ILERNA

CFGS DESARROLLO DE APLICACIONES MULTIPLATAFORMA
Proyecto Final de Ciclo

VEHICOOL

Autor: Carla Domínguez Espinosa , Rodrigo Moreno Aguilar

Tutor: Javier Martín Rivero

Fecha de entrega: 21 de Mayo de 2025

Convocatoria: 2024 2025

ÍNDICE

Introducción

Justificación del tema

Hoy en día, los talleres mecánicos necesitan herramientas que les ayuden a organizar su trabajo y a mejorar la comunicación con sus clientes. Esta aplicación surge para cubrir esas necesidades, facilitando tanto a los usuarios como al taller el manejo de la información y los servicios.

La idea es que los clientes puedan crear su perfil, registrar sus vehículos y consultar el historial de reparaciones y mantenimientos que se han hecho. Así, tienen todo más claro y pueden controlar mejor sus coches.

Además, la app incluye un calendario para que los clientes puedan agendar citas de forma fácil, eligiendo el tipo de servicio y recibiendo avisos para no olvidarse. Esto ayuda al taller a planificar mejor su tiempo y evita confusiones con las citas.

También, los usuarios podrán valorar y comentar los servicios recibidos, lo que permite al taller saber qué está funcionando bien y qué puede mejorar. Y para facilitar aún más las cosas, se gestiona la facturación digital, con acceso rápido a las facturas y la posibilidad de integrar pagos online en el futuro.

En resumen, esta aplicación busca hacer la experiencia del cliente más cómoda y transparente, al mismo tiempo que ayuda al taller a organizarse y ofrecer un mejor servicio.

Motivación

La motivación principal que nos impulsó a desarrollar esta aplicación surge de observar de cerca cómo funcionan los talleres mecánicos hoy en día y las dificultades que enfrentan tanto los empleados como los clientes. Aunque el sector de la automoción ha avanzado mucho, la gestión diaria en muchos talleres sigue siendo bastante tradicional y poco digitalizada. Esto genera ineficiencias que se traducen en pérdida de tiempo, errores y frustración para todos.

En muchos talleres, la gestión de citas, vehículos, reparaciones y facturación se hace todavía a mano o con sistemas muy básicos que no se comunican entre sí. Esto no solo sobrecarga al personal con tareas administrativas repetitivas, sino que también dificulta ofrecer un servicio rápido y transparente. Además, los clientes suelen depender de llamadas o desplazamientos físicos para reservar una cita, preguntar por el estado de su vehículo o recibir facturas, lo que puede ser poco cómodo y generar incertidumbre.

Queremos cambiar esa realidad creando una herramienta digital que facilite y agilice todas estas gestiones. La idea es que el cliente pueda manejar todo desde su móvil o computadora, sin perder tiempo ni esfuerzo, y que el taller pueda organizarse mejor, evitando errores y mejorando la comunicación. Al digitalizar y automatizar estos procesos, se gana en eficiencia, en calidad del servicio y en satisfacción para ambas partes.

Además, creemos que modernizar el sector no solo es importante para facilitar el día a día, sino también para adaptarnos a las nuevas expectativas de los usuarios, que cada vez buscan más rapidez, comodidad y transparencia.

Abstract

The main motivation that drove us to develop this application comes from closely observing how auto repair shops operate today and the challenges faced by both employees and customers. Although the automotive sector has advanced significantly, daily management in many workshops remains quite traditional and poorly digitized. This creates inefficiencies that lead to wasted time, errors, and frustration for everyone.

In many workshops, the management of appointments, vehicles, repairs, and billing is still done manually or with very basic systems that do not communicate with each other. This not only overloads the staff with repetitive administrative tasks but also makes it difficult to provide fast and transparent service. Additionally, customers often have to rely on phone calls or physical visits to book appointments, inquire about the status of their vehicles, or receive invoices, which can be inconvenient and cause uncertainty.

We want to change this reality by creating a digital tool that facilitates and speeds up all these processes. The idea is for customers to manage everything from their mobile phones or computers without wasting time or effort, and for the workshop to organize itself better, avoiding errors and improving communication. By digitizing and automating these processes, efficiency, service quality, and satisfaction for both parties are improved.

Furthermore, we believe that modernizing the sector is not only important to make daily operations easier but also to adapt to the new expectations of users, who increasingly seek speed, convenience, and transparency.

Objetivos propuestos (generales, específicos):

Objetivo General:

Desarrollar una aplicación móvil para la gestión de citas y reparaciones en un taller mecánico, permitiendo a los clientes programar servicios y comunicarse con los mecánicos de manera eficiente.

Objetivos Específicos:

1. **Implementar un sistema de gestión de citas** que permita a los clientes reservar horarios disponibles en un calendario, evitando conflictos.

2. **Automatizar la creación de reparaciones** para los mecánicos.

3. **Incorporar un sistema de chat** entre cliente y mecánico, disponible solo durante la reparación, para facilitar la comunicación.

4. **Proporcionar un historial de reparaciones y facturación** accesible para los clientes, con detalles sobre trabajos realizados y costos.

5. **Desarrollar una interfaz intuitiva y accesible**, asegurando una experiencia de usuario clara tanto para clientes como para mecánicos.

6. **Permitir la visualización de facturas y la simulación de pagos**, (sin integrar pasarelas de pago reales)

7. **Ofrecer una gestión de vehículos para los clientes**, permitiendo registrar y visualizar los automóviles que llevarán al taller.

Metodología utilizada

Para el desarrollo de esta aplicación hemos decidido utilizar la metodología Scrum, que es un marco ágil muy efectivo para gestionar proyectos de software. Scrum nos permite trabajar de manera organizada y flexible, dividiendo el proyecto en pequeñas etapas llamadas sprints.

Planificación y Diseño:

Se crearan prototipos visuales (MockUps) usando Figma, una herramienta que nos permite diseñar la interfaz de usuario de forma rápida y clara. Estos prototipos ayudarán a validar ideas y a asegurar una experiencia intuitiva antes de comenzar la implementación.

Tecnologías y herramientas utilizadas en el proyecto :

Tecnologías:

Para desarrollar la aplicación del taller mecánico, hemos seleccionado una serie de tecnologías y herramientas que se ajustan a las necesidades del proyecto y facilitan tanto el desarrollo como el mantenimiento.

Figma: Se utiliza para diseñar los MockUps y definir la interfaz de usuario. Nos permite visualizar y ajustar el diseño antes de empezar a programar, asegurando una experiencia clara y coherente para los usuarios.

Android Studio: Es el entorno de desarrollo donde creamos la aplicación móvil. Nos ofrece todas las herramientas necesarias para escribir, probar y depurar la app en Android.

Draw.io: Herramienta sencilla que utilizamos para elaborar diagramas de casos de uso y otros diagramas que ayudan a entender el flujo y la interacción del sistema.

Kotlin: Lenguaje principal para la programación tanto de la lógica en la app móvil como del backend. Kotlin nos ofrece un código moderno, conciso y seguro, facilitando la integración entre ambas partes.

XML: Empleado para definir la estructura y diseño visual de las pantallas en la aplicación móvil, separando la interfaz gráfica de la lógica de negocio.

Spring Boot: Framework utilizado para desarrollar el backend, también programado en Kotlin. Permite crear servicios REST eficientes y gestionar WebSockets para funcionalidades en tiempo real, como el chat.

Hibernate: Biblioteca que facilita la gestión de la base de datos, haciendo el mapeo entre objetos Kotlin y tablas de la base de datos sin tener que escribir consultas SQL manualmente.

MySQL: Sistema de gestión de bases de datos relacional donde se almacena toda la información importante: usuarios, vehículos, reparaciones, mensajes, facturas, etc.

Retrofit: Biblioteca que usamos en la aplicación móvil para comunicarse con el backend vía APIs REST, manejando las peticiones y respuestas de forma sencilla y eficiente.

WebSockets: Para la implementación del chat en tiempo real entre clientes y mecánicos.

Desarrollo:

Desarrollo del backend

Para asegurar una comunicación fluida entre la aplicación móvil y el servidor backend, se diseñó e implementó una API REST desarrollada en **Spring Boot con Kotlin**, que actúa como intermediaria para gestionar las operaciones sobre los datos. Esta API permite realizar las operaciones básicas de creación, consulta, actualización y eliminación (CRUD) sobre los recursos principales como usuarios, vehículos, reparaciones y citas.

La API está construida bajo principios de arquitectura **cliente-servidor** y es **stateless**, lo que significa que cada petición que realiza la app contiene toda la información necesaria para ser procesada de forma independiente. Esto contribuye a la escalabilidad del sistema y simplifica el mantenimiento.

Los datos que viajan entre la app y el backend se intercambian en formato **JSON**, elegido por su ligereza y compatibilidad con las plataformas móviles.

Controladores REST

La imagen que se muestra es un ejemplo real de uno de nuestros controladores, concretamente el que gestiona los mensajes del chat entre cliente y mecánico. En este caso, se responde a una solicitud GET que pide el historial de mensajes según el ID de una reparación concreta.

```kotlin
@RestController  RodrigoMoreno2001
@RequestMapping("/api/chat")
class ControladorRestChat(
    private val chatService: ChatServiceImpl
) {

    @GetMapping("/{reparacionId}")  RodrigoMoreno2001
    fun obtenerHistorial(@PathVariable reparacionId: Long): ResponseEntity<List<ChatDto>> {
        val historial = chatService.obtenerPorIdReparacion(reparacionId)
        return ResponseEntity.ok(historial)
    }
}
```

En el backend de nuestra aplicación, desarrollado con **Kotlin y Spring Boot**, una de las piezas clave son los **controladores REST**. Estas clases son las encargadas de

gestionar las peticiones que llegan desde la app móvil y darles una respuesta adecuada. Se puede decir que actúan como el “puente” entre la interfaz de usuario y la lógica interna del sistema.

Cada controlador se asocia a una parte específica de la aplicación: por ejemplo, hay uno para el chat, otro para las reparaciones, otro para las citas, etc. Dentro de cada uno, se definen los distintos endpoints, es decir, las rutas a las que puede llamar la app (por ejemplo: /api/chat, /api/citas, etc.). Estos endpoints suelen estar acompañados de anotaciones como @GetMapping, @PostMapping, @PutMapping o @DeleteMapping, que indican si la app quiere consultar, crear, modificar o eliminar algo, respectivamente.

Cuando el usuario realiza una acción en la app (por ejemplo, enviar un mensaje, ver sus reparaciones o pedir una cita), la aplicación manda una solicitud HTTP a uno de estos endpoints. El controlador la recibe, pasa los datos al servicio correspondiente, y este se encarga de hacer la lógica de negocio (consultar o guardar datos, aplicar reglas, etc.). Finalmente, el controlador devuelve una respuesta en formato JSON para que la app pueda mostrarla.

WebSockets

Una parte importante del proyecto ha sido implementar un sistema de chat en tiempo real entre cliente y mecánico. Para conseguir esto, hemos utilizado WebSockets, una tecnología que permite mantener una conexión abierta y bidireccional entre el servidor y el cliente, sin necesidad de estar enviando peticiones constantemente como en las API REST.

```
@Configuration  RodrigoMoreno2001
@EnableWebSocket
class WebSocketConfig(
    private val controladoraChat: ControladorChat
) : WebSocketConfigurer {

    // se registra el handler en la ruta "/ws/chat/{roomId}", roomId es el identificador del chat

    override fun registerWebSocketHandlers(registry: WebSocketHandlerRegistry) {  RodrigoMoreno2001
        registry.addHandler(controladoraChat,  ...paths: "/ws/chat/{roomId}")
            .setAllowedOrigins("*") // esto permite conexiones de cualquier origen (es peligroso)
    }
}
```

La clase que vemos en la imagen es una configuración específica de Spring Boot para habilitar y registrar el WebSocket. Aquí se define en qué ruta estará disponible el canal de comunicación en tiempo real (en este caso, /ws/chat/{roomId}) y qué clase se encargará de gestionar esa comunicación (el controlador del chat).

Gracias a esta configuración, cuando un cliente o mecánico entra en el chat de una reparación, ambos dispositivos se conectan a la misma “sala” usando el roomId, que es el identificador de esa reparación. Desde ese momento, pueden intercambiar mensajes instantáneamente, y estos se procesan en tiempo real.

También hay que destacar que en este ejemplo se permite el acceso desde cualquier origen (setAllowedOrigins("*")). Esto está bien durante el desarrollo, ya que facilita las pruebas desde distintos dispositivos, pero en un entorno de producción debería limitarse por motivos de seguridad.

Desarrollo de la aplicación móvil

La aplicación Android se desarrolló utilizando **Kotlin**, junto con **XML** para definir la interfaz gráfica. La elección de Kotlin responde a su integración nativa con Android Studio y sus facilidades para escribir un código limpio y seguro.

Para la comunicación con el backend, se utilizó la librería **Retrofit**, que facilita la implementación de las llamadas a la API REST de manera asíncrona y eficiente, asegurando una buena experiencia de usuario sin bloqueos en la interfaz.

Implementación del chat en tiempo real

Uno de los componentes clave de la aplicación es el sistema de chat entre clientes y mecánicos, que se implementó usando **WebSockets** para permitir una comunicación bidireccional y en tiempo real. Esta tecnología evita la necesidad de realizar consultas constantes al servidor y permite que los mensajes se envíen y reciban de manera instantánea.

Gestión de datos y persistencia

En el backend, se empleó **Hibernate** para la gestión de la persistencia de datos con una base de datos **MySQL**, donde se almacenan los usuarios, vehículos, reparaciones, citas y mensajes del chat.

En la aplicación móvil, se optó por almacenar localmente ciertos datos temporales o de sesión para mejorar la rapidez de acceso y permitir cierto grado de uso offline en áreas limitadas.

Desarrollo en Android:

```
/**
 * Objeto singleton que configura y proporciona acceso a Retrofit.
 */
object RetrofitClient {

    // 10.0.2.2 apunta al localhost de la máquina host.
    private const val BASE_URL = "http://10.0.2.2:8080/"

    /**
     * Configuración personalizada de Gson para adaptar tipos de fecha y hora.
     */
    val gson = GsonBuilder()
        .registerTypeAdapter(LocalDateTime::class.java, LocalDateTimeAdapter())
        .registerTypeAdapter(LocalDate::class.java, LocalDateAdapter())
        .create()

    private val retrofit: Retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create(gson))
        .build()

    val usuarioService: UsuarioService by lazy { retrofit.create(UsuarioService::class.java) }

    val facturaService: FacturaService by lazy { retrofit.create(FacturaService::class.java) }

    val reparacionService: ReparacionService by lazy { retrofit.create(ReparacionService::class.java) }

    val vehiculoService: VehiculoService by lazy { retrofit.create(VehiculoService::class.java) }

    val chatService: ChatService by lazy { retrofit.create(ChatService::class.java) }
}
```

El objeto RetrofitClient es el encargado de gestionar la comunicación entre la app Android y el backend. A través de Retrofit, se configuran las conexiones HTTP que permiten a la aplicación enviar peticiones y recibir respuestas del servidor. Se define una URL base que apunta al servidor local (10.0.2.2:8080), una dirección especial que

permite a un emulador de Android conectarse con el localhost de la máquina donde se ejecuta el backend.Este objeto actúa como un punto centralizado desde el que se accede a los distintos servicios que ofrece el backend. Por ejemplo, si la app necesita consultar los vehículos, enviar un mensaje por el chat o crear una reparación, usará alguna de las interfaces (UsuarioService, VehiculoService, ChatService, etc.) que este cliente expone. Estas interfaces están definidas previamente en la app y cada una describe los endpoints disponibles con sus métodos HTTP correspondientes. Gracias a esto, la app puede trabajar con datos complejos del backend de forma sencilla, sin necesidad de escribir el código de bajo nivel que gestiona manualmente las solicitudes.

```
fun inicioSesion(dto: AutenticarDTO) {
    RetrofitClient.usuarioService.autenticar(dto).enqueue(object : Callback<UsuarioDTO> {
        override fun onResponse(call: Call<UsuarioDTO>, response: Response<UsuarioDTO>) {
            if (response.isSuccessful && response.body() != null) {

                val usuario = response.body()
                val session = SessionManager(this@Login)

                session.persistirUsuario(usuario?.id ?: -1, usuario?.nombre ?: "", usuario?.privilegios ?: ""
                Toast.makeText(this@Login, "Bienvenido ${usuario?.nombre}", Toast.LENGTH_SHORT).show()

                val intent = Intent(this@Login, AppActivity::class.java)
                startActivity(intent)
            } else {
                Toast.makeText(this@Login, "Credenciales incorrectas", Toast.LENGTH_SHORT).show()
            }
        }

        override fun onFailure(call: Call<UsuarioDTO>, t: Throwable) {
            Log.e("LOGIN", "Error de conexión", t)
            Toast.makeText(this@Login, "Error al conectar a la API", Toast.LENGTH_SHORT).show()
        }
    })
}
```

Este método, llamado inicioSesion, se encarga de gestionar el proceso de autenticación del usuario desde la app Android. Recibe como parámetro un objeto AutenticarDTO, que contiene los datos introducidos por el usuario en el formulario de inicio de sesión (normalmente el correo y la contraseña).

Utilizando Retrofit, se realiza una llamada a la API del backend mediante el servicio usuarioService. Esta llamada es asíncrona, lo que significa que no bloquea la interfaz mientras espera la respuesta del servidor. Cuando el servidor responde, se ejecuta el bloque onResponse.

Si la respuesta es satisfactoria (isSuccessful) y el cuerpo de la respuesta no es nulo, se interpreta que las credenciales son correctas. Se recupera la información del usuario

(UsuarioDTO) y se guarda en una sesión local mediante SessionManager. Esta clase se encarga de persistir datos importantes como el ID, el nombre, los privilegios y el correo del usuario, permitiendo que se mantenga su sesión activa mientras navega por la app. A continuación, se muestra un mensaje de bienvenida y se redirige al usuario hacia la pantalla principal de la aplicación (AppActivity).

En caso de que las credenciales sean incorrectas o haya un fallo en la respuesta, se muestra un mensaje de error. Y si ocurre un problema de conexión (por ejemplo, si el servidor no responde), se entra en el bloque onFailure, donde se informa al usuario mediante un mensaje y se registra el error en el log para facilitar la depuración.

```
// Obtiene el historial de mensajes desde el servidor usando Retrofit
fun obtenerMensajes(id: Long){
    RetrofitClient.chatService.getMensajes(id).enqueue(object : Callback<List<ChatDto>> {
        override fun onResponse(call: Call<List<ChatDto>>, response: Response<List<ChatDto>>) {...}

        override fun onFailure(call: Call<List<ChatDto>>, t: Throwable) {
            Log.e("LOGIN", "Error de conexión", t)
        }
    })
}
// Establece una conexión WebSocket con el servidor para recibir mensajes en tiempo real
private fun conectarWebSocket(id: Long) {
    val request = Request.Builder()
        .url("ws://10.0.2.2:8080/ws/chat/$id")
        .build()

    val client = OkHttpClient()
    webSocket = client.newWebSocket(request, object : WebSocketListener() {
        // esta parte se ejecuta cuando se recibe un mensaje de un cliente externo
        override fun onMessage(webSocket: WebSocket, text: String) {
            val mensaje = Gson().fromJson(text, ChatDto::class.java)
            requireActivity().runOnUiThread {
                adapter.agregarMensaje(mensaje)
                recyclerView.scrollToPosition(adapter.itemCount - 1)
            }
        }
    })
}

private fun enviarMensaje(mensaje: ChatDto) {
    val jsonMensaje = Gson().toJson(mensaje)
    webSocket.send(jsonMensaje)
    editTextMensaje.text.clear()
}
```

El sistema de mensajería dentro de la aplicación combina dos tecnologías fundamentales: Retrofit, para recuperar el historial de mensajes ya guardados en el servidor, y WebSocket, para la comunicación en tiempo real entre cliente y mecánico. Cuando el usuario accede al chat asociado a una reparación, lo primero que ocurre es

una llamada HTTP mediante Retrofit a la API REST del servidor. Esta petición consulta todos los mensajes anteriores relacionados con la reparación concreta y, si la respuesta es exitosa, estos mensajes se cargan y se muestran en pantalla, desplazando automáticamente la vista al más reciente. Esta parte permite mantener una continuidad en la conversación, incluso si la app se ha cerrado anteriormente.

Una vez cargado el historial, se establece una conexión WebSocket con el servidor. Para ello, se genera una URL dinámica que contiene el identificador de la reparación y se crea un cliente con OkHttpClient. Gracias a esta conexión persistente, cada vez que uno de los participantes envía un nuevo mensaje, el servidor lo transmite de inmediato al otro extremo sin necesidad de refrescar o lanzar nuevas peticiones. La recepción de estos mensajes es gestionada mediante un WebSocketListener que, al detectar un nuevo mensaje entrante, lo convierte desde JSON a un objeto nativo y lo añade automáticamente a la vista del chat. Esto permite una interacción mucho más fluida y natural, imitando el comportamiento típico de aplicaciones de mensajería modernas.

El envío de mensajes, por su parte, también se realiza a través del WebSocket. Una vez que el usuario escribe su mensaje, este se transforma a formato JSON utilizando la librería Gson y se envía directamente al servidor. El mensaje es entonces retransmitido al otro participante, quien lo recibe al instante gracias a la conexión persistente. Además, tras cada envío, el campo de texto se limpia automáticamente para facilitar la escritura del siguiente mensaje.

```
pagarBtn.setOnClickListener {

    val reparacion = requireNotNull(reparacion)

    val reparacionCompletada = ReparacionOutputDTO(
        id = reparacion.id,
        fechaEntrada = reparacion.fechaEntrada,
        estado = "Completado",
        servicios = reparacion.servicios,
        motivos = reparacion.motivos,
        vehiculoId = reparacion.vehiculo.id!!,
        facturaId = reparacion.factura
    )
    RetrofitClient.reparacionService.crearReparacion(reparacionCompletada).enqueue(object : Callback<ReparacionDTO>
        override fun onResponse(call: Call<ReparacionDTO>, response: Response<ReparacionDTO>) {
            if (response.isSuccessful && response.body() != null) {
                Toast.makeText(requireContext(), "Solicitud procesada con éxito", Toast.LENGTH_SHORT).show()
            } else {
                Toast.makeText(requireContext(), "Hubo un error en la solicitud", Toast.LENGTH_SHORT).show()
            }
        }

        override fun onFailure(call: Call<ReparacionDTO>, t: Throwable) {
            Log.e("LOGIN", "Error de conexión", t)
            Toast.makeText(requireContext(), "Error al conectar a la API", Toast.LENGTH_SHORT).show()
        }
    })
}
```

Este fragmento de código se activa cuando el usuario pulsa el botón de “Pagar” para cerrar una reparación. Aunque el botón sugiere un pago, en esta versión de la aplicación no se ha implementado ninguna plataforma de pago real; la acción solo actualiza el estado de la reparación a “Completado” en el sistema.

Cuando se pulsa el botón, se crea un objeto con los datos de la reparación actual, cambiando su estado para indicar que ha finalizado. Luego, se envía esta información al servidor a través de una llamada API con Retrofit. Si la actualización es correcta, se muestra un mensaje que confirma que la reparación se ha cerrado con éxito; en caso de error, se informa al usuario.

Por tanto, esta funcionalidad únicamente sirve para marcar la reparación como completada, dejando abierta la necesidad de integrar un sistema de pagos real en futuras versiones.

```
    // Cargar servicios actuales de la reparación
    // y aplica un filtro para tolerancia a fallos

    listaServicios = reparacion?.servicios?.removeSuffix(";")?.split(";")?.filter { it.isNotBlank() && it.count

    serviciosAdapter = ServiciosAdapter(listaServicios,{posicion -> eliminarServicio(posicion)})
    recyclerView.layoutManager = LinearLayoutManager(requireContext())
    recyclerView.adapter = serviciosAdapter

    return vista
}
// Añade un servicio (formato: nombre:precio:cantidad) y actualiza la reparación en el backend
private fun anadirServicio(servicio: String, cantidad: String,precio: String) {

    val nuevoServicio = "$servicio:$precio:$cantidad;"
    reparacion?.servicios += nuevoServicio

    val reparacionModificada = ReparacionOutputDTO(
        id = reparacion?.id!!,
        fechaEntrada = reparacion?.fechaEntrada!!,
        estado = reparacion?.estado!!,
        servicios = reparacion?.servicios!!,
        motivos = reparacion?.motivos!!,
        vehiculoId = reparacion?.vehiculo?.id!!,
        facturaId = reparacion?.factura
    )
    pushearServicio(reparacionModificada)
    listaServicios.add(nuevoServicio)
    serviciosAdapter.notifyItemInserted(listaServicios.size - 1)
}
```

En la aplicación, los servicios asociados a una reparación se almacenan y gestionan como una cadena de texto con un formato específico para facilitar su manejo y transmisión. Cada servicio se representa con el formato nombre:precio:cantidad, y los servicios se concatenan en una sola cadena separados por el carácter punto y coma (;).

Por ejemplo, una cadena de servicios puede tener esta estructura:

"CambioAceite:25.0:1;RevisionFrenos:15.0:2;"

Para cargar los servicios actuales de una reparación, se procesa esta cadena eliminando el último punto y coma y dividiéndola en una lista, filtrando aquellos elementos que no cumplan con el formato esperado (es decir, que no contengan exactamente dos dos puntos :). Esto ayuda a mantener una cierta tolerancia a fallos y evita incluir servicios mal formateados.

Cuando se añade un nuevo servicio, este se concatena a la cadena de servicios con el formato indicado y un punto y coma al final. Luego, se crea un objeto de tipo ReparacionOutputDTO con los datos actualizados, que se envía al backend para actualizar la reparación.

Finalmente, la lista de servicios que se muestra en la interfaz se actualiza para reflejar los cambios, notificando al adaptador para que actualice la vista.

Configuración y pruebas

Durante el desarrollo, se configuró una dirección base en el proyecto móvil para apuntar al servidor local donde corre el backend. Debido a que este servidor puede cambiar de IP según la red utilizada, esta dirección es fácilmente configurable para facilitar el desarrollo y pruebas en distintos entornos.

Las pruebas se realizaron tanto de forma manual como con casos automatizados para asegurar el correcto funcionamiento de las funcionalidades clave, desde el registro e inicio de sesión hasta la gestión de citas y el chat.

Vías futuras

Para adaptar la aplicación a un entorno real y profesional, se plantean las siguientes mejoras:

- **Refuerzo de la seguridad:**

  Implementar control de acceso más estricto.

  Cifrado de datos sensibles.

  Protección de sesiones y comunicaciones WebSocket.

  Validación robusta de entradas y autenticaciones.
  Integración de un sistema de pagos:

  Permitir a los clientes pagar reparaciones directamente desde la app.

  Gestionar pagos seguros a través de pasarelas como Stripe o PayPal.

  Asociar pagos a reparaciones y generar comprobantes automáticos.

Mejoras en la lógica y funcionalidad:

Permitir la modificación y cancelación de citas por parte del cliente.

Asignación dinámica de mecánicos a reparaciones.

Envío de notificaciones automáticas (estado de reparaciones, recordatorios, etc.).

Optimización del flujo de trabajo en el backend para reflejar mejor la operativa real del taller.

Estimación de recursos y planificación

Fase 1: Planificación y Diseño (1 semana)

En esta etapa inicial, nos dedicamos a definir y concretar los objetivos del proyecto. Se diseñaron los MockUps en Figma para visualizar la interfaz y la experiencia de usuario, asegurando que el diseño fuese intuitivo y atractivo. Paralelamente, se elaboró el diagrama de casos de uso mediante Draw.io para comprender y documentar las diferentes interacciones entre los usuarios y el sistema. Finalmente, se establecieron los requisitos funcionales y no funcionales, sentando las bases para el desarrollo.

Fase 2: Desarrollo (7 semanas)

Durante esta fase, se implementaron todas las funcionalidades definidas en la planificación. Se utilizó Android Studio para la app móvil, con Kotlin y XML para la lógica y diseño, respectivamente. El backend se desarrolló con Spring Boot en Kotlin, integrando una base de datos MySQL para almacenar datos persistentes. Se implementaron funcionalidades clave como registro y autenticación de usuarios, gestión de vehículos, agenda de citas, historial de reparaciones y un sistema de chat en tiempo real mediante WebSockets.

Este periodo incluyó también la integración y ajuste de las diferentes partes, asegurando que la app funcionara de manera coordinada y eficiente.

Fase 3: Pruebas (1 semana)

Una vez desarrolladas las funcionalidades, se realizaron pruebas para garantizar el correcto funcionamiento. Estas pruebas incluyeron la validación de flujos completos de usuario, pruebas de rendimiento y detección de errores. Se hicieron ajustes en base a los resultados obtenidos para asegurar una experiencia estable y fluida.

Fase 4: Refactorización y Documentación (1 semana y 2 días)

En esta última fase, nos centramos en refactorizar el código para mejorar su calidad, legibilidad y mantenibilidad, asegurando que la estructura del proyecto sea sólida y fácil de extender o modificar en el futuro. Además, se perfeccionó la documentación técnica y de usuario, con el objetivo de facilitar la comprensión del sistema, su despliegue y uso posterior. Esta etapa es clave para garantizar la sostenibilidad del proyecto y apoyar futuras mejoras.

Diagrama de Gantt

null

Análisis:

Requisitos Funcionales

Ver Iniciar Sesión: Permite ver el login de la app.

Ver Registro: Es el registro de la aplicación

Registrarse: Es donde el usuario se crea una cuenta para el aplicativo

Iniciar Sesión: Registro e inicio de sesión de clientes con correo electrónico y contraseña, la app debe diferenciar entre los roles de cliente y mecánico, asignando permisos según cada perfil.

Ver Inicio: Vemos la pantalla principal de la aplicación

Ver Reparaciones : Consulta del historial de reparaciones en curso y finalizadas

Ver Reparaciones Finalizadas: Consulta del historial de reparaciones finalizadas

Ver Reparaciones en Curso: Consulta del historial de reparaciones en curso.

Ver Detalles: Ver los detalles de las reparaciones a tiempo real

Ver Chat: Chatear con el mecánico y con el cliente

Enviar Mensaje: Envian un mensaje a través del chat

Ver Añadir Vehiculo: Añade sus vehículos a la aplicación

Añadir vehículo: Registro de vehículos por parte de los clientes, incluyendo marca, modelo, matrícula y año.

Ver Pedir Cita: Al hacer click sobre el coche, se dispone el menú para pedir cita

Añadir Cita: Solicitud de citas en línea seleccionando fecha, hora y tipo de servicio.

Ver Factura: Ver la factura de las reparaciones

Ver Mecanico Inicio: Ver la página principal de administrador

Ver Reparaciones Abiertas: Ver las reparaciones que debe de hacer

Historial Reparaciones: Ver todas las reparaciones que ha tenido su vehículo

Ver Detalles Reparación Abierta: El usuario puede ver como va su reparación avanzando

Hacer Factura: El mecánico hace la factura de la reparación para el cliente

Ver Perfil: Permite a los usuarios ver su perfil.

Cerrar Sesión: Permite que los usuarios cierren su sesión activa.

Requisitos No Funcionales

Usabilidad: Interfaz intuitiva y fácil de navegar para clientes y mecánicos.

Diseño Responsivo: Adaptación del diseño a diferentes tamaños y resoluciones de pantalla para asegurar buena experiencia en dispositivos variados.

Mantenibilidad: Código modular y bien documentado para facilitar futuras mejoras y mantenimiento.

Compatibilidad: Soporte para dispositivos Android desde la versión 8.0 en adelante.

Rendimiento: Optimización de consultas a la base de datos para mejorar la velocidad de respuesta y garantizar una navegación fluida.

Conectividad: La aplicación requiere conexión a Internet para gestionar citas, consultar reparaciones y utilizar el chat en tiempo real.

Seguridad: Protección básica de datos sensibles, como contraseñas y facturas, para

garantizar la privacidad de los usuarios.

Escalabilidad: Capacidad para soportar un aumento de usuarios y datos sin afectar el rendimiento.

Tablas Casos de Uso

Caso de uso 1	Ver Iniciar sesión
Alias	
Actores	Usuario, Mecánico
Requisito funcional	Ver Iniciar Sesión
Descripción	Es el login de la aplicación
Referencias	Iniciar Sesión, Ver Registro
Comentarios	Ningún comentario

Caso de uso 2	Ver Registro
Alias	
Actores	Usuario, Mecánico
Requisito funcional	Ver Registro
Descripción	Es el registro de la aplicación
Referencias	Registrarse, Ver Iniciar Sesión
Comentarios	Ningún comentario

Caso de uso 3	Registrarse
Alias	
Actores	Usuario
Requisito funcional	Registrarse

Descripción	Es donde el usuario se crea una cuenta para el aplicativo
Referencias	Ver Registro
Comentarios	Ningún comentario

Caso de uso 4	Iniciar Sesión
Alias	
Actores	Usuario, Mecánico
Requisito funcional	Iniciar Sesión
Descripción	Login de la aplicación
Referencias	Ver Iniciar Sesión, Ver Mecanico Inicio, Ver Inicio
Comentarios	Ningún comentario

Caso de uso 5	Ver Inicio
Alias	
Actores	Usuario
Requisito funcional	Ver Inicio
Descripción	Vemos la pantalla principal de la aplicación
Referencias	Iniciar Sesión, Ver Reparaciones Abiertas, Historial Reparaciones, Ver Perfil
Comentarios	Ningún comentario

Caso de uso 6	Ver Reparaciones
Alias	
Actores	Mecánico

Requisito funcional	Ver Reparaciones
Descripción	Consulta del historial de reparaciones en curso y finalizadas
Referencias	Ver Inicio, Ver Reparaciones en Curso, Ver Reparaciones Finalizadas
Comentarios	Ningún comentario

Caso de uso 7	Ver Reparaciones Finalizadas
Alias	
Actores	Usuario
Requisito funcional	Ver Reparaciones Finalizadas
Descripción	Consulta de historial de reparaciones finalizadas
Referencias	Ver Reparaciones, Ver Facturas
Comentarios	Ningún comentario

Caso de uso 8	Ver Reparaciones en curso
Alias	
Actores	Usuario
Requisito funcional	Ver Reparaciones en curso
Descripción	Consulta de historial de reparaciones en curso
Referencias	Ver Detalles, Ver Reparaciones
Comentarios	Ningún comentario

Caso de uso 9	Ver Detalles

Alias	
Actores	Usuario
Requisito funcional	Ver Detalles
Descripción	Ver los detalles de las reparaciones a tiempo real
Referencias	Ver Chat, Ver Reparaciones en Curso
Comentarios	Ningún comentario

Caso de uso 10	Ver Chat
Alias	
Actores	Usuario, Mecánico
Requisito funcional	Ver Chat
Descripción	Chatear con el mecánico y con el cliente
Referencias	Ver Detalles, Enviar Mensaje
Comentarios	Ningún comentario

Caso de uso 11	Enviar Mensaje
Alias	
Actores	Usuario, Mecánico
Requisito funcional	Enviar Mensaje
Descripción	Envian un mensaje a través del chat
Referencias	Ver Chat
Comentarios	Ningún comentario

Caso de uso 12	Ver Añadir Vehículo

Alias	
Actores	Usuario
Requisito funcional	Ver Añadir Vehículo
Descripción	Añade sus vehículos a la aplicación
Referencias	Ver Inicio, Añadir Vehículo
Comentarios	Ningún comentario

Caso de uso 13	Añadir Vehículo
Alias	
Actores	Usuario
Requisito funcional	Añadir Vehículo
Descripción	Registro de vehículos por parte de los clientes, incluyendo marca, modelo, matrícula y año.
Referencias	Ver Añadir Vehículo
Comentarios	Ningún comentario

Caso de uso 14	Ver Pedir Cita
Alias	
Actores	Usuario
Requisito funcional	Ver Pedir Cita
Descripción	Al hacer click sobre el coche, se dispone el menú para pedir cita
Referencias	Ver Inicio, Añadir Cita

Comentarios	Ningún comentario

Caso de uso 15	Añadir Cita
Alias	
Actores	Usuario
Requisito funcional	Añadir Cita
Descripción	Es Solicitud de citas en línea seleccionando fecha, hora y tipo de servicio. login de la aplicación
Referencias	Ver Pedir Cita
Comentarios	Ningún comentario

Caso de uso 16	Ver Factura
Alias	
Actores	Usuario
Requisito funcional	Ver Facturas
Descripción	Ver la factura de las reparaciones
Referencias	Ver Reparaciones Finalizadas
Comentarios	Ningún comentario

Caso de uso 17	Ver Mecanico Inicio
Alias	
Actores	Mecánico
Requisito funcional	Ver Mecánico Inicio
Descripción	Ver la página principal de administrador
Referencias	Ver Reparaciones Abiertas, Historial

	Reparaciones, Ver Perfil, Iniciar Sesión
Comentarios	Ningún comentario

Caso de uso 18	Ver Reparaciones Abiertas
Alias	
Actores	Mecánico
Requisito funcional	Ver Reparaciones Abiertas
Descripción	Ver las reparaciones que debe de hacer
Referencias	Ver Mecanico Inicio, Ver Detalles Reparación Abierta
Comentarios	Ningún comentario

Caso de uso 19	Historial de Reparaciones
Alias	
Actores	Usuario
Requisito funcional	Historial de Reparaciones
Descripción	Ver todas las reparaciones que ha tenido su vehículo
Referencias	Ver Mecanico Inicio
Comentarios	Ningún comentario

Caso de uso 20	Ver Detalles de Reparación abierta
Alias	
Actores	Mecánico
Requisito funcional	Ver Detalles de Reparación abierta
Descripción	El usuario puede ver como va su reparación avanzando

Referencias	Ver Chat, Hacer Factura, Ver Reparaciones Abiertas
Comentarios	Ningún comentario

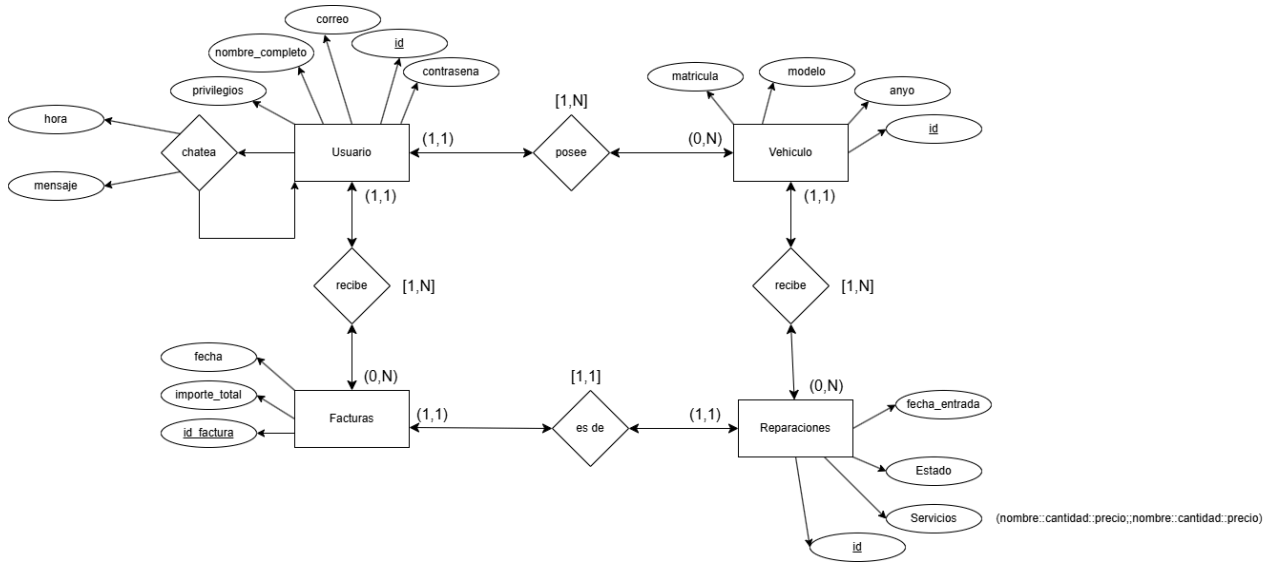
Caso de uso 21	Hacer Factura
Alias	
Actores	Mecánico
Requisito funcional	Hacer Facturas
Descripción	El mecánico hace la factura de la reparación para el cliente
Referencias	Ver Detalles Reparación Abierta
Comentarios	Ningún comentario

Caso de uso 22	Ver Perfil
Alias	
Actores	Usuario, Mecánico
Requisito funcional	Ver Perfil
Descripción	Permite al usuario ver su perfil
Referencias	Ver Mecanico Inicio,Cerrar Sesión, Ver Inicio
Comentarios	Ningún comentario

Caso de uso 23	Cerrar Sesión
Alias	
Actores	Usuario, Mecánico
Requisito funcional	Cerrar Sesión
Descripción	Permite al usuario Cerrar Sesión
Referencias	Ver Perfil,

Comentarios	Ningún comentario

Entidad-Relación:



Casos de usos

[Link casos de uso]

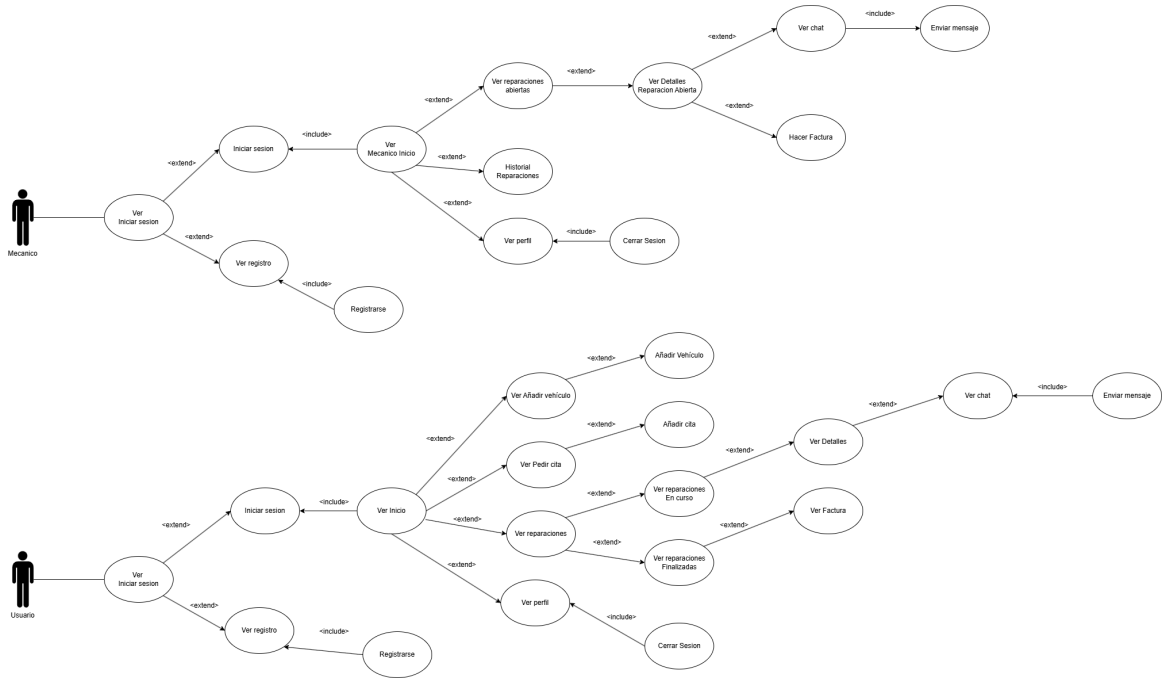
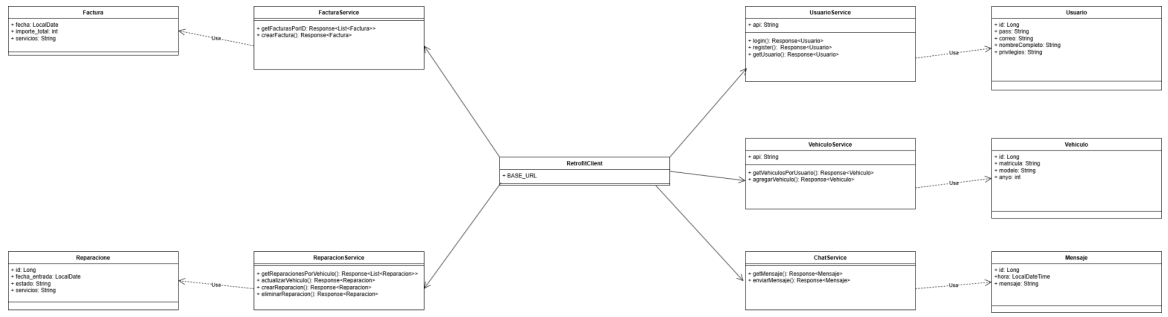


Diagrama de clases

[Link al Diagrama de clases]



Diseño

MockUp

[Link al Mockup]