend** debe desarrollarse en **Java (Spring Boot)** y la **base de datos** debe ser **MySQL**.

La **comunicación** entre **frontend y backend** se realizará mediante **API REST**.

El **sistema** debe estar **en español**.

Debe garantizarse la **protección de datos personales** conforme a la legislación vigente.

El **acceso a funcionalidades de administración** estará **restringido** a usuarios autenticados con rol de administrador.

4.1.1.2.5 Suposiciones y Dependencias

Se asume que los **usuarios** tendrán acceso a **dispositivos móviles compatibles y conexión a internet**.

El correcto funcionamiento del sistema depende de la disponibilidad de la red local o internet para la comunicación entre la app y el servidor.

Se asume que la infraestructura de la clínica permite el despliegue del backend y la base de datos en un servidor accesible por la red local, o por la contratación de un servicio de Hosting.

4.1.1.2.6 Requisitos Futuros

Integración de **pasarela de pagos** para compras en la tienda.

Cambiar contraseña mediante correo y recordar usuario logueado

Implementación de **productos favoritos**, **notificaciones** y **ajustes personalizados** en el perfil de usuario.

Mejora de la **sección de ayuda**, incluyendo **documentación legal** y soporte al consumidor.

Publicación de la app en tiendas oficiales y habilitación de la función de compartir.

Integración con sistemas externos de gestión o facturación si la clínica lo requiere en el futuro.

Contratación de un hosting para lanzar la aplicación

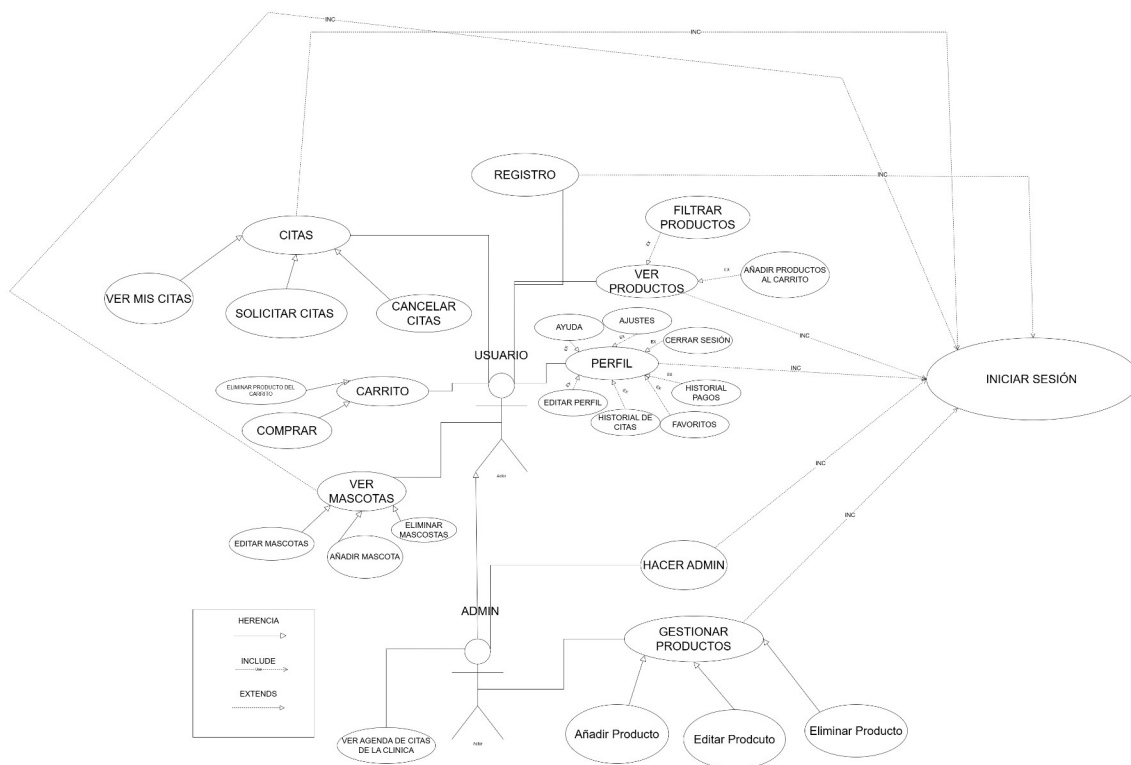


Figura 6: Diagrama casos de uso

4.1.2. Estudio de viabilidad

4.1.2.1 Viabilidad técnica

Tecnologías utilizadas: El proyecto se basa en tecnologías ampliamente utilizadas y con buena documentación: **Flutter** para el **frontend**, **Spring Boot** para el **backend** y **MySQL** como sistema de gestión de **bases de datos**. Todas ellas son de uso libre y cuentan con una gran comunidad de soporte.

Recursos Disponibles: El **desarrollo** se realiza en un **entorno local**, utilizando un **ordenador personal estándar**, sin requerir hardware especializado. La infraestructura necesaria (ordenador para el backend, dispositivos móviles para pruebas) está disponible.

Capacidades Técnicas: El desarrollador posee conocimientos y experiencia suficientes en las tecnologías seleccionadas, lo que garantiza la correcta implementación y mantenimiento del sistema.

Escalabilidad: El sistema está **diseñado para poder adaptarse** a más usuarios o a otras PYMEs en el futuro, si es necesario.

4.1.2.2 Viabilidad económica

Coste de licencias: Todas las **herramientas y frameworks empleados son gratuitos** o de código abierto, por lo que no existen costes de licencias de software.

Coste de infraestructuras: Podría usarse un ordenador como servidor en la propia clínica pero lo **ideal** sería **contratar** un servicio de **Hosting Web**.

Coste de desarrollo: Al ser un **proyecto realizado por una sola persona** (el propio estudiante), **no existen costes de personal adicionales**.

Mantenimiento: El mantenimiento **puede ser realizado por el propio desarrollador**, el cliente pagará una cantidad cada año por el servicio de mantenimiento.

PRESUPUESTO

Desarrollo de la aplicación

Horas de desarrollo: 90 horas

Tarifa del desarrollador: 12 €/hora

Coste total de desarrollo: 90 horas×12 €/hora=1.080 €

Licencias y herramientas

Frameworks y herramientas: Flutter, Spring Boot, MySQL, Visual Studio Code, Git, etc.

Coste: 0 € (todas son gratuitas o de código abierto)

Infraestructura y hosting

Opción básica (servidor local en clínica): 0 € (utilizando ordenador propio)

Opción recomendada (servicio de hosting web):

Coste: 96€/año en Hotinger tarifa Cloud StartUp

Mantenimiento anual

Mantenimiento básico:

Horas estimadas: 20 horas/año

Tarifa del desarrollador: 12 €/hora

Coste anual de mantenimiento: 20 horas × 12 €/hora = 240 €

Presupuesto total (primer año)

Desarrollo + Hosting (año1) = 1.200 € + 96 € = **1.296 €**

Mantenimiento anual (segundo año en adelante)

120 € (mantenimiento) + 96€ (hosting) = **336 €/año**

4.1.2.3 Viabilidad legal

El análisis legal del proyecto es fundamental para garantizar que el desarrollo y la puesta en marcha de la aplicación no infringen ninguna ley o normativa vigente. En este sentido, se han considerado los siguientes aspectos:

Protección de datos personales: La aplicación gestionará datos personales de los usuarios (nombre, DNI, contacto, información sobre mascotas, citas, etc.), por lo que debe cumplir con la **legislación vigente** en materia de protección de datos, concretamente el **Reglamento General de Protección de Datos (RGPD) de la Unión Europea y la Ley Orgánica de Protección de Datos (LOPDGDD) en España**.

Se prevé la adopción de medidas técnicas y organizativas para garantizar la confidencialidad, integridad y disponibilidad de los datos, como la autenticación de usuarios, el cifrado de contraseñas y el acceso restringido a la información sensible.

Propiedad intelectual y licencias: Todas las **tecnologías y librerías utilizadas** (Flutter, Spring Boot, MySQL, etc.) son de código abierto o **disponen de licencias libres** para su uso en proyectos comerciales y personales.

Condiciones de uso y documentación legal: Se prevé la inclusión, en futuras versiones, de una **sección de ayuda y documentación legal en la aplicación**, donde **se informará a los usuarios sobre el tratamiento de sus datos, los**

términos y condiciones de uso y los derechos.

Cumplimiento de normativa del sector: El proyecto **no infringe ninguna normativa específica del sector veterinario**, ya que se limita a la gestión interna de la clínica y a la interacción con los clientes, sin sustituir funciones reservadas a profesionales colegiados ni manipular información sanitaria sensible más allá de la gestión de citas y datos básicos de las mascotas.

Conclusión

El análisis de viabilidad **demuestra que el desarrollo de la aplicación es factible** desde el punto de vista técnico, económico y legal. No existen barreras significativas que impidan la realización del proyecto, y se espera que la implantación del sistema aporte beneficios claros a la gestión de la clínica veterinaria.

4.1.3. Plan de proyecto

Explicar con un diagrama de Gantt las distintas iteraciones del proyecto, así como las fechas de su realización.

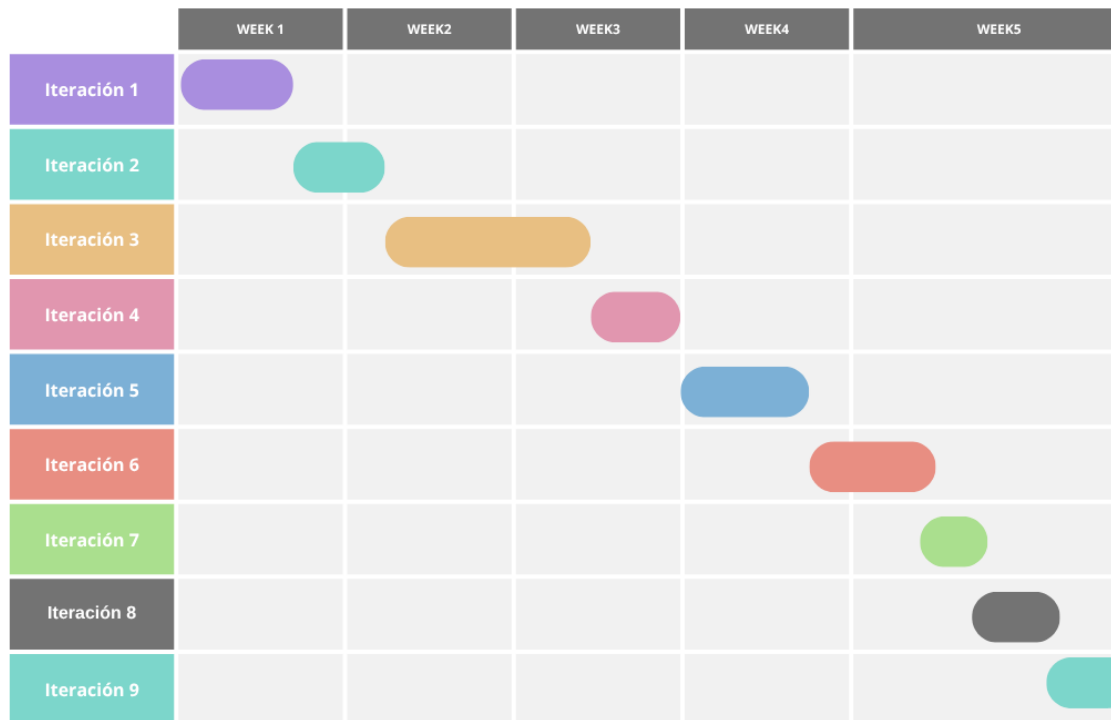


Figura 7: Diagrama de Gantt

4.2. Iteración 1 - Fase I: Análisis, Diseño Inicial y Prototipado

4.2.1. Diseño de las entidades principales

Se diseñan las entidades principales que intervendrán en la aplicación, para ello se realiza el siguiente diagrama de clases:

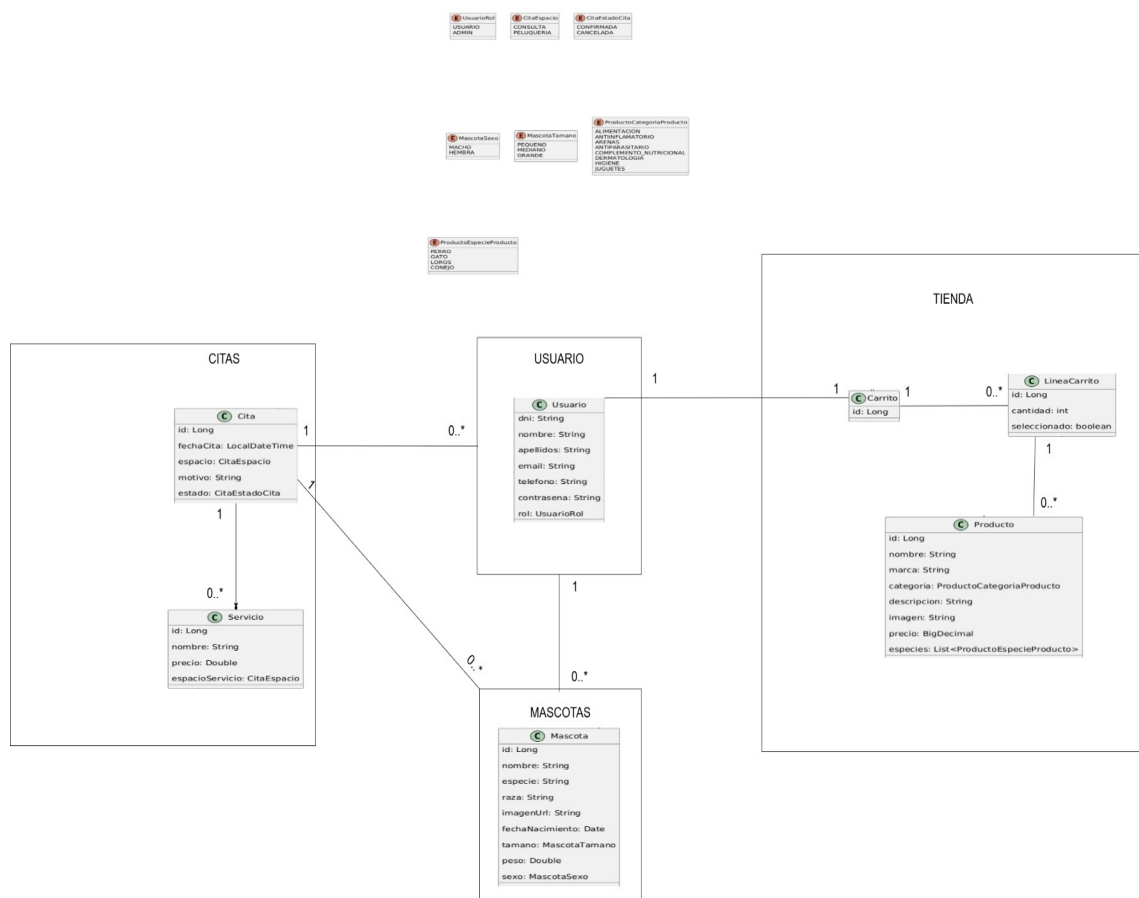


Figura 8: Diagrama de clases

4.2.2 Definición de la arquitectura del proyecto

La arquitectura del proyecto será una arquitectura cliente-servidor, desde el frontend se realizarán peticiones http al backend para obtener la información de la BBDD a través de los endpoints correspondientes. Adjunto imagen de una arquitectura cliente-servidor:

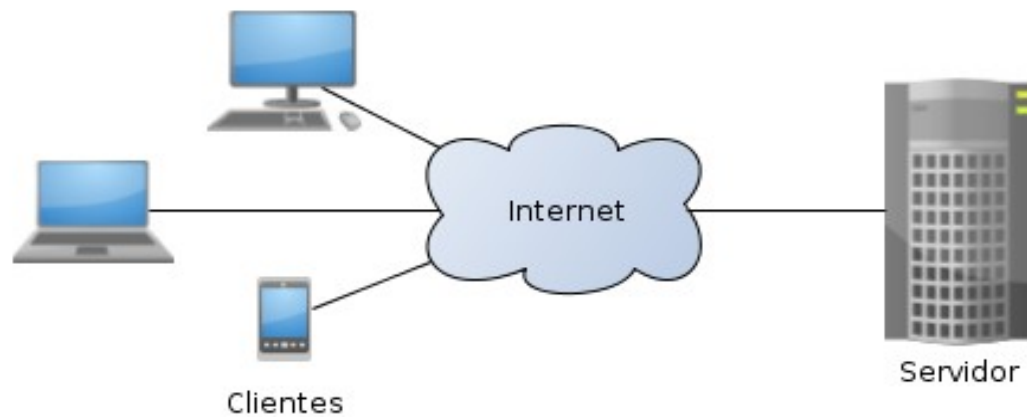


Figura 9: Arquitectura Cliente-Servidor

4.2.3 Interacción de los usuarios con la aplicación

Para describir la interacción de los distintos tipos de usuario con la aplicación, diseñamos el siguiente diagrama de casos de uso:

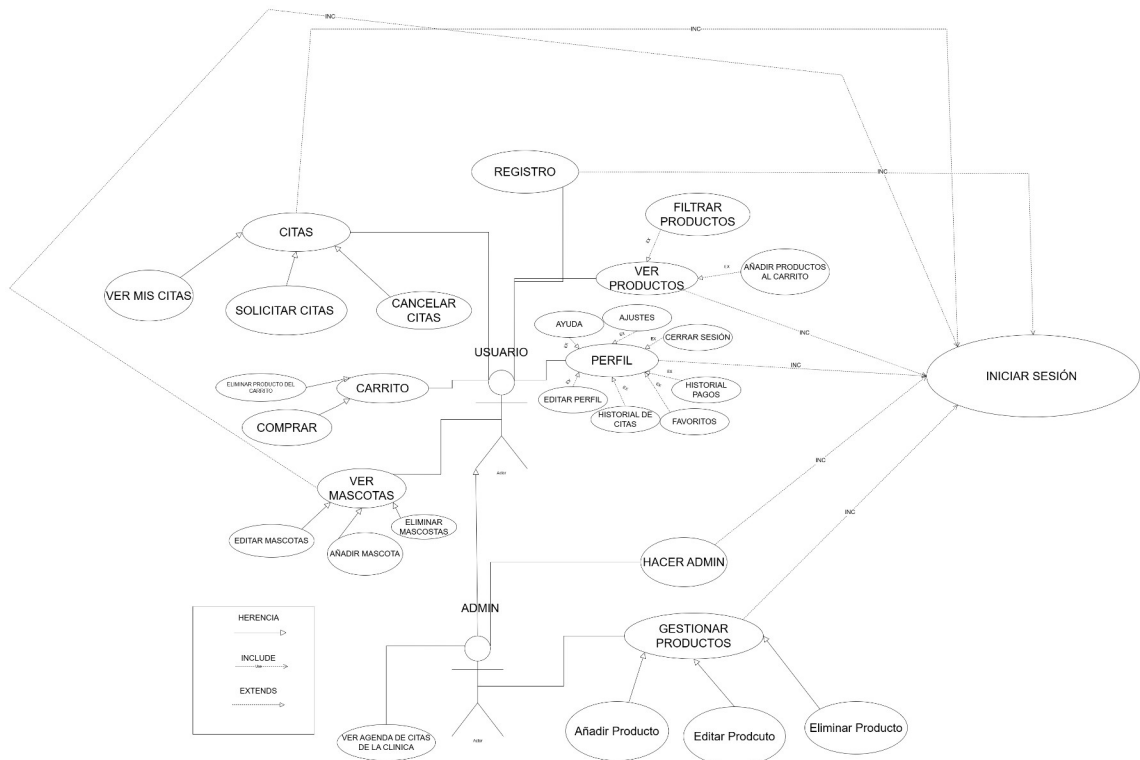


Figura 10: Diagrama casos de uso

4.3. Iteración 2 - Fase II: Diseño de backend básico

4.3.1 Creación del backend para la API REST

Se crea el backend, a través de la siguiente página <https://start.spring.io/>, añadiendo las dependencias correspondientes que ya mencionamos anteriormente.

Una vez abrimos el proyecto descargado empezamos a crear nuestros paquetes en mi caso serán los siguientes: advices, controllers, dtos, entities, exceptions, mappers, repositories y services. Estos se utilizarán de la siguiente manera:

Advices: Contiene clases para el manejo global de excepciones y respuestas personalizadas de error en la API.

Controllers: Gestionan las peticiones HTTP que llegan a la aplicación. Definen los endpoints de la API y coordinan la lógica de entrada/salida entre el frontend y el backend.

Dtos(Data Transfer Objects): Son objetos simples usados para transferir datos entre capas (por ejemplo, entre el backend y el frontend). Permiten controlar qué datos se exponen y en qué formato, evitando exponer directamente las entidades de la base de datos.

Entities: Representan las tablas de la base de datos en forma de clases Java. Se usan para mapear los datos almacenados en la base de datos mediante JPA/Hibernate.

Exceptions: Contienen clases que representan errores personalizados (por ejemplo, `ProductoNotFoundException`). Permiten lanzar y manejar errores específicos del dominio de la aplicación.

Mappers: Se encargan de convertir entre entidades y DTOs (y viceversa). Permiten separar la lógica de persistencia de la lógica de presentación, facilitando la transformación de datos.

Repositories: Definen la interfaz para acceder a los datos en la base de datos (por ejemplo, extendiendo `JPA Repository`). Permiten realizar operaciones CRUD y consultas personalizadas sobre las entidades.

Services: Contienen la lógica de negocio de la aplicación. Orquestan las operaciones entre repositorios, mappers y otras dependencias para cumplir con los requisitos funcionales.

4.3.2 Conexión con la BBDD

Teniendo el driver para la conexión con una bbdd instalado, solo queda conectarnos a la bbdd. Para ello necesitaremos crear la BBDD y ajustar unos parámetros en la carpeta `resources` `application.properties`.

Crearemos la bbdd de la siguiente manera en MySQL Workbench:

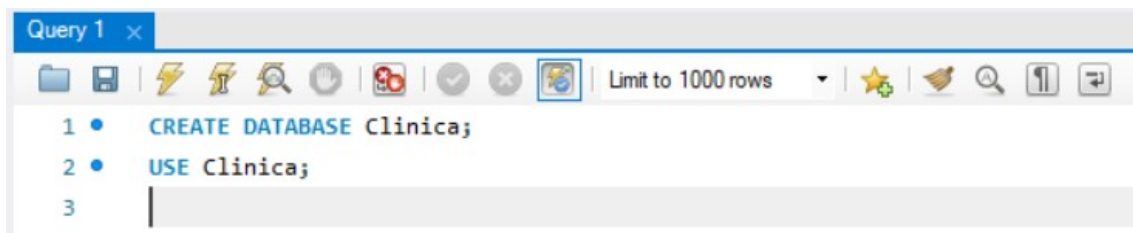
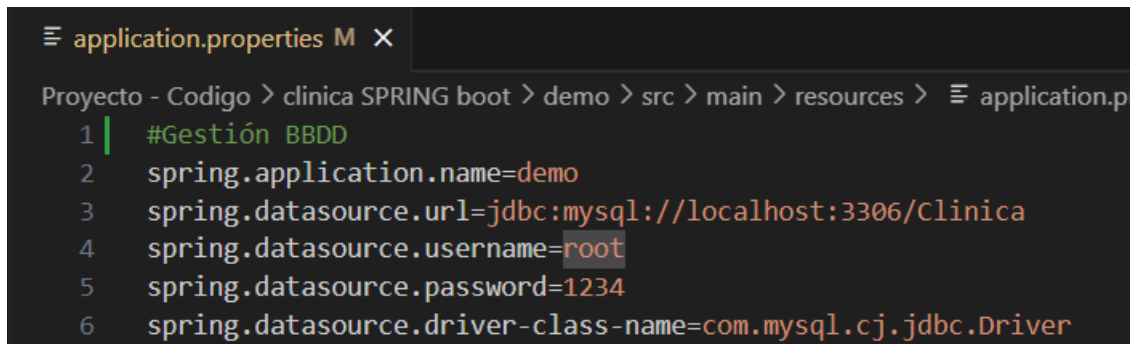


Figura 11: Creacion BBDD

Figura 4.4 Imagen Creación BBDD

Y una vez creada haremos la conexión mediante `application.properties` de la siguiente manera:



```
application.properties M X
Proyecto - Codigo > clinica SPRING boot > demo > src > main > resources > application.p
1 | #Gestión BBDD
2   spring.application.name=demo
3   spring.datasource.url=jdbc:mysql://localhost:3306/Clinica
4   spring.datasource.username=root
5   spring.datasource.password=1234
6   spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

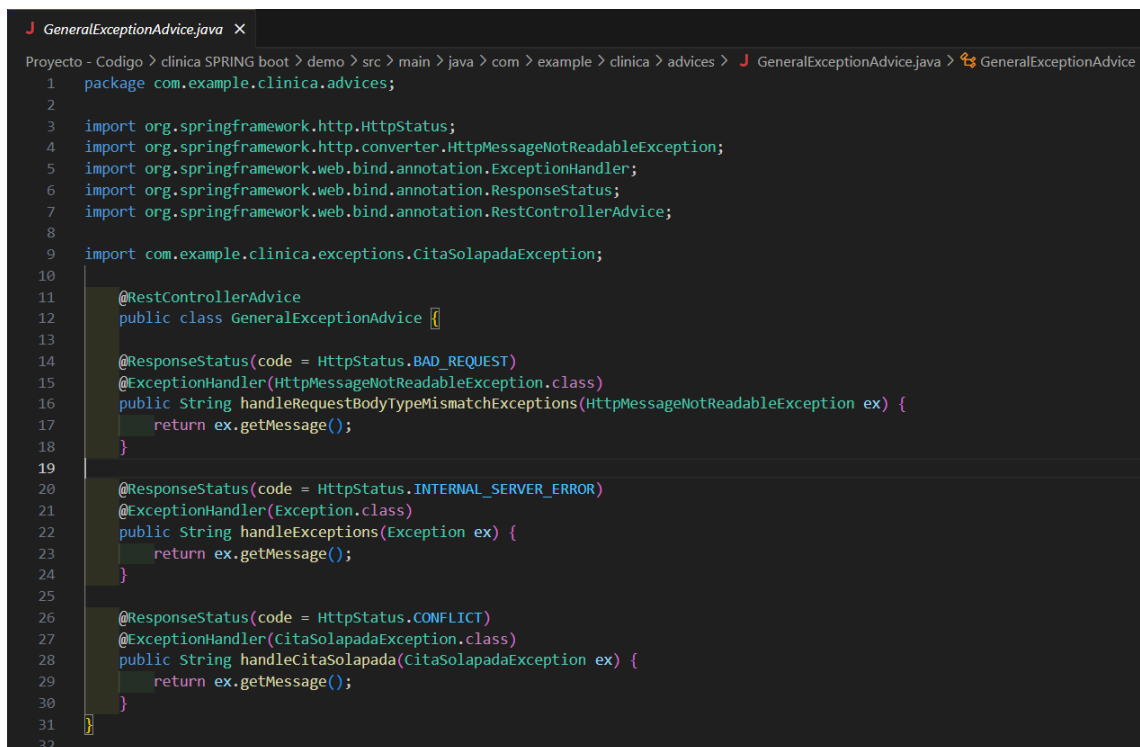
Figura 12: Conexion BBDD

Figura 4.5. Conexión BBDD

4.3.3 Diseño de entidades básico

Comenzaremos con el diseño básico de las entidades definidas en el diagrama de clases, lo fundamental será crear su entidad, dtos, mappers, repositories, controllers y services. Como ejemplo se adjunta el código de la entidad Servicio:

Advices, Servicio no dispone de ningún advice concreto por lo que se remite al `GeneralExceptionHandler`:



```
1 package com.example.clinica.advices;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.http.converter.HttpMessageNotReadableException;
5 import org.springframework.web.bind.annotation.ExceptionHandler;
6 import org.springframework.web.bind.annotation.ResponseStatus;
7 import org.springframework.web.bind.annotation.RestControllerAdvice;
8
9 import com.example.clinica.exceptions.CitaSolapadaException;
10
11 @RestControllerAdvice
12 public class GeneralExceptionHandler {
13
14     @ResponseStatus(code = HttpStatus.BAD_REQUEST)
15     @ExceptionHandler(HttpMessageNotReadableException.class)
16     public String handleRequestContentTypeMismatchExceptions(HttpMessageNotReadableException ex) {
17         return ex.getMessage();
18     }
19
20     @ResponseStatus(code = HttpStatus.INTERNAL_SERVER_ERROR)
21     @ExceptionHandler(Exception.class)
22     public String handleExceptions(Exception ex) {
23         return ex.getMessage();
24     }
25
26     @ResponseStatus(code = HttpStatus.CONFLICT)
27     @ExceptionHandler(CitaSolapadaException.class)
28     public String handleCitaSolapada(CitaSolapadaException ex) {
29         return ex.getMessage();
30     }
31 }
32
```

Figura 13: GeneralExceptionHandler

Entities → Servicio.java

```
Proyecto - Codigo > clinica SPRING boot > demo > src > main > java > com > example > clinica > entidades > Servicio.java > Servicio

1  package com.example.clinica.entidades;
2
3  import com.example.clinica.entidades.Cita.Espacio;
4  import jakarta.persistence.*;
5  import jakarta.validation.constraints.NotNull;
6  import jakarta.validation.constraints.Positive;
7  import lombok.*;
8
9  @Entity
10 @Table(name = "servicios")
11 @Data
12 @NoArgsConstructor
13 @AllArgsConstructor
14 public class Servicio {
15
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19
20     @Column(unique = true)//evita servicios duplicados
21     @NotNull(message = "El nombre es obligatorio")
22     private String nombre;
23
24     @NotNull(message = "El precio es obligatorio")
25     @Positive(message = "El precio debe ser mayor a 0")
26     private Double precio;
27
28     @Enumerated(EnumType.STRING)
29     private Espacio espacioServicio; // Ej: CONSULTA, PELUQUERIA
30 }
31
```

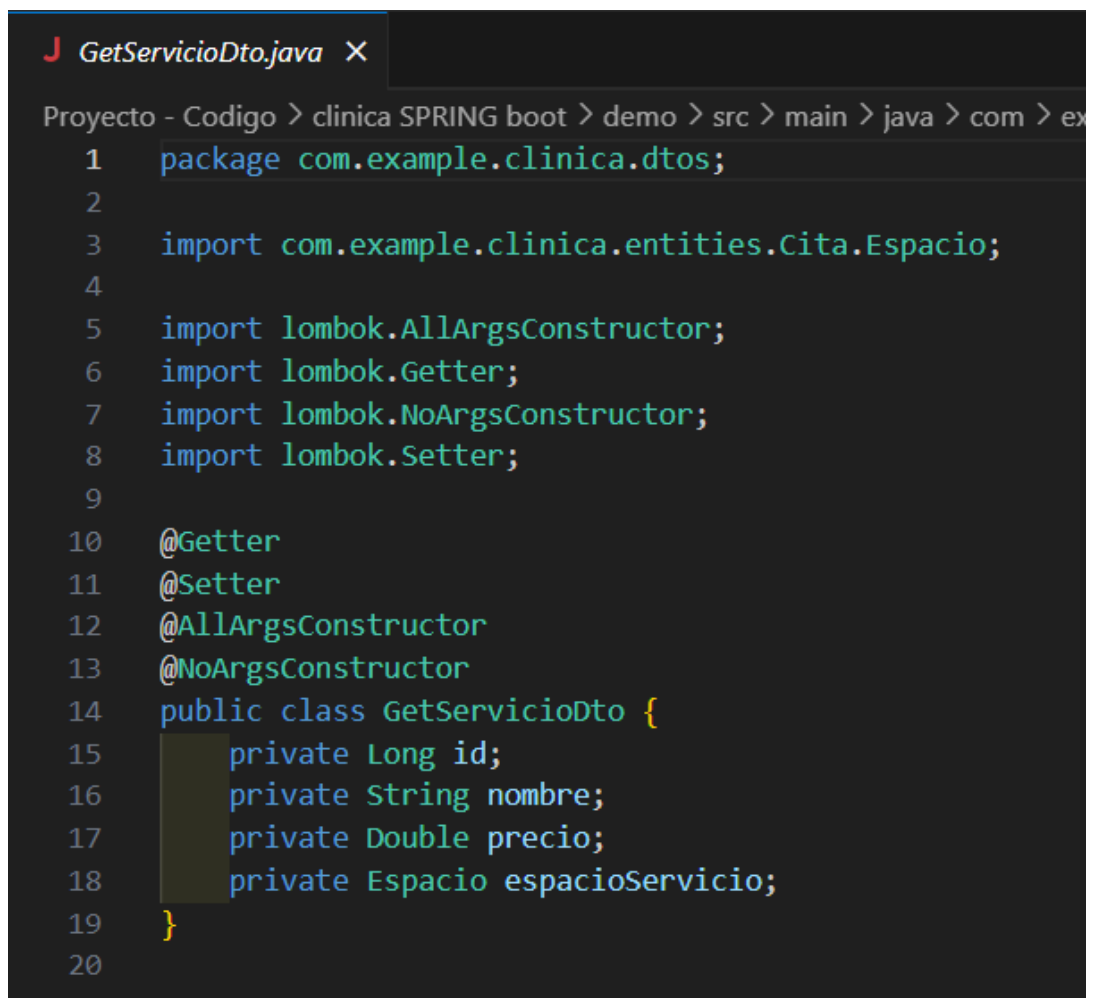
Figura 14: Servicio.java

dtos → CreateServicioDto.java

```
Proyecto - Codigo > clinica SPRING boot > demo > src > main > java > com > example > clinica > dtos > CreateServicioDto.java

1  package com.example.clinica.dtos;
2
3  import com.example.clinica.entidades.Cita.Espacio;
4
5  import jakarta.validation.constraints.Min;
6  import jakarta.validation.constraints.NotBlank;
7  import jakarta.validation.constraints.NotNull;
8  import lombok.*;
9  import lombok.Data;
10 import lombok.Getter;
11 import lombok.NoArgsConstructor;
12 import lombok.Setter;
13
14 @Getter//genera getters
15 @Setter//genera stters
16 @Data
17 @AllArgsConstructor
18 @NoArgsConstructor
19 public class CreateServicioDto {
20     private Long id;
21
22     @NotBlank(message = "nombre es obligatorio")
23     private String nombre;
24
25     @NotNull(message = "precio es obligatorio")
26     @Min(value = 0, message = "El precio debe ser mayor o igual a 0")
27     private Double precio;
28
29     @NotBlank(message = "espacio es obligatorio")
30     private Espacio espacioServicio;
31 }
32
```

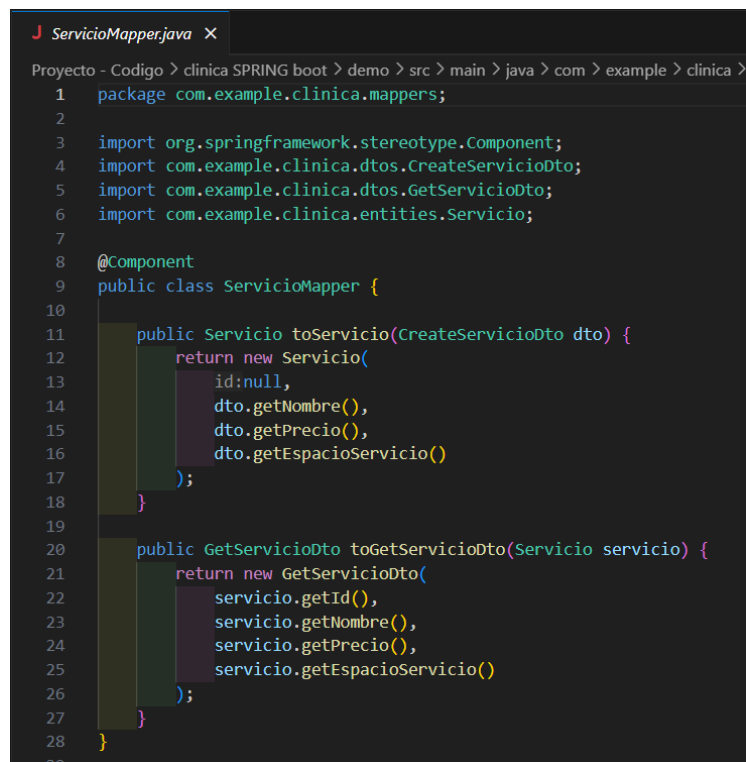

dtos → GetServicioDto.java



```
GetServicioDto.java X
Proyecto - Codigo > clinica SPRING boot > demo > src > main > java > com > ex
1  package com.example.clinica.dtos;
2
3  import com.example.clinica.entities.Cita.Espacio;
4
5  import lombok.AllArgsConstructor;
6  import lombok.Getter;
7  import lombok.NoArgsConstructor;
8  import lombok.Setter;
9
10 @Getter
11 @Setter
12 @AllArgsConstructor
13 @NoArgsConstructor
14 public class GetServicioDto {
15     private Long id;
16     private String nombre;
17     private Double precio;
18     private Espacio espacioServicio;
19 }
20
```

Figura 16: GetServicioDto

mappers → ServicioMapper.java



```
1 package com.example.clinica.mappers;
2
3 import org.springframework.stereotype.Component;
4 import com.example.clinica.dtos.CreateServicioDto;
5 import com.example.clinica.dtos.GetServicioDto;
6 import com.example.clinica.entities.Servicio;
7
8 @Component
9 public class ServicioMapper {
10
11     public Servicio toServicio(CreateServicioDto dto) {
12         return new Servicio(
13             id:null,
14             dto.getNombre(),
15             dto.getPrecio(),
16             dto.getEspacioServicio()
17         );
18     }
19
20     public GetServicioDto toGetServicioDto(Servicio servicio) {
21         return new GetServicioDto(
22             servicio.getId(),
23             servicio.getNombre(),
24             servicio.getPrecio(),
25             servicio.getEspacioServicio()
26         );
27     }
28 }
```

Figura 17: ServicioMapper

repositories → ServicioRepository.java



```
1 package com.example.clinica.repositories;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.stereotype.Repository;
7
8 import com.example.clinica.entities.Cita.Espacio;
9 import com.example.clinica.entities.Servicio;
10
11 @Repository
12 public interface ServicioRepository extends JpaRepository<Servicio, Long> {
13     List<Servicio> findByEspacioServicio(Espacio espacioServicio);
14 }
15
```

Figura 18: ServicioRepository

controllers → ServicioController.java

```
ServicioController.java X
Proyecto - Codigo > clinica SPRING boot > demo > src > main > java > com > example > clinica > controllers > J ServicioController.java > ...
23 public class ServicioController {
24     // Constructor que inyecta el servicio
25     public ServicioController(ServicioService servicioService) {
26         this.servicioService = servicioService;
27     }
28
29     // Obtener todos los servicios o filtrar por espacioServicio
30     @GetMapping
31     public List<GetServicioDto> getServicios(@RequestParam(required = false) String espacioServicio) {
32         if (espacioServicio != null) {
33             Espacio espacio = Espacio.valueOf(espacioServicio.toUpperCase());
34             return servicioService.getServiciosByEspacio(espacio);
35         }
36         return servicioService.getAllServicios();
37     }
38
39     // Obtener un servicio por ID
40     @GetMapping("/{id}")
41     public ResponseEntity<GetServicioDto> getServicioById(@PathVariable Long id) {
42         return servicioService.getServicioById(id)
43             .map(ResponseEntity::ok)
44             .orElseGet(() -> ResponseEntity.notFound().build());
45     }
46
47     // Crear un nuevo servicio
48     @PostMapping
49     public ResponseEntity<GetServicioDto> createServicio(@RequestBody CreateServicioDto dto) {
50         Servicio nuevoServicio = servicioService.createServicio(dto);
51         return ResponseEntity.status(HttpStatus.CREATED)
52             .body(servicioService.getServicioById(nuevoServicio.getId()).orElse(other:null));
53     }
54
55     // Actualizar un servicio
56     @PutMapping("/{id}")
57     public ResponseEntity<GetServicioDto> updateServicio(@PathVariable Long id, @RequestBody CreateServicioDto dto) {
58         return servicioService.updateServicio(id, dto)
59             .map(servicio -> ResponseEntity.ok(servicioService.getServicioById(servicio.getId()).orElse(other:null)))
60             .orElseGet(() -> ResponseEntity.notFound().build());
61     }
62
63     // Eliminar un servicio
64     @DeleteMapping("/{id}")
65     public ResponseEntity<Void> deleteServicio(@PathVariable Long id) {
66         try {
67             if (servicioService.deleteServicio(id)) {
68                 return ResponseEntity.noContent().build();
69             }
70             return ResponseEntity.notFound().build();
71         } catch (IllegalStateException ex) {
72             return ResponseEntity.status(HttpStatus.CONFLICT).build(); // 409 Conflict
73         }
74     }
75 }
76
77 }
```

Figura 19: ServicioService

4.4. Iteración 3 - Fase III: Diseño base Frontend

4.4.1 Diseño Login y Sign Up.

Creamos el Sign Up, un formulario para rellenar los siguientes campos DNI, contraseña, Nombre, apellidos, e-mail y teléfono. Y el correspondiente login para introducir el dni y contraseña. Por ahora nada funcional, solo diseño

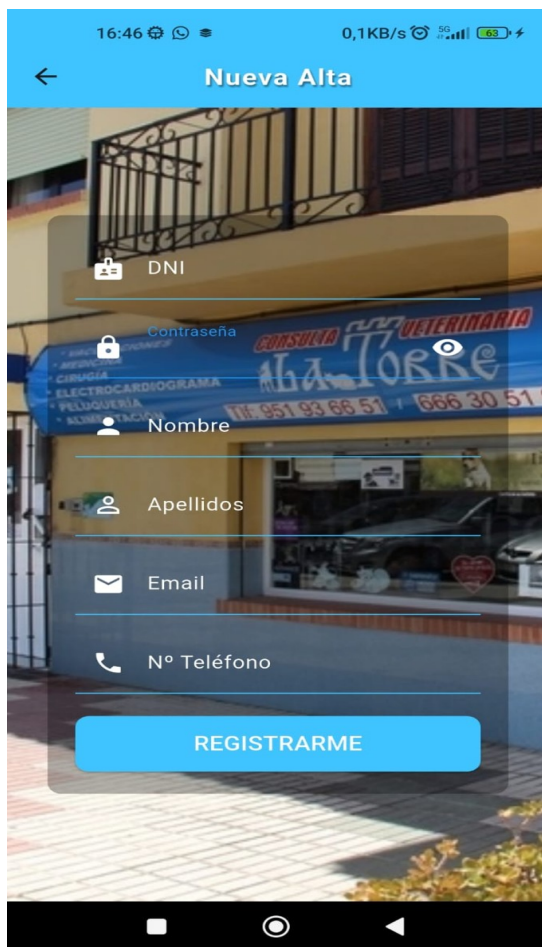


Figura 21: SignUp

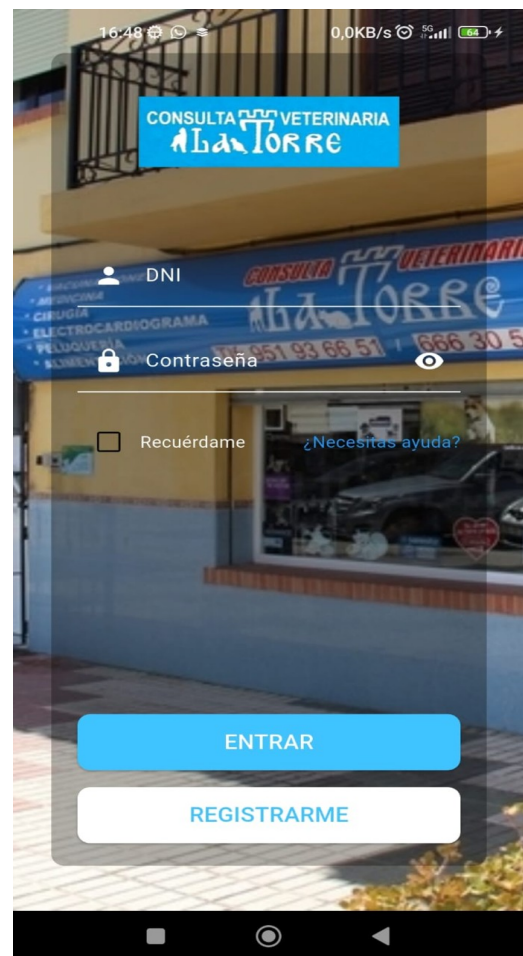


Figura 20: Login

En este fragmento código se ve como se ha creado el **container de la capa semitransparente** donde se encuentran los textfield. Se puede ver como el logo se ha guardado en una carpeta llamada assets, este se debe importar en el archivo pubspec.yaml, al igual que las dependencias, la imagen de fondo se trata de un Image.Network por lo que obtiene la imagen desde internet por su URL.

```
// Capa semitransparente para mejorar la legibilidad
Positioned.fill(
  child: Center(
    child: SingleChildScrollView(
      padding: const EdgeInsets.symmetric(vertical: 20),
      child: Container(
        padding: const EdgeInsets.all(20),
        width: MediaQuery.of(context).size.width * 0.85, // ancho container transparencia
        height: MediaQuery.of(context).size.height * 0.90, // alto container transparencia
        decoration: BoxDecoration(
          color: Colors.black.withOpacity(0.3), // opacidad
          borderRadius: BorderRadius.circular(10), // bordes redondeados
        ), // BoxDecoration
        child: Column(
          mainAxisAlignment: MainAxisAlignment.min,
          children: [
            Image.asset("lib/assets/logoClinica.png", width: 200, height: 100),
            const SizedBox(height: 50), // espacio entre escudo y campo DNI
          ],
        ),
      ),
    ),
  ),
);
```

Figura 22: Container Login

4.4.2 Diseño pantalla inicio.

Se crea una **pantalla de inicio** a la que se navegará al efectuar el login, la navegación se gestará a través de widgets flotantes. De fondo se encuentra una imagen de fondo básica pública de internet

Constará de una **parte superior** donde se encontrará las **letras del logo** en png junto a un icono para acceder a las funciones del perfil de usuario y otro para compartir. A modo de appBar.

Abajo otros dos **botones** uno con un icono de **telefono** y otro con el logo de whatsapp, a través de ellos se podrá llamar y enviar **whatsapp** a los teléfonos de la empresa.

En el **centro** contamos con varios **botones circulares**, formando una huella de perro **para acceder a las distintas secciones de la app**, en el centro y con mayor tamaño se encuentra el de pedir cita.

Por último abajo un semicírculo con un icono de huella en el centro. Donde en un futuro se puede implementar alguna funcionalidad.

La **mayor dificultad** en esta iteración se presenta a la hora de **colocar los botones de navegación principales**. Al querer hacer la navegación de esta forma la dificultad se presenta al situar los botones.

Lo **solucionamos creando 2 widgets** dentro de un contenedor con un espacio determinado. Lo resolvemos creando dos widgets uno donde se crea la estructura del propio botón `_circleMenuBottom` y `positionedCircleCustom`. Este último fue el más complejo ya que había que **posicionar los botones según coordenadas** que funcionaban de un modo un tanto complejo, por lo que al final terminé poco a poco probando y haciéndolo a ojo. En la siguiente imagen se adjuntan imágenes de estos widgets y como utilizarlos:

```
Widget _circleMenuButton({
  //los parámetros de icono e imagen son opcionales
  IconData? icon,
  String? imagePath, // parámetro para la ruta de la imagen
  required String label,
  required VoidCallback onTap,
  double size = 64,
  double fontSize = 14,
  Color color = Colors.white,
  Color iconColor = Colors.teal,
  double elevation = 4,
  bool isMain = false,
}) {
  return Material(
    elevation: elevation,
    shape: const CircleBorder(),
    color: color,
    child: InkWell{//Inkwell hace que los círculos sean clickables
      customBorder: const CircleBorder(),
      onTap: onTap,
      child: SizedBox(
        width: size,
        height: size,
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            // Muestra la imagen si hay un imagePath, de lo contrario muestra el icono
            if (imagePath != null)
              Image.asset(
                imagePath,
                width: isMain ? 50 : 32,
                height: isMain ? 50 : 32,
                fit: BoxFit.contain,
              ) // Image.asset
            else if (icon != null)
              Icon(icon, color: iconColor, size: isMain ? 40 : 28),

            const SizedBox(height: 4),
            //Nombre de la sección
            Text(
              label,
              style: TextStyle(
                color: Color(0xFF1466A3),
                fontWeight: FontWeight.bold,
                fontSize: fontSize,
              ), // TextStyle
              textAlign: TextAlign.center,
            ), // Text
          ], // Column
        ), // SizedBox
      ), // InkWell
    ); // Material
  }
```

Figura 23: código CircleMenuButton

```
// Widget que Posiciona los botones en semicírculo superior alrededor del botón central
Widget _positionedCircleCustom({
  required double angle, //Ángulo en grados donde quieres posicionar el botón respecto al centro.
  required double distance, //Distancia radial desde el centro (en píxeles).
  required Offset centerOffset, //Coordenadas (x,y) del centro del círculo (botón "Pedir Cita").
  required Widget child, //El widget del botón que quieres posicionar.
}) {
  //posicionar boton en plano cartesiano(que movida)
  final double rad = angle * pi / 180;
  return Positioned(
    left: centerOffset.dx + distance * cos(rad) - 32,
    top: centerOffset.dy + distance * sin(rad) - 32,
    child: child,
  );
}
```

Figura 24: código positionedCircleCustom

```
// Menú circular central ajustado
Expanded(
  child: Center(
    child: SizedBox(
      width: 300,
      height: 380,
      child: Stack(
        alignment: Alignment.center,
        children: [
          // Botón central: Pedir Cita con imagen
          Positioned(
            left: 80,
            top: 90,
            child: _circleMenuButton(
              imagePath: "lib/assets/icon_pedirCita.png", // Ruta a tu imagen
              label: "Pedir Cita",
              onTap: () {
                // Acción de navegación
                Navigator.push(
                  context,
                  MaterialPageRoute(builder: (context) => Pedircita()),
                );
              },
              size: 130,
              fontSize: 20,
              color: Colors.white,
              elevation: 8,
              isMain: true,
            ),
          // Positioned
          // Cesta con imagen
          _positionedCircleCustom(
            angle: 180,
            distance: 65,
```

Figura 25: Código MenuInicio

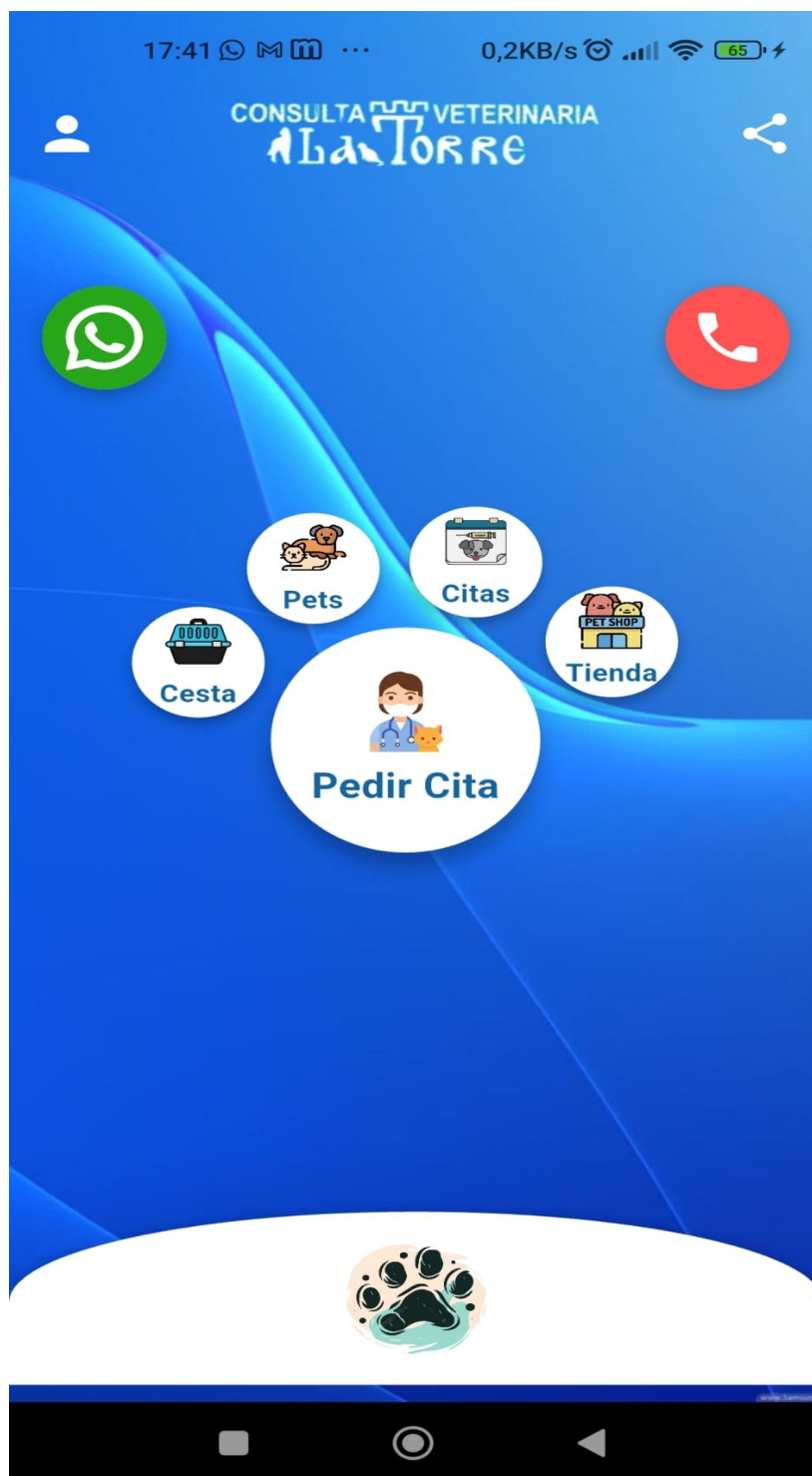


Figura 26: MenuInicio

4.4.3 Diseño Citas

Pantalla Pedir Cita

Se diseña una pantalla donde a través de un selector de espacio entre consulta y peluquería se pueda reservar una cita. Abajo se mostrara un calendario, del paquete table_calendar para seleccionar el día y abajo una lista de las horas disponibles.

La estética del selector de espacio la logramos de modo manual, ya que los widgets que había por defecto no me terminaron de gustar en un container con bordes redondeados y dos botones dentro.

El calendario se implementa a través de la librería. Adjunto código:

```
// --- Calendario para seleccionar la fecha ---
Container(
  decoration: BoxDecoration(
    color: Colors.white,
    borderRadius: BorderRadius.circular(18),
  ), // BoxDecoration
  child: TableCalendar(
    //hacemos que el día seleccionado se guarde en la variable
    focusedDay: _fechaSeleccionada,
    firstDay: DateTime.now(), //fecha en la que se inicia el calendario
    lastDay: DateTime.now().add(const Duration(days: 30)), //ultimo día seleccionable
    calendarFormat: CalendarFormat.month, //formato calendario
    onDaySelected: (selectedDay, _) {
      //cambiamos el estado de la variable cada vez que se cambia de día
      setState(() {
        _fechaSeleccionada = selectedDay;
      });
      // Recarga las horas según día y espacio seleccionado
      cargarHorasOcupadas();
    },
    selectedDayPredicate: (day) {
      return isSameDay(_fechaSeleccionada, day);
    },
    calendarStyle: const CalendarStyle(
      selectedDecoration: BoxDecoration(
        color: Colors.lightBlueAccent,
        shape: BoxShape.circle,
      ), // BoxDecoration
      //decoración del día seleccionado
      todayDecoration: BoxDecoration(
        color: Colors.lightBlueAccent,
        shape: BoxShape.circle,
      ), // BoxDecoration
    ), // CalendarStyle
  ), // TableCalendar
), // Container
```

Figura 27: Código Calendario

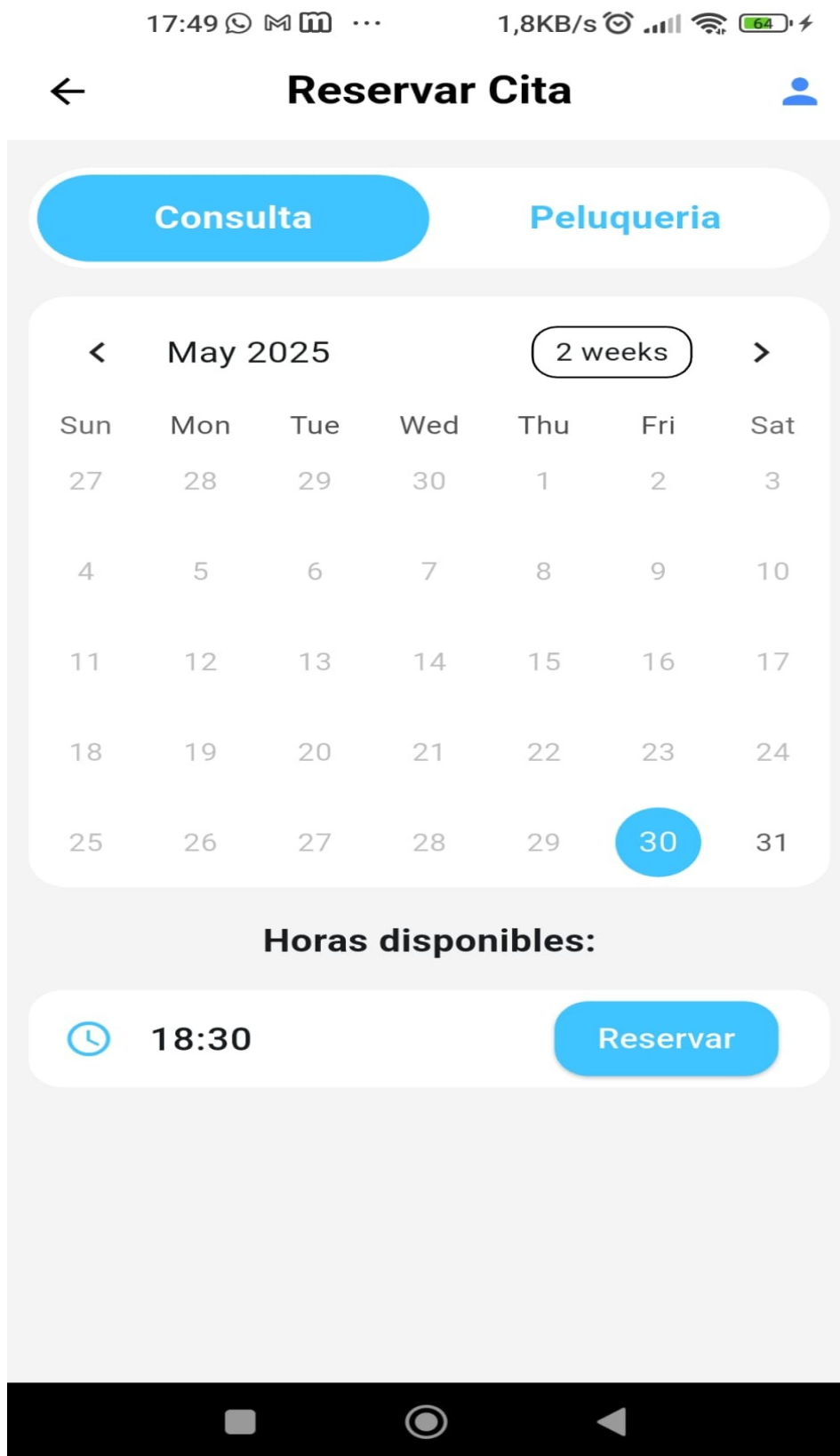


Figura 28: ReservarCita

Pantalla Mis citas

En esta pantalla se listarán las próximas citas que el usuario ha cogido para sus mascotas. Para ello se ha creado un Widget en un archivo independiente para reutilizar. Este recibirá los datos de la cita y los mostrará.

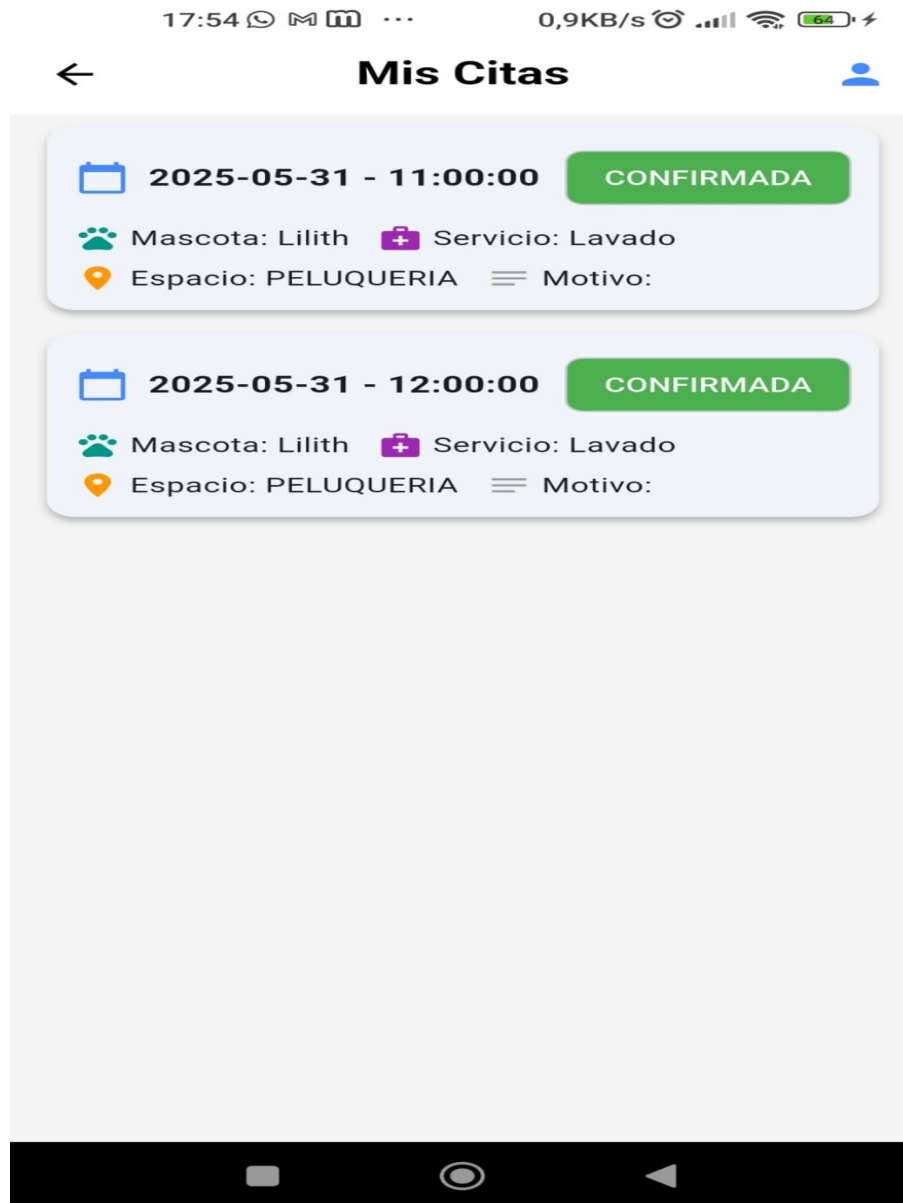


Figura 29: Mis Citas

4.4.4 Diseño pantalla mascotas

Pantalla Mascotas

Aquí se mostrarán las mascotas del usuario listadas, cada mascota tendrá una tarjeta donde se muestren sus datos, junto a un botón editar(para editar sus datos) y otro de eliminar. Abajo habrá un botón para añadir nuevas mascotas.

La pantalla para agregar mascotas se tratará de un formulario donde el usuario podrá subir una imagen de la mascota y su información. Abajo se muestra como se ven.

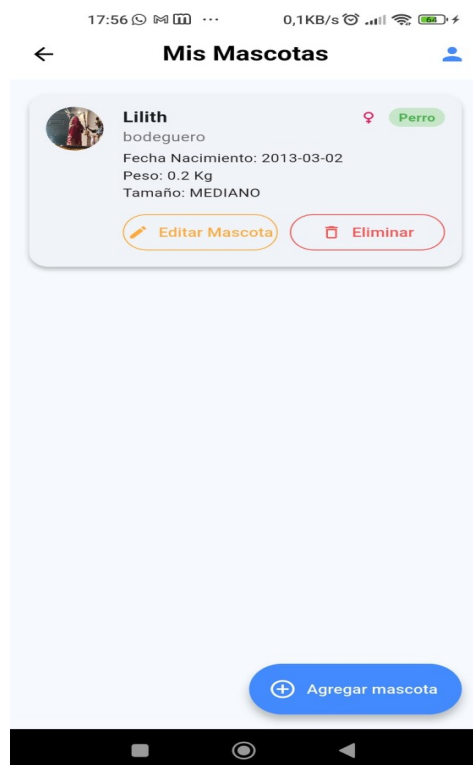


Figura 30: Mis mascotas



Figura 31: Registro Mascotas

4.4.5 Diseño Tienda

En esta pantalla diseñaremos el aspecto visual de la tienda y sus filtros. Contaremos con dos filas de filtros, en las que se podrá filtrar los productos en base a 1 criterio específico como una especie, marca o categoría. Abajo tendremos un buscador para buscar en base a 2 criterios. Y debajo una sección de productos destacados donde se mostrará una tarjeta del producto con su información e imagen. Abajo a la derecha también se incorpora un botón flotante para acceder al carrito.



Figura 32: Tienda

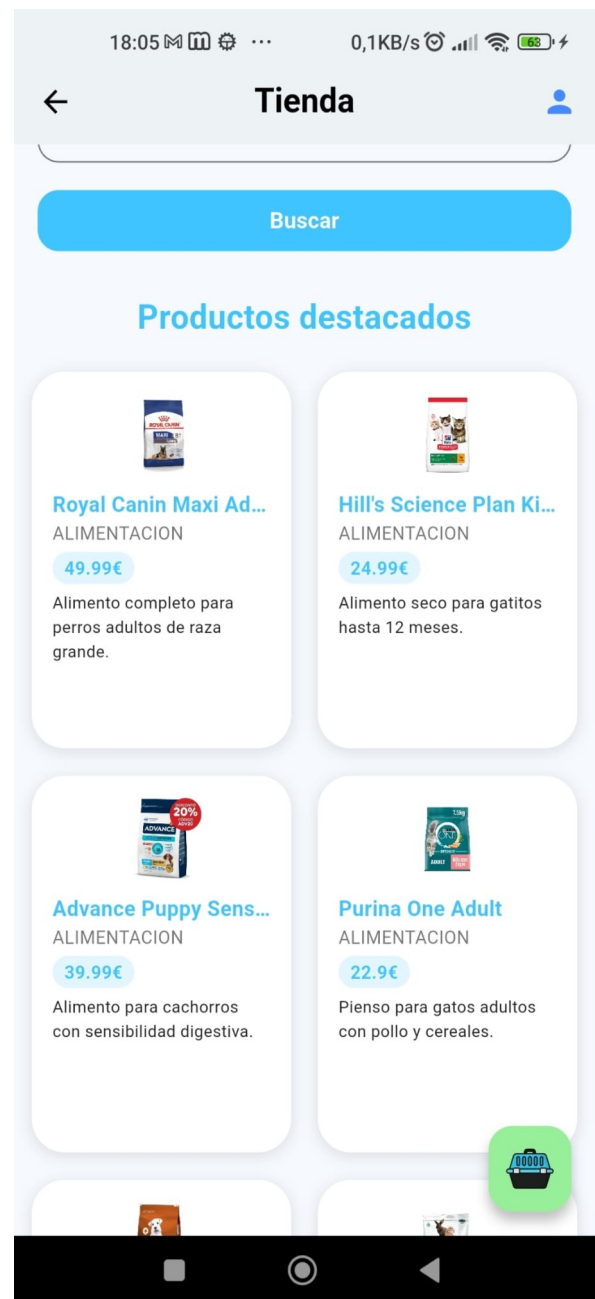


Figura 33: Productos

4.5. Iteración 4 - Fase IV: Implementación de la Autenticación y Gestión de Usuarios

4.5.1. Implementación Spring Security

Implementamos la librería Spring Security para proteger el acceso a los endpoints. Esta librería automáticamente protege todos los endpoints exigiendo una autenticación para acceder a ellos. Algunos endpoints como login y Sign Up deben ser accesibles para todos los usuarios. Para gestionar esto tendremos que crear un archivo para configuración de seguridad. Y dar acceso a los endpoints necesarios. Adjunto imagen de mi archivo SpringSecurity:

```
13
14 @Configuration
15 public class SecurityConfig {
16
17     @Bean
18     public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
19         http
20             .csrf().disable()
21             .authorizeHttpRequests(auth -> auth
22                 // RUTAS PUBLICAS (acceso sin autenticación)
23                 // Registro y login
24                 .requestMatchers(HttpMethod.POST, ...patterns: "/usuarios", "/usuarios/login").permitAll()
25                 // Mascotas
26                 .requestMatchers(HttpMethod.POST, ...patterns: "/mascotas", "/mascotas/*imagen").permitAll()
27                 .requestMatchers(HttpMethod.GET, ...patterns: "/mascotas/buscar").permitAll()
28                 // Citas
29                 .requestMatchers(HttpMethod.POST, ...patterns: "/citas").permitAll()
30                 .requestMatchers(HttpMethod.GET, ...patterns: "/citas/ocupadas", "/citas/usuario/*", "/citas/usuario/*proximas", "/citas/usuario/*historial").permitAll()
31                 .requestMatchers(HttpMethod.PATCH, ...patterns: "/citas/*", "/mascotas/*").permitAll()
32                 // Servicios y productos
33                 .requestMatchers(HttpMethod.GET, ...patterns: "/servicios", "/servicios/*", "/productos").permitAll()
34                 // Carrito
35                 .requestMatchers(HttpMethod.GET, ...patterns: "/carrito/*").permitAll()
36                 .requestMatchers(HttpMethod.POST, ...patterns: "/carrito/*").permitAll()
37                 .requestMatchers(HttpMethod.PUT, ...patterns: "/carrito/*", "/usuarios/*").permitAll()
38                 .requestMatchers(HttpMethod.DELETE, ...patterns: "/carrito/*", "/mascotas/*").permitAll()
39                 // Imágenes
40                 .requestMatchers(...patterns: "/*.jpg", "/*.png", "/*.jpeg").permitAll()
41                 // Usuarios (GET ya está permitido arriba, aquí solo para orden)
42                 .requestMatchers(HttpMethod.GET, ...patterns: "/usuarios/*").permitAll()
43
44                 // RUTAS PROTEGIDAS (requieren autenticación y rol ADMIN)
45                 // Todas las demás rutas requieren autenticación (y solo ADMIN puede autenticarse)
46                 .anyRequest().authenticated()
47             )
48             .httpBasic(); // Autenticación básica
49
50         return http.build();
51     }
52 }
```

Figura 34: Configuración Security

```
//usuario que tiene autorización básica, para Postman
@Bean
public InMemoryUserDetailsService userDetailsService() {
    UserDetails user = User.builder()
        .username(username:"root")
        .password(passwordEncoder().encode(rawPassword:"1234"))
        .roles(...roles:"ADMIN")
        .build();
    return new InMemoryUserDetailsService(user);
}
```

Figura 35: Autorización Básica

4.5.2 Implementación SignUp

Se crean los controllers que obtendrán los datos de los textField, estos se le pasarán al body del endpoint POST para usuarios que ya teníamos creado. Para. Adjunto imagen del metodo:

```
Future<void> registrarUsuario() async {
  final url = Uri.parse('http://192.168.1.131:8080/usuarios'); // Cambia la URL si usas dispositivo físico

  final response = await http.post(
    url,
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode({
      "dni": dniController.text,
      "nombre": nombreController.text,
      "apellidos": apellidosController.text,
      "email": emailController.text,
      "telefono": telefonoController.text,
      "contrasena": contrasenaController.text,
    })),
  );

  if (response.statusCode == 201) {
    // Usuario creado correctamente
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('¡Usuario registrado con éxito!'))
    );
    // Posible acción de navegación de vuelta al Login(por ahora ha fallado)
  } else {
    // Error al registrar
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Error: ${response.statusCode} - ${response.body}'))
    );
  }
}
```

Figura 36: registrar Usuario

4.5.3 Implementación Login

Para implementar esta función creamos un nuevo endpoint para comprobar el dni y contraseña introducidas con las de la BBDD. Además se introduce por primera vez el paquete `shared_preferences` que almacena información en variables de forma local, en este caso, nos será útil para guardar el DNI y rol del usuario logueado, para poder utilizar las demás funciones de la app. Adjunto imagen del código:

```
//Este metodo hace que si el login es correcto guarde, guardemos el dni y rol del usuario en sharedPreferences,
//obteniendolos a través de una consulta GET a la BBDD
Future<void> loginUsuario() async {
  //funcion de login
  final url = Uri.parse('http://192.168.1.131:8080/usuarios/login');
  final response = await http.post(
    url,
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode({
      "dni": usuarioController.text,
      "contrasena": contrasenaController.text,
    })),
  );
  //si login es exitoso
  if (response.statusCode == 200) {
    // Login correcto: ahora obtenemos la info del usuario desde la bbdd
    //(podria bastar con guardar el dni desde el textfield pero como tambien hace falta el rol, cogemos los dos desde laBBDD y ya esta)
    final dni = usuarioController.text.trim();
    final infoUrl = Uri.parse('http://192.168.1.131:8080/usuarios/$dni');
    final infoResponse = await http.get(infoUrl);

    if (infoResponse.statusCode == 200) {
      final data = jsonDecode(infoResponse.body);
      final prefs = await SharedPreferences.getInstance();
      await prefs.setString('dni_usuario', data['dni']); //guarda dni
      await prefs.setString('rol_usuario', data['rol']); // Guarda el rol

      // Ahora puedes navegar según el rol si quieres:
      Navigator.pushReplacement(
        context,
        MaterialPageRoute(builder: (context) => Inicio()),
      );
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('No se pudo obtener la información del usuario')),
      );
    }
  } else {
    // Login incorrecto
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Error: ${response.statusCode} - ${response.body}')),
    );
  }
}
```

Figura 37: implementacion endpoint login

4.6. Iteración 5 - Fase V: Gestión de Mascotas y Citas

4.6.1 Listar mascotas del usuario

Para listar las mascotas primero deberemos obtener el dni del usuario del almacenamiento de shared_preferences, esto se hace a través de un método muy sencillo que emplearemos a lo largo del proyecto. A continuación se creará un nuevo endpoint para buscar las mascotas según el dni de su dueño. GET mascotas/buscar/{dni}. A continuación adjunto una imagen con el código en el frontend de estos métodos comentado:

```
64 // Función que hace la petición HTTP al backend para obtener las mascotas del usuario
65 // Recibe el dni o email y devuelve una lista de objetos GetMascotaDto
66 Future<List<GetMascotaDto>> fetchMascotasUsuario(String dniOrEmail) async {
67   final url = Uri.parse('http://192.168.1.131:8080/mascotas/buscar?dniOrEmail=$dniOrEmail');
68   final response = await http.get(url);
69
70   // Imprime en consola el código de estado y el cuerpo de la respuesta para depuración
71   print('Mascotas API status: ${response.statusCode}');
72   print('Mascotas API body: ${response.body}');
73
74   if (response.statusCode == 200) {
75     // Decodifica el JSON y convierte cada elemento en un objeto GetMascotaDto
76     final list<dynamic> data = jsonDecode(response.body);
77     return data.map((json) => GetMascotaDto.fromJson(json)).toList();
78   } else {
79     // Si ocurre un error, lanza una excepción con información detallada
80     throw Exception('Error al cargar mascotas: ${response.statusCode} - ${response.body}');
81   }
82 }
83
84 // Método asíncrono que obtiene el dni/email del usuario guardado en SharedPreferences
85 // y luego hace una petición HTTP para obtener sus mascotas.
86 Future<void> cargarMascotasDelUsuario() async {
87   try {
88     // Recupera el dni o email del usuario logueado
89     final dniOrEmail = await obtenerDniUsuario();
90
91     // Si no hay usuario logueado, muestra un mensaje de error
92     if (dniOrEmail == null) {
93       setState(() {
94         errorMsg = 'No hay usuario logueado';
95         isLoading = false;
96       });
97       return;
98     }
99
100    // Llama al método que obtiene la lista de mascotas del backend
101    final resultado = await fetchMascotasUsuario(dniOrEmail);
102
103    // Actualiza el estado con la lista de mascotas y oculta el loader
104    setState(() {
105      mascotas = resultado;
106      isLoading = false;
107    });
108  } catch (e) {
109    /** Si ocurre un error (de red, backend, parsing, etc.), muestra mensaje de error
110    setState(() {
111      errorMsg = 'Error al cargar mascotas: $e';
112      isLoading = false;
113    });*/
114  }
115 }
116
117 // Función que obtiene el dni o email del usuario logueado desde SharedPreferences
118 Future<String?> obtenerDniUsuario() async {
119   final prefs = await SharedPreferences.getInstance();
120   return prefs.getString('dni_usuario'); // Cambia la clave si usas otro nombre
121 }
122
```

Figura 38: fetch mascotasUsuario

4.6.2 Añadir mascotas

En este apartado se presenta una nueva complicación ya que al registrar una mascota el usuario debe tener la posibilidad de subir una foto(es opcional). La solución por la que optamos será por hacer 2 endpoints de tipo POST, uno para subir los datos de la mascota y otro para la imagen. Adjunto imágenes del código de los endpoints con comentarios:

```
// Crear una nueva mascota
@PostMapping
public ResponseEntity<GetMascotaDto> createMascota(@RequestBody CreateMascotaDto dto) {
    Usuario usuario = usuarioRepository.findByDni(dto.getUsuarioDni())
        .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, reason:"Usuario no encontrado"));
    Mascota nuevaMascota = mascotaService.createMascota(dto, usuario);
    return ResponseEntity.status(HttpStatus.CREATED)
        .body(mascotaService.getMascotaById(nuevaMascota.getId()).orElse(other:null));
}

//subir una imagen de la mascota
@PostMapping(value = "{id}/imagen", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public ResponseEntity<GetMascotaDto> subirImagenMascota(
    @PathVariable Long id,
    @RequestParam("imagen") MultipartFile imagen) throws IOException {

    // Verifica si la mascota existe
    Optional<Mascota> mascotaOpt = mascotaRepository.findById(id);
    if (mascotaOpt.isEmpty()) {
        return ResponseEntity.notFound().build();
    }

    // Guarda la imagen y obtén la ruta
    String imageUrl = mascotaService.guardarImagen(id, imagen);

    // Actualiza la mascota con la ruta de la imagen
    Mascota mascota = mascotaOpt.get();
    mascota.setImagenUrl(imageUrl);
    mascotaRepository.save(mascota);

    return ResponseEntity.ok(mascotaMapper.toGetMascotaDto(mascota));
}
```

Figura 39: endpoints subirImagen

```
// Crear una nueva mascota
public Mascota createMascota(CreateMascotaDto dto, Usuario usuario) {
    Mascota nuevaMascota = mascotaMapper.toMascota(dto, usuario);
    return mascotaRepository.save(nuevaMascota);
}

// Define la carpeta donde se guardarán las imágenes (ajusta la ruta según tu SO)
private static final String UPLOAD_DIR = "C:/Users/david/Desktop/2DAM/PROYECTO-CFGS/Proyecto - Codigo/imagenesMascotas/";
//metodo para guardar la imagen
public String guardarImagen(Long mascotaId, MultipartFile imagen) throws IOException {
    // Crea la carpeta si no existe
    Path uploadPath = Paths.get(UPLOAD_DIR);
    if (!Files.exists(uploadPath)) {
        Files.createDirectories(uploadPath);
    }

    // Genera un nombre único para el archivo, con su ID y un UUID
    String extension = imagen.getOriginalFilename().split(regex:"\\.")[1];
    String nombreArchivo = mascotaId + "_" + UUID.randomUUID() + "." + extension;

    // Guarda el archivo
    Path filePath = uploadPath.resolve(nombreArchivo);
    Files.copy(imagen.getInputStream(), filePath);

    // Retorna la ruta relativa (ej: "uploads/mascotas/1_abc123.jpg")
    return nombreArchivo;
}
```

Figura 40: CrearMascota

Al probar el endpoint para subir las imágenes surge una **problemática** y es que al intentarlo da error y no funciona, probando con varias me doy cuenta que un par de ellas si me deja y otras no. Investigando resulta que este error podría ser causado porque **springboot** en principio **tiene limitado el tamaño máximo de archivo que se puede subir** en 1 MB por lo que **tenemos que aumentarlo**, esto se **modifica en el archivo application.properties**. También para que funcione tendremos que **permitir al backend servir archivos estáticos** y poner la ruta de las carpetas donde se almacenarán. Adjunto imagen del código de nuestra application.properties para que quede más claro.

```
Proyecto - Codigo > clinica SPRING boot > demo > src > main > resources > application.properties
1 | #Gestión BBDD
2 | spring.application.name=demo
3 | spring.datasource.url=jdbc:mysql://localhost:3306/Clinica
4 | spring.datasource.username=root
5 | spring.datasource.password=1234
6 | spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
7 | #usuario y contraseña para poder acceder a endpoints
8 | spring.security.user.name=root
9 | spring.security.user.password=1234
10 | #para que la API sea accesible externamente
11 | server.address=0.0.0.0
12 | # Permite servir archivos estáticos desde las carpetas 'imagenesMascotas' e imagenesProductos
13 | spring.web.resources.static-locations=file:/C:/Users/david/Desktop/2DAM/PROYECTO-CFGS/Proyecto - Codigo/imagenesMascotas/,file:./imagenesProductos/
14 |
15 |
16 | #aumentar el tamaño de los archivos que se pueden subir
17 | spring.servlet.multipart.max-file-size=50MB
18 | spring.servlet.multipart.max-request-size=50MB
19 | # JPA (Para que cree las tablas automáticamente)
20 | spring.jpa.hibernate.ddl-auto=update
21 | spring.jpa.show-sql=true
22 | spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
23 |
```

Figura 41: application.properties

4.6.3 Editar y eliminar mascotas

Para editar las mascotas se reutiliza el formulario para crear las mascotas, en él se cargan los datos de la mascota seleccionada, se bloquean algunos campos que se consideran inalterables como la raza, especie o sexo y se realiza el update con un endpoint PATCH sobre el resto de datos. Para obtener la imagen de la mascota se hace a través de un nuevo endpoint GET, si el usuario actualiza la imagen, al actualizar esta se subirá al servidor a través del mismo endpoint que al añadir una nueva mascota.

Para eliminar una mascota utilizaremos el endpoint DELETE por id creado al inicio del backend. Al eliminar una mascota se eliminarán todas las citas que había programadas para ella.

4.6.4 Pedir Cita

Incorporamos funcionalidad al botón reservar, al hacer click en él nos abrirá una ventana emergente(Show Dialog), en el nos dejará elegir los diferentes servicios que se ofrecen en ese espacio consulta o peluquería en un dropdown, abajo en otro dropdown nos dejará elegir entre las mascotas del usuario y abajo finalmente podrán poner una breve descripción o algún motivo de la cita. Código de la ventana emergente:

```

    ), // RoundedRectangleBorder
  ),
  onPressed: () async {
    try{

      final horaSeleccionada = horasLibres[index]; //Variable para la hora Ej: "10:00"
      final fecha = "${_fechaSeleccionada.day.toString().padLeft(2, '0')}/${_fechaSeleccionada.month.toString().padLeft(2, '0')}/${_fechaSeleccionada.year}"; // Formato dd/MM/yyyy

      // Obtener servicio seleccionado
      final servicio = serviciosDisponibles.firstWhere(
        (s) => s['nombre'] == servicioSeleccionado,
        orElse: () => {},
      );

      // Obtener mascota seleccionada
      final mascota = mascotasUsuario.firstWhere(
        (m) => m['nombre'] == mascotaSeleccionada,
        orElse: () => {},
      );

      final prefs = await SharedPreferences.getInstance();
      final usuarioDni = prefs.getString('dni_usuario');

      if (servicio.isEmpty || mascota.isEmpty || usuarioDni == null) {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('Complete todos los campos')),
        );
        return;
      }

      final body = jsonEncode({
        'fecha': fecha,
        'hora': horaSeleccionada,
        'espacio': _selectedEspacio == 0 ? 'CONSULTA' : 'PELUQUERIA',
        'motivo': motivo,
        'estado': 'CONFIRMADA',
        'mascotaId': mascota['id'],
        'usuarioDni': usuarioDni,
        'servicioId': servicio['id'],
      });

      final response = await http.post(
        Uri.parse('http://192.168.1.131:8080/citas'),
        headers: {'Content-Type': 'application/json'},
        body: body,
      );
    }
  }
}

```

Figura 42: Implementacion POST crear cita

A continuación adjunto el código de algunos de los endpoints claves:

```
// Obtener todos los servicios o filtrar por espacioServicio
@GetMapping
public List<GetServicioDto> getServicios(@RequestParam(required = false) String espacioServicio) {
    if (espacioServicio != null) {
        Espacio espacio = Espacio.valueOf(espacioServicio.toUpperCase());
        return servicioService.getServiciosByEspacio(espacio);
    }
    return servicioService.getAllServicios();
}
```

Figura 43: Obtener Servicios

```
//obtiene las horas ocupadas por citas confirmadas
@GetMapping("/ocupadas")
public List<String> getHorasOcupadas(
    @RequestParam String fecha,
    @RequestParam String espacio
) {
    LocalDate dia = LocalDate.parse(fecha, DateTimeFormatter.ofPattern(pattern:"dd/MM/yyyy"));
    //horas que empieza y acaba el dia
    LocalDateTime start = dia.atStartOfDay();
    LocalDateTime end = dia.atTime(LocalTime.MAX);
    //obtener espacio
    Cita.Espacio espacioEnum = Cita.Espacio.valueOf(espacio.trim().toUpperCase());
    //IMPORTANTE: ponemos el estado de la cita por defecto en CONFIRMADA
    List<Cita> citas = citaRepository.findByEspacioAndEstadoAndFechaCitaBetween(
        espacioEnum, Cita.EstadoCita.CONFIRMADA, start, end
    );
    return citas.stream()
        .map(c -> c.getFechaCita().toLocalTime().toString().substring(beginIndex:0,endIndex:5))
        .toList();
}
```

Figura 44: Horas ocupadas

En el front tendremos un metodo loadServicios() y otro cargarHorasOcupadas(), importante llamar a estos cada vez que se cambie el espacio seleccionado entre Consulta o peluquería.

4.6.5 Visualizar citas del usuario

Pantalla Mis Citas

En esta pantalla el usuario podrá visualizar sus citas confirmadas listadas y debe poder cancelarlas. Para ello se implementa un endpoint `cita/usuario/dni/proximas`, filtra las citas del usuario por fecha (muestra las superiores a la de este momento) y que tengan el estado de 'CONFIRMADA'.

Para añadir la función de poder cancelar cita utilizamos el PATCH creado inicialmente con un body cambiando la variable estado a 'CANCELADA'. Adjunto imagen de la implementación de los métodos en el front:

```
class CitasScreenState extends State<CitasScreen> {
    //Recupera el DNI del usuario logueado desde el almacenamiento local para identificarlo en el backend.
    Future<String?> obtenerDniOEmailUsuario() async {
        final prefs = await SharedPreferences.getInstance();
        return prefs.getString('dni_usuario');
    }

    //Petición get para obtener las citas del usuario
    Future<List<Map<String, dynamic>>> obtenerCitasUsuario(String dni) async {
        final url = 'http://192.168.1.131:8080/citas/usuario/$dni/proximas';
        final response = await http.get(Uri.parse(url));
        if (response.statusCode == 200) {
            final List<dynamic> data = jsonDecode(response.body);
            return data.cast<Map<String, dynamic>>();
        } else {
            throw Exception('Error al cargar las citas');
        }
    }

    //petición patch para cambiar el estado de una cita
    Future<bool> cancelarCita(int idCita) async {
        final url = 'http://192.168.1.131:8080/citas/$idCita';
        final response = await http.patch(
            Uri.parse(url),
            headers: {'Content-Type': 'application/json'},
            body: jsonEncode({'estado': 'CANCELADA'}),
        );
        return response.statusCode == 200;
    }
}
```

Figura 45: implementación mis citas

4.7. Iteración 6 - Fase VI: Desarrollo de la Tienda y Carrito de Compras

4.7.1 Pantalla Tienda

Listado de productos

En la sección de productos destacados listamos todos los productos de la tienda. Para ello utilizamos el **endpoint GET de productos creado inicialmente** y cargamos los productos.

Filtros

Para el **filtrado de productos en base a un criterio** nos vale con modificar el get de productos inicial de la siguiente manera:

Modificaremos el endpoint para que requiera tres parametros especie, categoria y marca y en el service tendremos una serie de condicionales que manejarán los casos posibles. Si por ejemplo, todos los parámetros fueran null obtiene todos los productos al igual que anteriormente. Adjunto imagen del service:

```
Proyecto - Codigo > clinica SPRING boot > demo > src > main > java > com > example > clinica > services > J ProductoService.java > Produ
23 public class ProductoService {
87     public List<GetProductoDto> findByEspecieCategoriaMarca(
88         Producto.EspecieProducto especie,
89         Producto.CategoriaProducto categoria,
90         String marca
91     ) {
92         List<Producto> productos;
93
94         // Caso 1: Solo especie
95         if (especie != null && categoria == null && marca == null) {
96             productos = productoRepository.findByEspeciesContaining(especie);
97         }
98         // Caso 2: Solo categoria
99         else if (categoria != null && especie == null && marca == null) {
100             productos = productoRepository.findByCategoria(categoria);
101         }
102         // Caso 3: Solo marca
103         else if (marca != null && especie == null && categoria == null) {
104             productos = productoRepository.findByMarca(marca);
105         }
106         // Caso 4: Especie y categoria
107         else if (especie != null && categoria != null && marca == null) {
108             productos = productoRepository.findByEspeciesContainingAndCategoria(especie, categoria);
109         }
110         // Caso 5: Especie y marca
111         else if (especie != null && marca != null && categoria == null) {
112             productos = productoRepository.findByEspeciesContaining(especie)
113                 .stream()
114                 .filter(p -> p.getMarca().equalsIgnoreCase(marca))
115                 .collect(Collectors.toList());
116         }
117         // Caso 6: Categoria y marca
118         else if (categoria != null && marca != null && especie == null) {
119             productos = productoRepository.findByCategoria(categoria)
120                 .stream()
121                 .filter(p -> p.getMarca().equalsIgnoreCase(marca))
122                 .collect(Collectors.toList());
123         }
124         // Caso 7: Especie, categoria y marca
125         else if (especie != null && categoria != null && marca != null) {
126             productos = productoRepository.findByEspeciesContainingAndCategoria(especie, categoria)
127                 .stream()
128                 .filter(p -> p.getMarca().equalsIgnoreCase(marca))
129                 .collect(Collectors.toList());
130         }
131         // Caso 8: Sin filtros
132         else {
133             productos = productoRepository.findAll();
134         }
135
136         return productos.stream()
137             .map(productoMapper::toGetProductoDto)
138             .collect(Collectors.toList());
139     }
}
```

Para el **filtrado de productos en base a 2 criterios** donde podremos seleccionar una especie y una categoría para filtrar, **se utiliza el mismo endpoint anterior**, correspondiéndole el caso 5 de nuestro condicional donde categoría y especie son diferentes de null. En los condicionales vemos como para filtrar se usan **metodos del repositorio de productos**, estos metodos para filtrar los productos son **generados por JPA**. Adjunto imagen del codigo de ProductoRepository:

```
@Repository
public interface ProductoRepository extends JpaRepository<Producto, Long> {
    // filtro para buscar por especies, con containing, JPA busca dentro de listas
    List<Producto> findByEspeciesContaining(EspecieProducto especies);
    List<Producto> findByCategoria(Producto.CategoriaProducto categoria);
    List<Producto> findByEspeciesContainingAndCategoria(Producto.EspecieProducto especie, Producto.CategoriaProducto categoria);
    List<Producto> findByMarca(String marca);
}
```

Figura 47: Producto Repository

4.7.2 Carrito

Creamos endpoints para carrito

El carrito de un usuario constará de líneas de carrito, estas contienen el producto añadido, su correspondiente n.º de unidades y un boolean para saber si este está seleccionado o no (por defecto pondremos que si). Para añadir un producto al carrito desde la pantalla de detalle de un producto se utiliza el siguiente endpoint:

```
// Añadir una línea al carrito
@PostMapping("/{usuarioDni}/linea")
public ResponseEntity<GetLineaCarritoDto> addLineaCarrito(
    @PathVariable String usuarioDni,
    @RequestBody CreateLineaCarritoDto dto) {
    // Buscar el carrito o crearlo si no existe
    Carrito carrito = carritoRepository.findByUsuarioDni(usuarioDni)
        .orElseGet(() -> {
            Usuario usuario = usuarioRepository.findById(usuarioDni)
                .orElseThrow(() -> new RuntimeException(message:"Usuario no encontrado"));
            Carrito nuevoCarrito = new Carrito();
            nuevoCarrito.setUsuario(usuario);
            return carritoRepository.save(nuevoCarrito);
        });

    // Buscar el producto
    Producto producto = productoRepository.findById(dto.getProductoId())
        .orElseThrow(() -> new RuntimeException(message:"Producto no encontrado"));

    // Buscar si ya existe una línea para ese producto en el carrito
    Optional<LineaCarrito> lineaExistente = lineaCarritoRepository.findByCarritoIdAndProductoId(
        carrito.getId(), producto.getId());

    LineaCarrito linea;
    if (lineaExistente.isPresent()) {
        // Si ya existe, actualizar la cantidad
        linea = lineaExistente.get();
        linea.setCantidad(linea.getCantidad() + dto.getCantidad());
    } else {
        // Si no existe, crear una nueva línea
        linea = LineaCarritoMapper.toEntity(dto, carrito, producto);
    }
    linea = lineaCarritoRepository.save(linea);

    GetLineaCarritoDto lineaDto = LineaCarritoMapper.toGetLineaCarritoDto(linea);
    return ResponseEntity.ok(lineaDto);
}
```

Figura 48: código añadir carrito

Pantalla Carrito

Se crea la pantalla del carrito, se mostrarán los productos que añada el usuario listados, **podrás modificar el n.º de unidades y eliminar o comprar los productos seleccionados**. Imagen pantalla carrito:

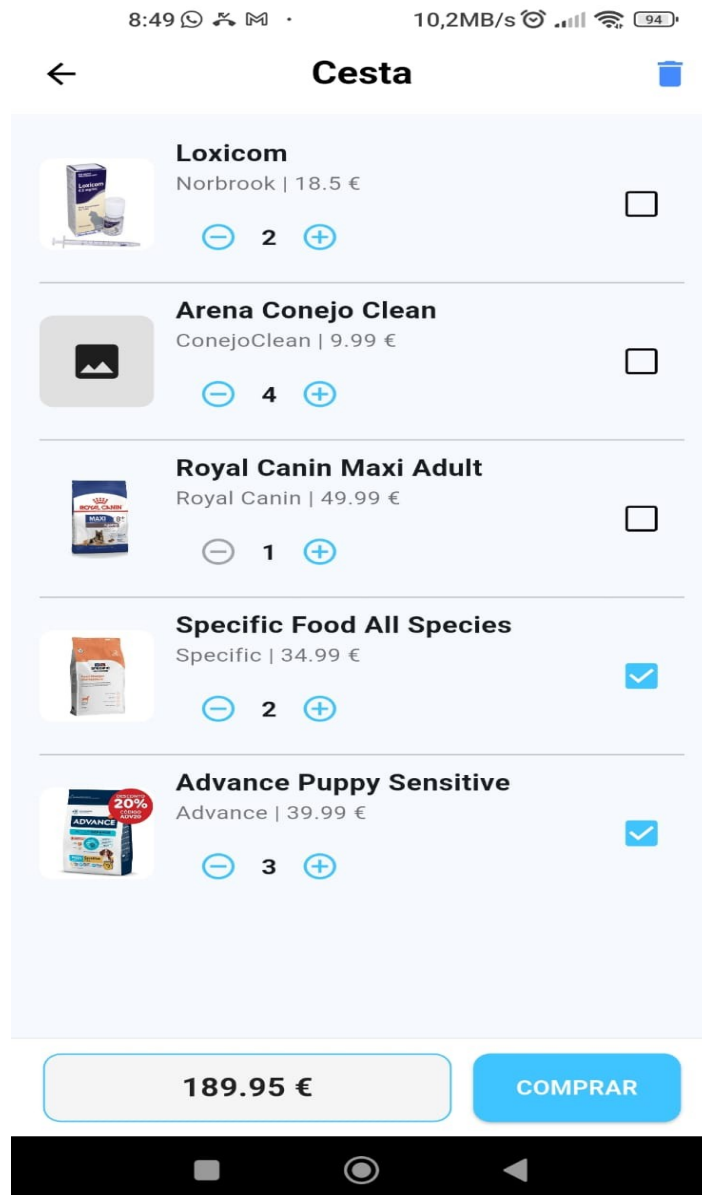


Figura 49: cesta

4.8. Iteración 7 - Fase VII: Funcionalidades de Perfil de Usuario

4.8.1 Diseño pantalla perfil de usuario

Se plantea un diseño simple, fácil e intuitivo para el usuario, constará de 2 columnas y 5 filas en las que se encontrará un icono y debajo el nombre de su apartado. A esta sección se podrá acceder desde los iconos de persona de todas las appBar. Imagen del resultado:

FIGURA 4.43. PerfilUsuario

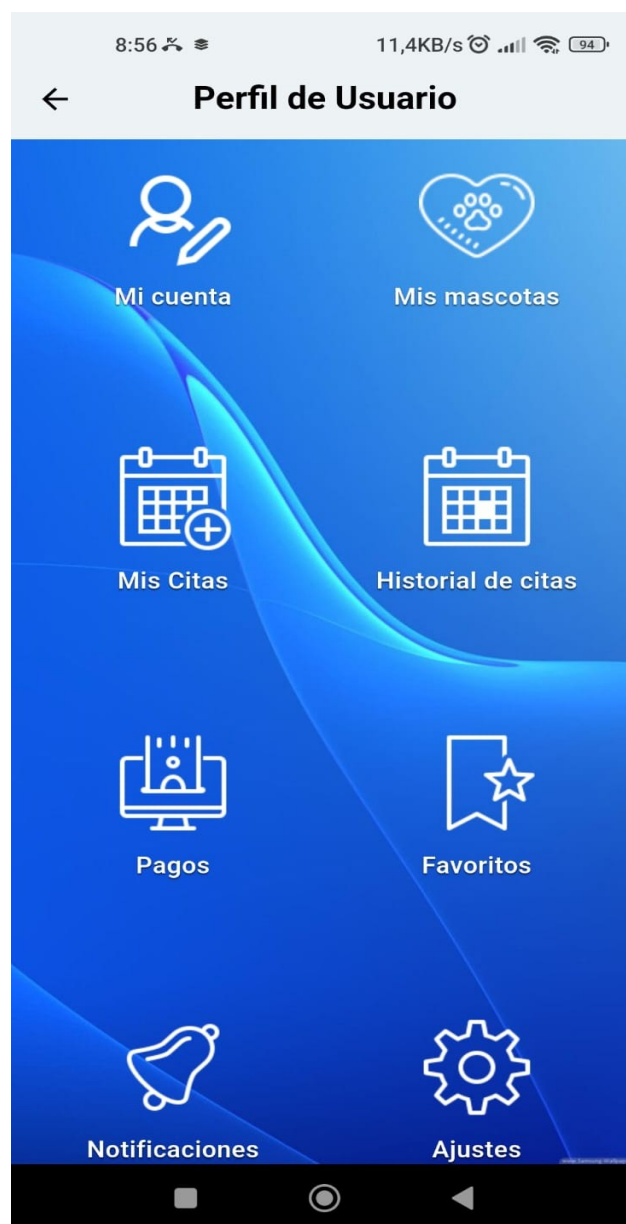


Figura 50: perfil usuario

4.8.2 Funcionalidades añadidas

Editar perfil

Se crea un formulario cargando los datos del usuario, donde poder editar los campos email, teléfono y contraseña. Si la contraseña es nula se mantiene la actual. Imagen del método actualizarUsuario:

```
68  Future<void> actualizarUsuario() async {
69      final dni = dniController.text;
70      final url = Uri.parse('http://192.168.1.131:8080/usuarios/$dni');
71      final body = {
72          "dni": dni,
73          "nombre": nombreController.text,
74          "apellidos": apellidosController.text,
75          "email": emailController.text,
76          "telefono": telefonoController.text,
77          //si la contraseña es nula, se mantiene la actual
78          "contrasena": contrasenaController.text.isNotEmpty
79              ? contrasenaController.text
80              : null, // Solo si quiere cambiarla
81      }..removeWhere((k, v) => v == null); // Elimina campos nulos
82
83      final response = await http.put(
84          url,
85          headers: {'Content-Type': 'application/json'},
86          body: jsonEncode(body),
87      );
88      if (response.statusCode == 200) {
89          ScaffoldMessenger.of(context).showSnackBar(
90              const SnackBar(content: Text('Datos actualizados correctamente')),
91          );
92          Navigator.pop(context, true); // Vuelve atrás e indica éxito
93      } else {
94          ScaffoldMessenger.of(context).showSnackBar(
95              SnackBar(content: Text('Error al actualizar: ${response.body}')),
96          );
97      }
98  }
```

Figura 51: Actualizar Usuario

Historial de citas

Se muestran las citas listadas de la misma manera que en la pantalla mis citas, solo cambia el endpoint, para ello realizamos la siguiente query a la BBDD desde el repositorio de citas.

Adjunto imagen codigo CitaRepository.java:

```
14 @Repository
15 public interface CitaRepository extends JpaRepository<Cita, Long> {
16     List<Cita> findByMascotaId(Long mascotaId); // Obtener citas de una mascota
17     List<Cita> findByEspacio(String espacio);
18     List<Cita> findByServicio(Servicio servicio);
19     // Busca citas en el mismo espacio, fecha y hora
20     boolean existsByEspacioAndEstadoAndFechaCita(Cita.Espacio espacio, Cita.EstadoCita estado, LocalDateTime fechaCita);
21     //Lista de horas ocupadas en una fecha, espacio y que estén confirmadas
22     List<Cita> findByEspacioAndEstadoAndFechaCitaBetween(
23         Cita.Espacio espacio, Cita.EstadoCita estado, LocalDateTime start, LocalDateTime end
24     );
25
26     //Buscar todas las citas de un usuario. Ordenado por fecha, descendiente.
27     List<Cita> findByUsuarioDniOrderByFechaCitaAsc(String dni);
28     //eliminar citas de una mascota
29     @Modifying
30     @Query("DELETE FROM Cita c WHERE c.mascota.id = :mascotaId")
31     void deleteByMascotaId(@Param("mascotaId") Long mascotaId);
32     //buscar citas proximas confirmadas de un usuario
33     List<Cita> findByUsuarioDniAndEstadoAndFechaCitaAfterOrderByFechaCitaAsc(String dni, Cita.EstadoCita estado, LocalDateTime fechaCita);
34     //historial citas del usuario
35     @Query("SELECT c FROM Cita c WHERE c.usuario.dni = :dni AND ((c.estado = 'CANCELADA') OR (c.fechaCita < :ahora)) ORDER BY c.fechaCita DESC")
36     List<Cita> findHistorialByUsuario(@Param("dni") String dni, @Param("ahora") LocalDateTime ahora);
37 }
38
39
```

Figura 52: Cita Repository

Cerrar Sesión

Al seleccionar sesión se nos abrirá una ventana emergente que nos cuestionará sobre si estamos seguros de realizar esta opción, al responder afirmativo nos borrará los datos de nuestra sesión guardado en el almacenamiento de Shared Preferences(dni y rol) y nos devolverá al Login. Esto se consigue asignando en la lista de Items null como ruta para el item cerrar sesión. A continuación adjunto imagen del código en el que se logra esta acción:

```
@override
Widget build(BuildContext context) {
  return InkWell(
    borderRadius: BorderRadius.circular(18),
    onTap: () async {
      //si la ruta es null
      if (item.destino == null) {
        //muestra ventana emergente
        final confirmed = await showDialog<bool>(
          context: context,
          builder: (context) => AlertDialog(
            title: const Text('Cerrar sesión'),
            content: const Text('¿Seguro que quieres cerrar sesión?'),
            actions: [
              TextButton(onPressed: () => Navigator.pop(context, false), child: const Text('Cancelar')),
              TextButton(onPressed: () => Navigator.pop(context, true), child: const Text('Cerrar sesión')),
            ],
          ), // AlertDialog
        );
        //si responde Cerrar Sesión
        if (confirmed == true) {
          try {
            // Borrar datos de sesión (SharedPreferences)
            final prefs = await SharedPreferences.getInstance();
            await prefs.remove('dni_usuario');
            await prefs.remove('rol_usuario');
            // Navegar al login
            Navigator.pushAndRemoveUntil(
              context,
              MaterialPageRoute(builder: (context) => Login()),
              (route) => false,
            );
          } catch (e) {
            ScaffoldMessenger.of(context).showSnackBar(
              SnackBar(content: Text('Error al cerrar sesión: $e')),
            );
          }
        } else {
          Navigator.push(
```

Figura 53: Cerrar sesion

Resto de funcionalidades específicas

El resto de funcionalidades se mostrarán en la siguiente versión de la aplicación, antes del lanzamiento, en ellas por ahora se mostrará una pantalla que indica que estarán disponibles próximamente.

4.9. Iteración 8 - Fase VIII: Funcionalidades de los usuarios Administradores

4.9.1. Agenda Citas

El usuario con rol **admin** al **seleccionar citas en el menú de inicio** en vez de a la pantalla mis citas, accederá a la agenda de citas tanto de la consulta como de la peluquería.

Esta pantalla basada en la de pedir cita constará de un **calendario donde al seleccionar los distintos días podrás ver las citas que hay agendadas**, podrás navegar entre consulta y peluquería con un selector de espacios. Abajo saldrán listadas todas las citas confirmadas por ese día(las de todos los clientes). Adjunto imagen del controller y service del endpoint:

```
//obtener citas proximas confirmadas de un usuario
public List<GetCitaDto> getCitasProximasConfirmadasByUsuario(String dni) {
    List<Cita> citas = citaRepository.findByUsuarioDniAndEstadoAndFechaCitaAfterOrderByFechaCitaAsc(
        dni, Cita.EstadoCita.CONFIRMADA, LocalDateTime.now()
    );
    return citas.stream().map(citaMapper::toGetCitaDto).collect(Collectors.toList());
}
```

Figura 54: citas confirmadas

```
//endpoint para obtener las proximas citas del usuario
@GetMapping("/usuario/{dni}/proximas")
public List<GetCitaDto> getCitasProximasConfirmadasByUsuario(@PathVariable String dni) {
    return citaService.getCitasProximasConfirmadasByUsuario(dni);
}
```

Figura 55: citas proximas

Para gestionar la navegación por rol, se hará a través de un condicional, utilizando una variable cargada con el rol del usuario desde el almacenamiento de Shared Preferences.
Imagen código condicional:

```
// Citas con imagen
_positionedCircleCustom(
  angle: 320,
  distance: 90,
  centerOffset: const Offset(110, 110),
  child: _circleMenuButton(
    imagePath: "lib/assets/icon_citas.png", // Ruta a tu imagen
    label: "Citas",
    onTap: () async {
      //obtencion del rol desde sharedPreferences
      final prefs = await SharedPreferences.getInstance();
      final rol = prefs.getString('rol_usuario');
      //si rol es admin navegamos a la pantalla de admin, que muestra las citas que hay para cada dia
      if (rol == 'ADMIN') {
        Navigator.push(
          context,
          MaterialPageRoute(builder: (context) => AdminCitas()),
        );
      } else {
        Navigator.push(
          context,
          MaterialPageRoute(builder: (context) => CitasScreen()),
        );
      }
    },
  ),
),
```

Figura 56: navegacion rol

4.9.2. Gestión Productos Tienda.

Añadir Producto

En la pantalla de la Tienda a un usuario con el Rol 'ADMIN' le saldrá un botón flotante arriba del de la cesta . El formulario para crear el producto será muy similar al de crear una mascota aunque con su correspondientes campos. El proceso para subir imágenes lo manejaremos de la misma manera. Para utilizar el endpoint necesitaremos autorización. Lo hacemos de la siguiente manera. Adjunto imagen del código de la petición POST:

```
202
203 // Prepara el body para el POST
204 final url = Uri.parse('http://192.168.1.131:8080/productos');
205 final body = {
206   "nombre": nombreController.text.trim(),
207   "marca": marcaController.text.trim(),
208   "categoria": categoriaSeleccionada,
209   "especies": especiesSeleccionadas,
210   "descripcion": descripcionController.text.trim(),
211   "precio": double.tryParse(precioController.text.replaceAll(',', '.').trim()) ?? 0.0,
212   // "imagen": null, // El campo imagen se actualiza tras subir la imagen
213 };
214
215 final response = await http.post(
216   url,
217   headers: {"Content-Type": "application/json",
218 //autenticación para utilizar el endpoint con su usuario y contraseña
219   'Authorization': basicAuthHeader('root', '1234'),
220 },
221   body: jsonEncode(body),
222 );
223
224 if (response.statusCode == 200 || response.statusCode == 201) {
225   final producto = jsonDecode(response.body);
226   final productoId = producto['id'] as int;
227
228   // Si se seleccionó una imagen, súbela
229   if (_imagenProducto != null) {
230     await _subirImagenProducto(productoId, _imagenProducto!);
231   }
232 }
```

Figura 57: post productos

El metodo basicAuthHeader se encuentra en el paquete utils. Adjunto imagen codigo:

```
import 'dart:convert';

String basicAuthHeader(String username, String password) {
  String credentials = '$username:$password';
  Codec<String, String> stringToBase64 = utf8.fuse(base64);
  return 'Basic ${stringToBase64.encode(credentials)}';
}
```

Editar y eliminar producto

Para editar y eliminar un producto se habilita una nueva pantalla de detalle producto a la que podrá navegar un usuario admin. No lo haga sobre la misma que la de los usuarios con condicionales para evitar posibles filtraciones de funciones tan importantes como editar la descripción y precio de un producto o eliminarlo.

Los parámetros a editar para un producto serán el precio y la descripción. Esto será a través de una ventana emergente, se cargarán los datos actuales del producto, podrás modificar estos y al pulsar el botón guardar se ejecutará el endpoint PATCH.

El botón de eliminar es muy simple, al pulsarlo te pedirá una confirmación y ejecutará el DELETE por id.

4.10. Iteración 9 - Fase IX: Últimas Pruebas, Despliegue y Mantenimiento

4.10.1. Definición Regex (expresiones regulares)

Para incorporar una expresión regular deberemos **definir la expresión y un metodo para validarla**, envolver el código del body en un **widget Form** creando una key, **llamar al método de validación** en el textField y por último **al ejecutar** el registro y **validar que todos los campos tienen un formato válido**. Esto será añadido para los campos dni, email, fechas y teléfono de toda la aplicación. Como ejemplo de la implementación adjunto código del SignUp.

```
25 //Definición expresiones regulares (Regex)
26 // Expresiones regulares
27 final RegExp dniRegex = RegExp(r'^[0-9]{8}[A-Za-z]$');
28 final RegExp emailRegex = RegExp(r'^[\w-\.\.]+\@([\w-]+\.)+[\w-]{2,4}$');
29 final RegExp telefonoRegex = RegExp(r'^(\+|00)[0-9]{1,5}(?:[ -]?[0-9]{3,}){2,}$');
30
31 String? validarDNI(String? value) {
32   if (value == null || value.isEmpty) return 'DNI obligatorio';
33   if (!dniRegex.hasMatch(value)) return 'Formato inválido (Ej: 12345678A)';
34   return null;
35 }
36
37 String? validarEmail(String? value) {
38   if (value == null || value.isEmpty) return 'Email obligatorio';
39   if (!emailRegex.hasMatch(value)) return 'Formato inválido (Ej: carmen@gmail.com)';
40   return null;
41 }
42
43 String? validarTelefono(String? value) {
44   if (value == null || value.isEmpty) return 'Teléfono obligatorio';
45   if (!telefonoRegex.hasMatch(value)) return 'Formato inválido (Ej: +34 666303491)';
46   return null;
47 }
```

Figura 59: Regex

```
// Widget TextFormField para el campo Email.
TextFormField(
  controller: emailController, // controlador API
  // validar formato
  validator: validarEmail,
  // Define la decoración del campo, incluyendo la etiqueta, el estilo de la etiqueta, el borde y el icono.
  decoration: const InputDecoration(
    labelText: "Email",
    labelStyle: TextStyle(color: Colors.white),
    floatingLabelStyle: TextStyle(color: Colors.blue), // cambiar el color del labelText, cuando esta flotando
    enabledBorder: UnderlineInputBorder(
      borderSide: BorderSide(color: Colors.lightBlueAccent),
    ), // UnderlineInputBorder
    focusedBorder: UnderlineInputBorder(
      borderSide: BorderSide(color: Colors.blue),
    ), // UnderlineInputBorder
    errorBorder: UnderlineInputBorder( // Borde rojo cuando hay error
      borderSide: BorderSide(color: Colors.red),
    ), // UnderlineInputBorder
    focusedErrorBorder: UnderlineInputBorder( // Borde rojo cuando hay error y está enfocado
      borderSide: BorderSide(color: Colors.red),
    ), // UnderlineInputBorder
    prefixIcon: Icon(Icons.email, color: Colors.white),
  ), // InputDecoration
  // Define el estilo del texto como blanco.
  style: const TextStyle(color: Colors.white),
), // TextFormField
// Widget SizedBox para añadir un espacio vertical de 15 píxeles.
const SizedBox(height: 15),
```

Figura 60: TextFormField

```
// Botón Registrarse
ElevatedButton(
  onPressed: () {
    // Acción al presionar "Registrarme"
    // valida que todos los campos tienen un formato válido
    if (_formKey.currentState!.validate()) {
      registrarUsuario();
    }
  },
  style: ElevatedButton.styleFrom(
    backgroundColor: Colors.lightBlueAccent,
    minimumSize: const Size(double.infinity, 50),
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(10),
    ), // RoundedRectangleBorder
  ),
  child: const Text(
    "REGISTRARME",
    style: TextStyle(fontSize: 18, color: Colors.white),
  ), // Text
), // ElevatedButton
```

Figura 61: validacion regex

4.10.2 Encriptado de contraseñas

Esta funcionalidad podemos implementarla utilizando la librería Spring Security, así antes de enviar alguna contraseña o datos sensibles a la BBDD se encriptará y se almacenará el Hash. Bien, para ello en nuestro archivo de configuración de seguridad añadimos el siguiente metodo:

```
//Configuracion para encriptar contraseñas
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Figura 62: validacion encriptado

Deberemos implementarlo en el service de usuario en los metodos utilizados para crear y para actualizar un usuario:

```
// Crear un nuevo usuario
public Usuario createUsuario(CreateUsuarioDto dto) {
    Usuario nuevoUsuario = usuarioMapper.toUsuario(dto);
    nuevoUsuario.setContrasena(passwordEncoder.encode(dto.getContrasena()));
    return usuarioRepository.save(nuevoUsuario);
}

// Actualizar un usuario
public Optional<Usuario> updateUsuario(String dni, UpdateUsuarioDto dto) {
    return usuarioRepository.findByDni(dni).map(usuario -> {
        usuario.setNombre(dto.getNombre());
        usuario.setApellidos(dto.getApellidos());
        usuario.setEmail(dto.getEmail());
        usuario.setTelefono(dto.getTelefono());
        if (dto.getContrasena() != null && !dto.getContrasena().isEmpty()) {
            usuario.setContrasena(passwordEncoder.encode(dto.getContrasena()));
        }
        return usuarioRepository.save(usuario);
    });
}
```

Figura 63: service usuario

4.10.3 Despliegue

Para finalizar el despliegue de esta primera versión de la aplicación ejecutaremos el siguiente comando a través de **terminal**:

flutter build apk --release

Esto compilará el proyecto y generará el apk lista para instalar y usar

Capítulo 5 Conclusiones

5.1. Revisión de los objetivos

5.1.1. Grado de cumplimiento de objetivos

El **objetivo principal** del proyecto era **desarrollar una aplicación móvil** multiplataforma **para optimizar** la gestión integral de **una pequeña clínica veterinaria**, facilitando tanto la administración interna como la experiencia del cliente. **Tras el desarrollo, se puede afirmar que este objetivo se ha cumplido en gran medida.** La aplicación **permite** a los clientes **gestionar sus mascotas, reservar y cancelar citas, y acceder a una tienda online de productos especializados.** Además, la **arquitectura cliente-servidor** (Flutter + Spring Boot + MySQL) **garantiza la separación de responsabilidades, seguridad y escalabilidad** del sistema.

Respecto a los objetivos específicos

Gestión de Mascotas y Citas

Se ha implementado un sistema robusto para el **alta, edición y eliminación de mascotas**, así como la **solicitud, consulta y cancelación de citas para diferentes servicios y espacio**(consulta y peluquería). El sistema de reservas muestra la disponibilidad de días y horas, evitando solapamientos y facilitando la organización interna.

Tienda online y carrito

La **tienda online permite visualizar, filtrar y seleccionar productos**, así como **gestionar el carrito de compras** de forma intuitiva. La funcionalidad de pago online, aunque prevista, ha quedado pendiente para una futura actualización.

Comunicación y visibilidad

El contacto directo con la clínica (llamada y WhatsApp) y la opción de compartir la aplicación se han diseñado, pero su implementación efectiva **se ha pospuesto hasta la publicación oficial de la app en las tiendas.**

Gestión de productos y agenda para administradores

El administrador puede **gestionar productos (alta, edición y eliminación)** y **visualizar la agenda de citas** en formato calendario, permitiendo una organización eficiente de los recursos.

Seguridad y privacidad

Se ha garantizado la seguridad y privacidad de los datos como contraseñas encriptándolas con la librería **Spring Security**, también la protección de endpoints, mediante la autenticación. En

definitiva, **las contraseñas se almacenan de forma segura y el acceso a información sensible está restringido.**

Escalabilidad y adaptabilidad

La **aplicación** ha sido **diseñada de forma modular, permitiendo su adaptación a otras pequeñas empresas o comercios** con necesidades **similares**. La arquitectura facilita futuras ampliaciones y personalizaciones.

5.1.2. Desviaciones y justificaciones

Funcionalidades pendientes

Algunas funcionalidades, como la pasarela de pagos, el contacto directo por WhatsApp y llamada, la opción de compartir la app, ciertas opciones del perfil de usuario y las opciones de recuérdame y recuperar contraseña del login (historial de pagos, favoritos, notificaciones, ajustes y ayuda), han quedado pendientes para futuras actualizaciones. Estas desviaciones se deben principalmente a limitaciones de tiempo y recursos, así como a la necesidad de priorizar las funcionalidades esenciales para la presentación de una primera versión al cliente.

Planificación

Aunque la planificación inicial fue realista, la implementación de **algunas funcionalidades requirió más tiempo del previsto**, especialmente en la integración entre frontend y backend. Sin embargo, la metodología en cascada permitió avanzar de forma ordenada y controlada, minimizando los riesgos asociados a cambios de requisitos.

5.1.3. Competencias adquiridas

Durante el desarrollo del TFG se han puesto en práctica y consolidado las competencias específicas de la tecnología cursada, entre ellas:

Desarrollo Multiplataforma con Flutter

Se ha adquirido **experiencia en el desarrollo de aplicaciones móviles para Android e iOS**, utilizando widgets, gestión de estados y navegación entre pantallas **en un nuevo framework y lenguaje distintos a los estudiados durante el curso.**

Desarrollo de APIs REST con SpringBoot

Se han **implementado endpoints** para la gestión de usuarios, mascotas, citas, productos y carrito, **aplicando buenas prácticas de diseño y seguridad. Ampliando** así los **conocimientos** adquiridos sobre SpringBoot en **PSP** durante el curso.

Gestión de BBDD con MySQL

Se ha **diseñado y optimizado el modelo de datos**, garantizando la integridad y consistencia de la información.

Control de versiones con GitHub

Se ha utilizado GitHub para el **control de versiones**, facilitando el mantenimiento y la

evolución del código.

Pruebas y depuración

Se han realizado **pruebas manuales y validación de endpoints con Postman**, consolidando competencias en detección y corrección de errores. Ampliando los conocimientos que ya teníamos.

Gestión de proyectos

Se ha **planificado y organizado el trabajo** en fases, priorizando tareas y adaptando la planificación **según las necesidades detectadas**.

5.2. Trabajos Futuros

El **desarrollo** de la aplicación móvil para la gestión de la clínica veterinaria **ha cubierto los requisitos esenciales** tanto para los usuarios como para el personal administrativo, **pero** se han **identificado varias áreas de mejora y funcionalidades pendientes** que enriquecerán el sistema y permitirán adaptarlo a nuevas necesidades del sector.

Entre los **principales trabajos futuros** se encuentran:

Integración pasarela de pago

Se prevé **implementar** una solución segura para la **gestión de pagos online**, permitiendo a los usuarios completar el proceso de compra de productos desde la aplicación. Antes de realizar este apartado **se deberá cuestionar al cliente sobre los métodos de pago que desea establecer**.

Contacto directo por Whatsapp y llamada telefónica

Aunque ya **se ha diseñado la interfaz**, la integración efectiva de los botones de contacto directo (WhatsApp y llamada) **quedará habilitada antes del lanzamiento en tiendas oficiales**.

Opción de recordar usuario y cambiar contraseña desde el correo.

Antes del lanzamiento será importante implementar la función para poder cambiar la contraseña enviando un correo de recuperación por si el usuario la ha olvidado.

Resto de funcionalidades de perfil de usuario

El historial de pagos, la gestión de productos favoritos, las notificaciones personalizadas y los ajustes avanzados del perfil serán incorporados en **futuras versiones** antes del lanzamiento, ofreciendo una experiencia más completa y personalizada.

Sección ayuda y documentación legal

Se desarrollará una **sección específica para mostrar la documentación legal de la clínica**, información sobre el tratamiento de datos y recomendaciones para el usuario, **asegurando la transparencia y el cumplimiento normativo**.

Publicación en tiendas oficiales y opción compartir

Una vez **finalizadas las pruebas y validaciones**, **la aplicación será publicada** en las tiendas de aplicaciones móviles (Google Play y App Store), permitiendo su descarga y uso por parte de cualquier cliente. Además, **se habilitará la función de compartir la app para aumentar la visibilidad del negocio**.

Escalabilidad a otras PYMES

Se continuará trabajando en la modularidad y flexibilidad del sistema, facilitando su adaptación a otras pequeñas empresas del sector servicios con necesidades similares.

Mejora de la experiencia de usuario y accesibilidad

Se incorporarán nuevas funcionalidades y mejoras en la interfaz, siguiendo las tendencias actuales de diseño centrado en el usuario y accesibilidad universal. También se deberá de realizar un **mejor control de excepciones**.

Estos trabajos futuros permitirán que la aplicación evolucione y se consolide como una herramienta integral para la digitalización y mejora de la gestión en pequeñas y medianas empresas del sector veterinario, respondiendo a las necesidades detectadas tanto por los usuarios como por los profesionales del sector

Capítulo 6: Bibliografía

(Estos ejemplos siguen el formato American Psychological Association (APA) 6th Edition:

[Estilo APA 7ª ed. - Citas y elaboración de bibliografía: el plagio y el uso ético de la información - Biblioguías at Universidad Autónoma de Madrid \(uam.es\)](#))

Flutter dev

Flutter. Flutter: Beautiful native apps in record time. <https://flutter.dev/>

Wikipedia

Wikipedia. Flutter (software). <https://en.wikipedia.org/wiki/>

Learn Microsoft

Microsoft.Learn Microsoft. <https://learn.microsoft.com/>

Baeldung.

Baeldung: Learn and grow as a developer. <https://www.baeldung.com/>

Canva.

Canva: Diseño gráfico online. <https://www.canva.com/>

CapitanApps.

CapitanApps: Desarrollo y diseño de aplicaciones. <https://capitanapps.com/>

Codea.

Codea: Desarrollo y programación. <https://codea.app/>

Dev Community.

DEV Community. <https://dev.to/>

Draw.io.

Draw.io: Diagramas en línea. <https://draw.io/>

DZone.

DZone: The world's largest community of software developers. <https://dzone.com/>

Flaticon.

Flaticon: Iconos y gráficos gratis. <https://www.flaticon.com/>

Freepik.

Freepik: Vectores, fotos y PSD gratis. <https://www.freepik.com/>

GitHub.

GitHub. <https://github.com/>

Google Codelabs.

Google Developers Codelabs. <https://codelabs.developers.google.com/>

Iconos8.

Icons8: Free icons, photos, and illustrations. <https://icons8.com/>

IONOS.

IONOS: Hosting, dominios y servidores. <https://www.ionos.com/>

MySQL.

MySQL. <https://www.mysql.com/>

PNGEgg.

PNGEgg: PNG images and cliparts for free download. <https://www.pngegg.com/>

Pngtree.

Pngtree: PNG images, backgrounds and templates for free download.

<https://pngtree.com/>

Pub.dev.

Pub.dev: Dart packages. <https://pub.dev/>

Reddit.

Reddit: The front page of the internet. <https://www.reddit.com/>

Spring.io.

Spring.io: Spring framework. <https://spring.io/>

Stack Overflow.

Stack Overflow. <https://stackoverflow.com/>

Visual Studio.

Visual Studio. <https://visualstudio.microsoft.com/>

YouTube.

YouTube. <https://www.youtube.com/>

Anexo I.

I.1.1. Sección 1 del Anexo I

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.