Universidad Zaragoza
1542

Trabajo Fin de Grado

Desarrollo de aplicación móvil multiplataforma: Gestión de órdenes de trabajo en construcciones

Autor/es

Alejandro Catalán Tejedor

Director/es

Miguel Ángel Aranda Alcañiz
Jesús Gallardo Casero

Escuela Universitaria Politécnica de Teruel

2023

Resumen

Este Trabajo de Fin de Grado consiste en el desarrollo y la documentación de una aplicación móvil multiplataforma. El objetivo de este proyecto ha sido crear una herramienta para facilitar la gestión del trabajo en construcciones. Esta herramienta se utilizará para la creación y edición de partes de trabajo sobre una orden de trabajo. Además, otro propósito de este proyecto es servir de guía para aquellas personas interesadas en el aprendizaje del desarrollo de aplicaciones móviles multiplataforma, utilizando el framework Flutter, o para las que ya saben y quieren mejorar sus habilidades aprendiendo nuevos conceptos.

Abstract

This Final Degree Project consists of the development and documentation of a multiplatform mobile application. The objective of this project has been to create a tool to facilitate the management of construction work. This tool is accelerated for the creation and editing of work tickets on a work order. In addition, another purpose of this project is to serve as a guide for those people interested in learning how to develop cross-platform mobile applications, using the Flutter framework, or for those who already know and want to improve their skills by learning new concepts.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas aquellas personas que han contribuido de manera directa o indirecta en la realización de mi Trabajo de Fin de Grado.

Agradezco en primer lugar a mi tutor Miguel Ángel, por su orientación, su apoyo y su comprensión durante todo el proceso.

También deseo expresar mi gratitud a los docentes de la Escuela Universitaria Politécnica de Teruel por brindarme una educación de calidad, que me ha permitido obtener todos los conocimientos que poseo hoy en día.

Además, les doy las gracias a mis amigos por darme mucho ánimo durante mis años de estudio. Vuestras palabras de aliento siempre me ayudaron a superar aquellos obstáculos que surgieron durante el camino.

Por último, no puedo dejar de mencionar a mi familia, quienes han sido mi mayor fuente de apoyo y motivación. Habéis sido mi pilar durante todos estos años. Gracias por siempre creer en mí y por estar a mi lado cuando lo he necesitado, sin vosotros todo esto nunca hubiera sido posible.

A todas estas personas, gracias por vuestro tiempo, paciencia y contribución en la realización de este Trabajo de Fin de Grado y en la del resto de mi grado universitario. Vuestro apoyo ha sido crucial durante todo el recorrido.

¡Muchas gracias a todos!

ÍNDICE

1. Introducción

Este proyecto consiste en la creación y documentación de una aplicación móvil multiplataforma, llamada *GestorTareas*, con la finalidad de facilitar la gestión de órdenes trabajo a empresas situadas en el sector de la construcción.

La aplicación se utilizará para la creación y edición de partes de trabajo sobre una orden de trabajo. La finalidad es facilitar el registro y gestión del trabajo realizado y de los recursos empleados en una construcción.

Se listarán todas las órdenes de trabajo abiertas, y seleccionando una de ellas se podrán ver las indicaciones y los recursos a emplear. Estas instrucciones -o indicaciones- serán tenidas en cuenta por el operario a cargo del manejo del dispositivo con la aplicación instalada. Éste creará partes de trabajo registrando cómo se ha repartido el trabajo; registrando el trabajo realizado y los recursos utilizados.

1.1. Motivación

La industria de la construcción es una parte elemental en la sociedad. Las casas en las que vivimos, los edificios donde trabajamos, los caminos que recorremos, todo es el resultado de esta industria. Por esto mismo las empresas dedicadas a la construcción se enfrentan a numerosos obstáculos. Organizar todo el trabajo que hay que realizar y el ya realizado, puede llegar a ser bastante problemático si no se posee la herramienta adecuada. Además, si a causa de una mala organización se pierde información, podría suponer una gran pérdida para la empresa.

Fue entonces cuando me di cuenta de que había una necesidad clara de una solución que facilitara el trabajo a las empresas constructoras y a sus empleados. Así nació la idea de crear una aplicación móvil que pusiera fin al caos y brindara una herramienta eficiente y centralizada para gestionar órdenes y partes de trabajo.

Mi objetivo con esta aplicación es hacer que la vida de las empresas constructoras sea más sencilla. Desde la creación y asignación de tareas hasta el seguimiento del progreso del proyecto, la aplicación proporcionará una plataforma fácil de usar que simplificará y agilizará los procesos.

Además, esta aplicación también beneficiará a los trabajadores de campo. Dado que podrán ver a tiempo real todas las tareas demandadas y podrán gestionar el trabajo realizado de forma sencilla y eficiente.

Por otro lado, otra de las razones que me llevaron a desarrollar este proyecto, fue que, en mi empresa actual, Inycom, soy la única persona con conocimientos del framework Flutter. Esto es debido a que cuando me incorporé a la empresa surgió un proyecto demandado por un cliente, este cliente deseaba que utilizáramos Flutter como medio de desarrollo. Como en el equipo de desarrollo móvil (del que formo parte) mis compañeros sólo poseían conocimientos centrados en Android o iOS, se me encomendó investigar esta tecnología y desarrollar la aplicación. Después de esto seguí trabajando casi exclusivamente con Flutter. Como actualmente sigo siendo el único desarrollador con estos conocimientos, que han ido evolucionado durante todo este tiempo, pensé que podía ser buena idea aplicar en un proyecto todos los conocimientos adquiridos hasta el día de hoy.

Por este motivo el documento desarrollado, principalmente los requisitos específicos y el diseño, es lo más técnico posible, para que mis compañeros puedan ver el proceso real del desarrollo de una aplicación en Flutter, y consultar este documento siempre que lo necesiten.

1.2. Marco de trabajo

Para el desarrollo de esta aplicación móvil multiplataforma, se ha seleccionado un marco de trabajo que se basa en el uso de las siguientes herramientas y tecnologías:

- Flutter: Se utilizará el framework Flutter como base para el desarrollo de la aplicación. Flutter es una tecnología de código abierto desarrollada por Google que permite crear aplicaciones nativas para Android e iOS a partir de un único código base. Es altamente eficaz y personalizable, lo que lo convierte en la elección ideal para el proyecto. En el *apartado 1.6* se amplía el porqué de esta elección.
- Dart: Se utilizará el lenguaje de programación Dart, que es el lenguaje principal utilizado en el desarrollo con Flutter. Dart es un lenguaje que brinda una sintaxis clara y concisa, así como una gran cantidad de bibliotecas y herramientas para facilitar el desarrollo de la aplicación.
- SQLite: Se utilizará SQLite como herramienta para la creación y gestión de la base de datos. Se integrará en el código importando el paquete de Flutter utilizado para ello. En el *apartado 1.7* se amplía el porqué de esta elección.
- Git: Se utilizará un sistema de control de versiones, como Git, para mantener un registro de los cambios realizados en el código. Esto permitirá asegurar la integridad del código a lo largo del desarrollo del proyecto. El programa Fork se utilizará para gestionar repositorios Git haciendo uso de su interfaz gráfica. El repositorio será creado en Azure DevOps, debido a que es la herramienta utilizada en la empresa para crear y almacenar repositorios.

1.3. Audiencia

La aplicación móvil multiplataforma está diseñada pensando en las empresas y profesionales involucrados en el sector de la construcción. Está especialmente dirigida a las constructoras, porque les brinda una herramienta eficiente para agilizar y simplificar la gestión de las órdenes de trabajo. La aplicación les permitirá llevar un seguimiento organizado de las tareas y recursos asignados en cada proyecto. La empresa en sí es la más interesada, dado que con esta herramienta se podrán evitar pérdidas de información que perjudiquen el beneficio de la empresa. Además, los trabajadores de la empresa también se verán beneficiados con el uso de la aplicación:

- Gerentes y supervisores de proyectos: Los gerentes y supervisores de proyectos en el sector de la construcción serán los usuarios clave de la aplicación. Podrán utilizarla para supervisar el progreso de las órdenes de trabajo, asegurándose de que se cumplan las indicaciones y se utilicen los recursos de manera eficiente. Además, podrán acceder a los datos actualizados en tiempo real, lo que les permitirá tomar decisiones informadas y realizar ajustes según sea necesario.
- Operarios y técnicos de construcción: La aplicación también está diseñada para facilitar el trabajo diario de los operarios y técnicos que llevan a cabo las tareas en el sitio de construcción. Podrán acceder a las órdenes de trabajo asignadas, consultar las

instrucciones detalladas y los recursos requeridos, y registrar de forma sencilla el trabajo realizado y los recursos utilizados. Esto permitirá un seguimiento preciso del progreso de las tareas y asegurará una comunicación fluida entre el equipo de trabajo.

Por otra parte, el documento va dirigido a todos aquellos interesados en su lectura, si bien sean académicos, alumnos, compañeros de empresa en Inycom, etc. Y, sobre todo, va dirigido a aquellos clientes interesados en comprar la aplicación para su uso.

1.4. Tipo de publicación

La aplicación va dirigida a empresas constructoras. Podría ser pública o privada.

Por un lado, si fuese pública, habría un desarrollo unificado, no adaptado para cada empresa. No sería necesario hacer cambios pronunciados en la aplicación (ni en la funcionalidad, ni en el diseño). Tampoco sería necesario un mantenimiento y supervisión intensivo. Pero, a no ser que se decidiese añadir publicidad integrada, la empresa desarrolladora no obtendría beneficio del proyecto.

Por el otro lado, si fuese privada, aunque requiriese más esfuerzo, se adaptaría mucho mejor a las necesidades de cada empresa. Si se necesitase hacer algún cambio, de diseño o funcionalidad, se haría sin problema. Si surgiese algún inconveniente, la empresa desarrolladora podría arreglarlo inmediatamente. Tanto a las empresas constructoras que deseen utilizar la aplicación y garantizar la seguridad del sistema, como a la empresa desarrolladora que desee obtener beneficio del proyecto, les interesa que sea privada.

Es por esto por lo que se ha decidido que la aplicación sea de uso privado.

1.5. Plataforma de publicación

Dado que esta aplicación será privada y será vendida a empresas que deseen comprarla para su uso interno. No será pública, por lo que no se publicará en ninguna plataforma convencional de acceso universal, como Google Play Store o Apple Store.

La plataforma habitualmente utilizada en Inycom es AppCenter. Como el autor conoce esta herramienta, también se utilizará para este proyecto.

Esta plataforma pertenece a Microsoft y es de uso privado. Está diseñada para desarrolladores de aplicaciones móviles. Esta herramienta permite construir, probar y distribuir aplicaciones tanto para Android como para iOS. Algunas de las características son:

- Compilación automatizada: Permite a los desarrolladores compilar automáticamente las aplicaciones.
- Pruebas: Permite a los desarrolladores ejecutar pruebas automatizadas en dispositivos reales sin la necesidad de tener uno físicamente.
- Distribución de versiones: Se pueden distribuir las aplicaciones a los usuarios de forma sencilla. Además, AppCenter permite crear grupos de usuarios, por ejemplo, un grupo de testers donde no se incluya a los usuarios finales, para probar ciertas características de una versión que todavía no está preparada para que use el cliente.
- Monitoreo y análisis: Existen herramientas dentro de AppCenter que permiten monitorear y analizar datos en tiempo real, lo que permite a los desarrolladores obtener información sobre el rendimiento de la aplicación.

1.6. Tecnología de desarrollo

Como se menciona en los apartados anteriores, se ha escogido Flutter como framework para desarrollar la aplicación, utilizando Dart como lenguaje de programación.

Flutter es un framework desarrollado por Google. Cuando lo desarrolló la compañía, tenían pensado hacer un uso interno de él, pero viendo el gran potencial que tenía, decidieron sacarlo como un framework de código abierto.

Este framework posee una gran cantidad de ventajas respecto a las tecnologías que existían hasta el momento. Primero de todo, sirve para desarrollar tanto en Android como en iOS, es decir, para el desarrollo de aplicaciones multiplataforma. Compila el código de forma nativa, lo que lo hace tremendamente rápido, siendo posible ver los cambios realizados en el código a tiempo real, utilizando la llamada *hot reload*, sin necesidad de compilar o ejecutar el proyecto de nuevo. Contiene una gran cantidad de librerías por sí mismo y también existen miles de paquetes para añadir al proyecto si se desea. Añadir paquetes al proyecto es muy sencillo, basta con ejecutar un comando en la terminal del entorno de programación (VS Code en el caso de este proyecto), también se puede añadir a mano indicando el nombre y la versión del paquete en el archivo destinado para ello. Es muy adaptable gráficamente, se pueden combinar los elementos de numerosas formas. A diferencia de otros lenguajes de programación utilizados en el desarrollo de aplicaciones móviles como Objetive-C, el lenguaje de programación que utiliza, Dart, es sencillo de aprender, siendo similar a otros lenguajes de programación orientados a objetos. El propio Dart es utilizado para crear los elementos gráficos de la aplicación, no es necesario utilizar otros lenguajes como en el caso de la programación Android, donde se suele utilizar Java haciendo uso de XML para la interfaz gráfica. Además, cuenta con una extensa documentación y comunidad, todavía menor que otras comunidades de desarrollo de aplicaciones móviles, como la de React (también utilizada para desarrollo multiplataforma), pero lo suficientemente grande como para esclarecer las posibles dudas que le surgiesen al programador. Otra ventaja que cabe mencionar es que permite la integración de código nativo en el propio proyecto.

Como se puede observar son múltiples las ventajas de Flutter sobre el resto de frameworks o tecnologías empleadas en el desarrollo de aplicaciones móviles. Sin embargo, también existe alguna desventaja, aunque en menor medida. Una de ellas es que los tamaños de las aplicaciones suelen ser más grandes que las aplicaciones nativas. A parte de esto, la mayor desventaja que puede tener Flutter es la curva de aprendizaje, si bien es cierto que la sintaxis es parecida a Java y otros lenguajes de programación similares, Dart cuenta con conceptos de programación avanzados y difíciles de aprender. Uno de estos conceptos avanzados es el uso del patrón BLoC (Bussiness Logic Component) como arquitectura de la aplicación para la gestión de eventos y la generación de estados a partir de estos eventos, el cual tiene una curva de aprendizaje muy pronunciada. Aunque es cierto que se pueden crear aplicaciones en Flutter sin utilizar este concepto, utilizando patrones como el MVVM (Model View View-Model), surgen dos inconvenientes: el primero es que el patrón BLoC ofrece una gran cantidad de ventajas y prestaciones al utilizarlo, así que no hacer uso de él, hace que el código sea menos óptimo (en el punto 5.2 se ve el por qué); la segunda es que existen otros conceptos avanzados en Dart, además del patrón BLoC, que hay que aprender igual, como los Futures, async/awaits,

Providers, etc. Además, Dart cuenta con multitud de elementos gráficos complejos que un programador de front-end debe saber utilizar.

Dado que el autor tiene una extensa experiencia con este framework y que la desventaja principal, la curva de aprendizaje, no aplica en este caso, se mantiene la elección de desarrollar la aplicación con Flutter.

1.7. Elección del motor de base de datos

Para la base de datos del proyecto se ha decidido utilizar SQLite. El autor había trabajado previamente con el framework Drift (Moor) que permitía crear y gestionar bases de datos utilizando Dart para realizar consultas, en lugar de SQL. Sin embargo, al utilizar archivos autogenerados para conseguir lidiar con la base de datos usando Dart, había que utilizar comandos de generación de código, lo que provocaba que muchas veces se perdiese más tiempo del que se ganaba. Además, cada cambio en la estructura de la base de datos requería aumentar la versión de ésta y obligaba a tener que borrarla y crearla de nuevo. Aunque en la documentación se especificase que sí es posible crear tablas surgidas de la relación de otras tablas, no funcionaban correctamente las claves ajenas. Por todos estos motivos se decidió usar la herramienta más extendida para la creación de bases de datos relacionales, SQLite. SQLite es muy rápida, intuitiva y trabaja con el lenguaje SQL, con el que el autor había tratado extensamente a lo largo del grado universitario.

1.8. Definiciones, acrónimos y abreviaturas

Término	Descripción
OT	Orden de trabajo.
PT	Parte de trabajo.
Maestros	Recursos.
AWS	Amazon Web Services.
Background	En segundo plano.
App	Aplicación.
Tester	Probador de aplicaciones.
Card	Elemento gráfico que se utiliza para representar alguna información relacionada, por ejemplo, un álbum, una ubicación geográfica, una comida, detalles de contacto, etc.
UI	Interfaz de usuario.
Framework	Estructura de herramientas que facilita el desarrollo de aplicaciones de software al proporcionar funcionalidades predefinidas y una estructura organizada.
ListView	Lista vertical de elementos en una aplicación.

2. Descripción general

2.1. Perspectiva del producto

2.1.1. Interfaces de sistema

- *GestorTareas* deberá obtener las órdenes de trabajo y los recursos a través de la API del servidor, así como enviar los partes de trabajo para su almacenamiento.
- El servidor será interno, creado y gestionado por el equipo de la parte web.

2.1.2. Interfaces de usuario

- La interfaz de usuario se encuentra en el *Anexo 1*.

2.1.3. Interfaces software

- Se recibirán los datos enviados por la parte web a través de peticiones a una API propia.
- La aplicación ha de mantener datos almacenados de forma persistente. Para esto se utilizará una base de datos integrada.

2.1.4. Interfaces de comunicaciones

- *GestorTareas* es una aplicación móvil que deberá tener conexión online y conocer la hora actual, por esto deberá utilizar las interfaces de comunicaciones inalámbricas del dispositivo: datos móviles y geolocalización.

2.1.5. Restricciones de Memoria y Almacenamiento

- La aplicación móvil estará disponible para las plataformas Android y iOS. Por lo tanto, se deben considerar los requisitos de memoria y almacenamiento específicos de cada sistema operativo.
- Para dispositivos móviles Android, se requerirá un mínimo de 4 GB de memoria RAM y 16 GB de almacenamiento para un rendimiento óptimo. Se recomienda que los dispositivos utilicen Android 7.0 o superior.
- Para dispositivos móviles iOS, se requerirá un mínimo de 2 GB de memoria RAM y 16 GB de almacenamiento para un rendimiento óptimo. Se recomienda que los dispositivos utilicen iOS 12 o superior.
- Además, la aplicación requerirá permisos de cámara para capturar imágenes relacionadas con los partes de trabajo. Estos permisos deben solicitarse al usuario al momento de la instalación y deben ser compatibles con los sistemas operativos Android y iOS.
- Se utilizará una base de datos local donde se almacenará la información recibida por la web, además de los partes de trabajo no sincronizados con la API. Por esto mismo también se pedirán al usuario permisos de almacenamiento.

2.1.6. Modos de Operación

- Hay un único modo de operación: usuario estándar.

En el *apartado 2.2* se detallará las características de este modo de operación.

2.1.7. Necesidades de infraestructura (para el alojamiento u operación)

- La aplicación se distribuirá a través de un archivo APK descargable en la plataforma AppCenter de Microsoft. Los clientes podrán acceder a la aplicación a través de esta plataforma.
- Al utilizar AppCenter, se garantiza que la aplicación no estará disponible en plataformas convencionales de acceso universal, como Google Play Store o Apple App Store.
- La plataforma AppCenter también permite una fácil gestión de versiones y actualizaciones de la aplicación para los clientes.
- Será necesario un servidor de donde se recojan y almacenen los datos.
- El servidor será una instancia de AWS (pactado con el cliente), por lo que el hardware será una infraestructura externa.

2.2. Funcionalidad del Producto

Debido a que las credenciales serán generadas y otorgadas por la empresa compradora de la aplicación, únicamente habrá un tipo de usuario, capaz de iniciar sesión, pero no de registrarse.

- Usuario
    - Iniciar sesión.
        - El usuario podrá iniciar sesión con las credenciales otorgadas por la compañía.
    - Cerrar sesión.
        - Se podrá cerrar sesión, si así lo desea el usuario. Si bien es cierto que cada usuario tendrá su dispositivo personal.
    - Refrescar datos.
        - Se podrán refrescar los datos manualmente. Cuando el usuario disponga de conexión, podrá querer probar a refrescar los datos manualmente para descargar las órdenes y recursos asociados a las mismas.
    - Visualizar lista OTs.
        - El usuario podrá ver la lista de órdenes de trabajo. Es la pantalla principal al iniciar sesión.
    - Seleccionar OT.
        - Dentro de la lista de órdenes de trabajo, mencionada en el punto anterior, se podrá seleccionar la orden de la que se desee ver el detalle.
        - Visualizar maestros. Se podrán visualizar los maestros (personal, materiales y maquinaria) asociados a una orden, desde el detalle de ésta.
    - Visualizar lista PTs.

        - Desde una orden de trabajo, se podrán visualizar la lista de partes de trabajo asociados a esa orden.
    - Crear PT.
        - En la lista de partes de trabajo existirá un botón desde el cual se podrá crear un nuevo parte. Editando y añadiendo sus campos y recursos.
    - Editar PT.
        - En la lista de partes también se podrá seleccionar un parte para ver y/o editar los campos de éste.
        - Editar maestros. Además de editar los parámetros también se podrán editar los maestros asociados a ese parte, tanto quitar, como modificar, como añadir nuevos recursos.
        - Cerrar parte. Haciendo uso de un botón, se podrá cerrar el parte, siendo así enviado al servidor y no editable de nuevo.

2.3. Restricciones

2.3.1. Auditoría

Se le dará al comprador un mes de tiempo máximo para probar la aplicación y notificar el cambio de algún aspecto de diseño relacionado con la aplicación. Esto permitirá realizar los ajustes necesarios para cumplir con las expectativas del usuario y mejorar la UX.

También, se dispondrá de tres años de garantía para poder corregir cualquier posible error o vulnerabilidad identificada durante el uso de la aplicación. Con esto se garantiza el correcto funcionamiento, además del rendimiento y la seguridad.

2.3.2. Protocolos de comunicaciones

La aplicación se comunicará con el servidor a través de una API propia mediante el protocolo HTTPS y datos en formato JSON.

2.3.3. Fiabilidad

Se contará con una base de datos interna para que no se pierdan datos si estos no se han sincronizado correctamente. El propósito es no perder absolutamente ninguna información. Todos los datos creados se guardarán instantáneamente en la base de datos, desde la creación de un parte y sus recursos, hasta cada letra cambiada en alguno sus campos. No se permitirá al usuario guardar o borrar información manualmente. Únicamente cuando los datos se hayan sincronizado correctamente con el servidor, y sólo entonces, se eliminarán de la base de datos y dejará de ser visible para el usuario. La fiabilidad en la persistencia de la información es crucial, puesto que la aplicación será utilizada por empresas en el área de la construcción, y cualquier pérdida de información en el registro de sus actividades puede suponer un gran costo para la empresa.

2.3.4. Seguridad

Se otorgará un token al comprobar que las credenciales introducidas son válidas para así verificar al usuario y poder iniciar sesión. Este token será un Bearer Token, el tipo de token utilizado en el protocolo OAuth 2.0. Por lo tanto, sin un token válido, la aplicación no permitirá intercambios de datos con la API. Éste es el único punto en la aplicación móvil que afecta a la seguridad, del resto la responsabilidad es responsabilidad del back-end.

2.4. Asunciones y dependencias

- Se asume que el modo offline no se selecciona manualmente, si no que, cuando no exista una conexión estable, se activará automáticamente. El usuario no percibirá ningún cambio.
- Se asume que la aplicación no podrá ser utilizada horizontalmente.

3. Requisitos específicos

3.1. Interfaces externos

3.1.1. Interfaces de usuario

RQ 1. Los colores principal y secundario, así como el logo, serán indicados por la empresa compradora. La aplicación por defecto llevará los colores: rojo, blanco y gris claro. Por defecto llevará el logo de Inycom.

RQ 2. El tamaño será adaptable a las dimensiones del dispositivo. Siempre que se encuentre en el rango de dimensiones de un celular estándar, esto quiere decir que no está diseñado para tamaños de dispositivos tipo Tablet.

RQ 3. Cada pantalla (excepto la principal y la de inicio de sesión) contará con una flecha orientada hacia la izquierda en la esquina superior izquierda, ésta se utilizará para regresar a la pantalla anterior.

3.1.2. Interfaces software

RQ 4. La aplicación obtendrá los datos de la API, así como enviará los datos correspondientes mediante un formato JSON.

RQ 5. Se utilizará la API para toda comunicación con el servidor.

RQ 6. Se utilizará una base de datos integrada que guardará la información pertinente.

3.1.3. Interfaces de comunicaciones

RQ 7. La comunicación entre la aplicación y el servidor será a través de peticiones a la API mediante el protocolo HTTP. El formato será el mencionado en el *RQ 8*.

RQ 8. Las peticiones y respuestas serán métodos GET y POST. Seguirán el formato de intercambio de datos JSON. Fácil de leer y escribir para los humanos, y de parsear y generar para las máquinas.

3.2. Características del sistema

La enumeración de los requisitos funcionales de *GestorTareas,* va a organizarse a partir de una serie de grupos de escenarios de utilización. Estos grupos son:

- Arranque / Parada
- Eventos periódicos
    - Descarga de datos
    - Envío de partes
- Escenarios del usuario

- Iniciar sesión
- Cerrar sesión
- Refrescar datos
- Visualizar lista OTs
- Seleccionar OT
- *Visualizar maestros de OT*
- Visualizar lista PTs
- Crear PT
- Visualizar/Editar PT
  - Cerrar PT
- *Visualizar/Editar maestros de PT*

** Nota: el escenario 'Visualizar/Editar maestros de PT' es común a los escenarios 'Crear PT' y 'Visualizar/Editar PT', siendo una extensión de éstos. Lo mismo ocurre con el escenario 'Visualizar maestros de OT', es común al escenario 'Seleccionar OT'.*

3.2.1. Arranque / Parada

ESC1. Arranque

- Propósito

El escenario de arranque es habitual (y necesario) en todos los sistemas informáticos. Contiene toda esa serie de acciones que deben llevarse a cabo cuando se inicia la ejecución de la aplicación (proceso de arranque/boot), de forma previa a que dicha aplicación esté en condiciones de ser utilizada por sus usuarios.

- RQs asociados

RQ 9. En la fase de arranque se deberá comprobar el idioma en el que está configurado el dispositivo móvil para así utilizar los literales (textos) asociados a este idioma.

RQ 10. En la fase de arranque también se consultará la información de permisos otorgados por el sistema. Dado que son necesarios varios permisos, como el almacenamiento interno, se consultará si están permitidos o no. Si no lo están, se preguntará al usuario si desea otorgar los permisos necesarios.

ESC2. Parada

- Propósito

El escenario de parada también es necesario en toda aplicación informática. En él, la aplicación ya habrá dejado de estar operativa para sus usuarios, pero todavía deberán ejecutarse algunas acciones antes de que el sistema operativo pueda considerarla como finalizada.

- RQs asociados

RQ 11. Al cerrar la aplicación se comprobará que todos los partes estén sincronizados, en caso de no estar sincronizados saldrá un popup de confirmación con un título

de alerta y un mensaje avisando que todavía existen partes no sincronizados y si está seguro de que desea cerrar la aplicación.

RQ 12. Se cerrarán todas las vistas y se liberarán los recursos utilizados por la aplicación, excepto el almacenamiento utilizado por la base de datos.

3.2.2. Eventos periódicos

ESC3. Descarga de datos

RQ 13. Cuando se pulse el botón de refrescar, situado en el menú lateral (*Ilustración 3*), se llamará a la API para obtener todos los datos. Si no hay conexión, o aun si habiendo conexión, no es estable, y no se consigue obtener los datos, se realizará la llamada automáticamente cada 10 minutos.

RQ 14. Cuando se obtengan los datos correctamente se le notificará al usuario mediante un popup, debido a que, si no se han conseguido obtener en el primer intento, el usuario no sabrá cuándo se han descargado correctamente.

ESC4. Envío de partes

RQ 15. Cuando se cierra un parte (*Ilustración 15*) se intenta enviar al servidor para que lo almacene. Si cuando se intente cerrar el parte no se consigue debido a la conexión, se tratará de enviar cada 10 minutos, notificándoselo así al usuario cuando se haya logrado enviar correctamente. Ocurre lo mismo que al descargar datos *ESC3*.

3.2.3. Escenarios asociados al usuario

ESC5. Iniciar sesión

RQ 16. Para iniciar sesión se deberán introducir las credenciales en sus campos correspondientes. Dado que las credenciales son creadas por la parte web, éstas se le comunicarán personalmente al usuario y éste deberá introducirlas para iniciar sesión. *Ilustración 1*.

RQ 17. Cada vez que caduque el token de inicio de sesión (TBD), se tendrá que volver a iniciar sesión, debido a que, si no, no funcionarán las funciones relacionadas con la transmisión de datos.

RQ 18. La primera vez que se inicie sesión se almacenarán las credenciales y el token para poder ser utilizados cuando no haya señal, o la señal sea débil.

RQ 19. Para enviar las credenciales se deberá pulsar el botón continuar. *Ilustración 1*.

RQ 20. Una vez se haya verificado el login correctamente, se redirigirá automáticamente a la pantalla que muestra la lista de órdenes de trabajo. *Ilustración 4*.

ESC6. Cerrar Sesión

RQ 21. El cierre de sesión se realizará desde el menú lateral, pulsando el botón 'Salir'. *Ilustración 3*.

RQ 22. Cuando se cierre sesión se eliminarán automáticamente las credenciales y el token del sistema para no quedar registrados al volver a iniciar sesión.

ESC7. Refrescar datos

RQ 23. Se podrán refrescar los datos desde el botón 'Refrescar' en el menú lateral. *Ilustración 3*.

RQ 24. Al refrescar los datos se esperará haber recibido correctamente las órdenes y los datos maestros. En caso de ser así, se borrarán de la base de datos los datos anteriores para sustituirlos por los nuevos.

RQ 25. El botón refrescar también intentará enviar los partes no sincronizados. En caso de que no se hayan sincronizado tampoco al darle al botón manualmente, se reiniciará el contador, probando envíos automáticamente cada 10 minutos.

RQ 26. Las ejecuciones en segundo plano están contempladas en el *ESC3*.

ESC8. Visualizar lista de OTs

RQ 27. La pantalla con la lista de órdenes de trabajo se visualiza automáticamente al ser redirigido desde el inicio de sesión, cuando éste ha sido correcto. *Ilustración 4*.

RQ 28. Cada orden de la lista de órdenes se verá como un card, mostrando un pequeño resumen que describa la orden. Se mostrarán los campos con el número de la orden (identificador), con la fecha de inicio y con el trabajo a realizar.

RQ 29. Estos cards se agruparán en una sección deslizable de la pantalla, por si no entrasen todas las órdenes en una vista.

RQ 30. Se podrá buscar una o varias órdenes concretas en el buscador superior de órdenes. Este buscador tendrá en cuenta el número de orden. Si se buscase "10", aparecerían la 10, la 100, la 1089, etc. *Ilustración 5*.

RQ 31. Se podrá seleccionar una de esas órdenes para ver en detalle, lo que llevará al *ESC 9*. *Ilustración 6*.

ESC9. Seleccionar OT

RQ 32. Para seleccionar una orden se deberá pulsar en una de las órdenes de la pantalla donde se encuentran la lista de órdenes disponibles. *Ilustración 5*.

RQ 33. Cuando se haya seleccionado una orden se redirigirá a una pantalla que muestre el detalle de dicha orden. En esta pantalla aparecerán los campos: 'Fecha de inicio, 'Fecha de fin, 'Tipo', 'Instalación', 'Código orden cliente', 'Observaciones' y 'Trabajo a realizar'. *Ilustración 6*.

RQ 34. Además de los campos mencionados, aparecerán tres botones secundarios con los textos: 'Personal', 'Materiales y 'Maquinaria. Al pulsar cada uno de estos botones, el usuario será redirigido a la pantalla que muestre la respectiva información de cada uno. *Ilustración 7*. *ESC10*.

RQ 35. Por último, deberá aparecer un botón principal que mostrará el texto 'Ver partes de trabajo', este botón redirigirá a la pantalla con la lista de partes asociados a la orden seleccionada actualmente. *Ilustración 7*.

ESC10. Visualizar maestros de OT

RQ 36. Cada una de las tres pantallas mostrará una tabla con la columna 'Descripción' y 'Horas' (o 'Unidades' para los materiales), donde aparecerán los valores pertinentes asociados a la orden de la que se viene. *Ilustraciones 8, 9 y 10*.

ESC11. Visualizar lista de PTs

RQ 37. Se accederá a través de una orden de trabajo, se visualizarán los partes de trabajo asociados a dicha orden. *Ilustraciones 11 y 12*.

RQ 38. Se podrá buscar uno o varios partes concretos en el buscador superior de partes. Este buscador tendrá en cuenta el número de parte. Si se buscase "10", aparecerían el 10, el 100, el 1089, etc. *Ilustración 13*.

RQ 39. Al igual que las órdenes, los partes se mostrarán también en forma de cards, mostrando un resumen del parte con los parámetros: número de parte (identificador) y fecha de inicio. *Ilustración 12*.

RQ 40. Se distinguirán los partes cerrados no sincronizados de varias formas. La primera: se mostrará un tercer parámetro en las cards de los partes de trabajo ya cerrados. También, cambiará el color, aplicando una capa para darle un tono desactivado. Por último, no serán pulsables. *Ilustración 12*.

RQ 41. Los partes cerrados y sincronizados no se mostrarán, siendo así eliminados de la vista de partes y de la base de datos.

RQ 42. La lista de partes será deslizable, para poder ver los que no se muestran a primera vista.

RQ 43. Desde esta pantalla se podrá crear un nuevo parte mediante un botón con el símbolo más, o visualizar uno ya existente. *Ilustraciones 11, 12 y 13*. *ESC12* y *ESC13*.

ESC12. Crear PT

RQ 44. Desde la pantalla de visualización de la lista de partes dada una orden determinada, se podrá crear un nuevo parte asociado a esta orden. *Ilustraciones 11, 12 y 13*.

RQ 45. Cuando se pulse el botón se creará el parte automáticamente en la base de datos, para así prevenir pérdidas de información si la aplicación se cerrase repentinamente.

RQ 46. A este nuevo parte se le asignará automáticamente un id (número de parte único). También se añadirá automáticamente como parámetro la fecha de inicio actual. *Ilustración 14*.

RQ 47. El resto de los parámetros (observaciones y trabajo realizado) se crearán en blanco, pudiendo ser editados por el usuario. Cuando éstos se editen, se modificarán también en la base de datos. *Ilustración 14*.

RQ 48. Desde esta pantalla de creación de parte de trabajo se contemplarán también el ESC 13. *Ilustración 14*.

RQ 49. El parte se guardará automáticamente en todo momento, es decir, no es posible eliminarlo. Esto se hará para evitar cualquier posible pérdida de información. No existirán botones para guardar o cancelar la creación de un parte. Sus campos si pueden ser modificados. Si se hubiese cometido un error se debería hablar con el responsable para solucionarlo desde la central. Esto es debido a que, en una empresa de construcción, cualquier pérdida de datos puede suponer una gran pérdida económica.

ESC13. Visualizar/Editar PT

RQ 50. Desde la pantalla de visualizar lista de partes dada una orden determinada se podrá editar un parte de trabajo asociado a una orden. *Ilustraciones 12 y 13*.

RQ 51. Se pulsará sobre el parte que se desee visualizar y/o editar. Cuando se realice esta acción se mostrará la pantalla con el detalle del parte. Ahí se podrán editar o visualizar los parámetros que se desee. *Ilustración 15*.

RQ 52. Desde esta pantalla de detalle de parte de trabajo se contemplarán también el *ESC 14*. *Ilustración 15*.

RQ 53. **Cerrar parte (ESC13.1).** Si se ha acabado con un parte de trabajo, se podrá enviar al servidor pulsando el botón de cerrar parte. Si se cuenta con conexión y se recibe correctamente en el servidor, el parte se eliminará de la base de datos y ya no se verá en la lista de partes. En caso de que no se haya podido sincronizar, permanecerá en la lista de partes, pero se mostrará con fecha de fin, a diferencia del resto, y no será pulsable. *Ilustración 15*.

ESC14. Visualizar/Editar maestros de PT

RQ 54. Hay tres pantallas diferentes para los tres tipos de datos maestros. Estas pantallas son: personal, materiales y maquinaria. Las tres pantallas son prácticamente iguales, eso sí, con sus respectivos títulos y datos. *Ilustraciones 16-27*.

RQ 55. Estas pantallas estarán organizadas por cards, al igual que los resúmenes de órdenes y partes en sus respectivas listas. En esta card aparecerán dos campos: uno para las horas y otro para los minutos (un sólo campo en el caso de los materiales). Cuando se cambie alguno de los valores de un elemento se podrán aplicar los cambios para guardar la información o resetear para dejarla como estaba. *Ilustraciones 16-27*.

RQ 56. Tanto en la pantalla de creación de parte como en la de visualización/edición, se podrán gestionar los datos maestros mediante tres botones, uno para cada tipo de dato maestro. En la pantalla de creación empezarán establecidos todos a 0, pudiendo ser añadidos o borrados. En la pantalla de visualización/edición aparecerán como hayan sido editados. *Ilustraciones 14 y 15.*

3.3. Requisitos de Persistencia

RQ 57. Deberá persistir toda la información almacenada en la base de datos.

RQ 58. Cuando se actualice versión debería eliminarse la aplicación entera, incluidos todos los datos que deberían permanecer intactos de normal. Esto es debido a que, si en una nueva versión se actualizase la estructura de la base de datos, daría problemas con la base ya existente.

3.4. Restricciones de diseño de la interfaz de usuario

RQ 59. Se diseñará la aplicación siguiendo los patrones de estilo mencionados (colores, títulos, botones, iconos, etc.). Pudiendo cambiar los colores y el logo a petición del cliente. Además de tamaños u otros retoques por usabilidad.

RQ 60. El diseño debe ser lo más sencillo e intuitivo posible, esto es debido a que el usuario promedio que utilizará la aplicación tendrá un perfil de conocimiento tecnológico bajo. Por este motivo es por el que se han evitado otras formas de acceso a pantallas como menús laterales.

3.5. Restricciones de diseño arquitectónico

RQ 61. El diseño del software de la aplicación se realiza bajo el principio de programación orientada al UI.

RQ 62. El lenguaje que se va a utilizar, en coherencia con el requisito anterior, es Dart. Se elige este lenguaje debido a que está optimizado para la interfaz de usuario.

RQ 63. Se utilizará el patrón de diseño BLoC, patrón propio de Flutter. Es un patrón muy similar al MVVM (Modelo-Vista-VistaModelo).

4. Diseño

Véase que, aunque este documento no recoja los requisitos y funcionalidad del servidor, se incluyen los casos e interacciones (únicamente los relacionados con la app móvil) en la parte de diseño para un mayor entendimiento del funcionamiento de la aplicación.

Explicaciones, aclaraciones y suposiciones generales:

- Para mejorar la comprensión del lector, se ha optado por utilizar el término "maestro" como una generalización para representar a las personas, materiales y maquinaria en todos los diagramas. En lugar de repetir las clases, funciones, elementos, etc., para cada tipo de maestro, se ha simplificado la representación utilizando únicamente el término "maestro". Esto se ha hecho con el propósito de evitar confusiones y reducir la complejidad visual de los diagramas.
- Siempre que se borren los datos de la base de datos, bien sea en el inicio de sesión o al refrescar, nunca se borrarán los partes no sincronizados con el servidor. Éstos permanecerán en la base de datos hasta que consigan sincronizarse.
- Siempre que se sincronice un parte de trabajo, se borrará automáticamente.
- Los IDs de todas las órdenes de trabajo, partes de trabajo y maestros se generarán siempre automáticamente.
- Cuando se introduzca o extraiga un dato de la base de datos se debe supone que, mientras los parámetros sean correctos, siempre funcionará.
- Las fechas se usarán siguiendo el formato ‘dd/MM/yyyy – hh/mm’.
- Se debe suponer que en todas las llamadas a la API se utilizará el bearer token como método de comprobación.

4.1. Casos de uso

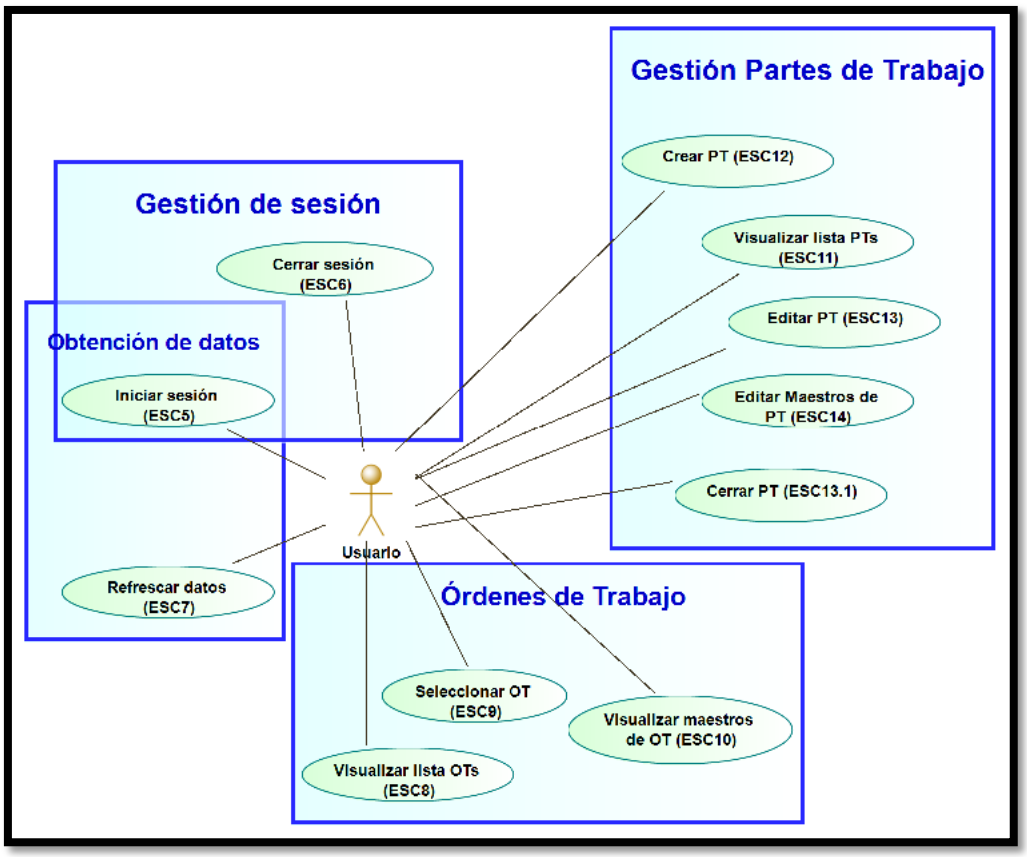


Figure 1: Diagrama de Casos de Uso del Cliente 'GestorTareas'

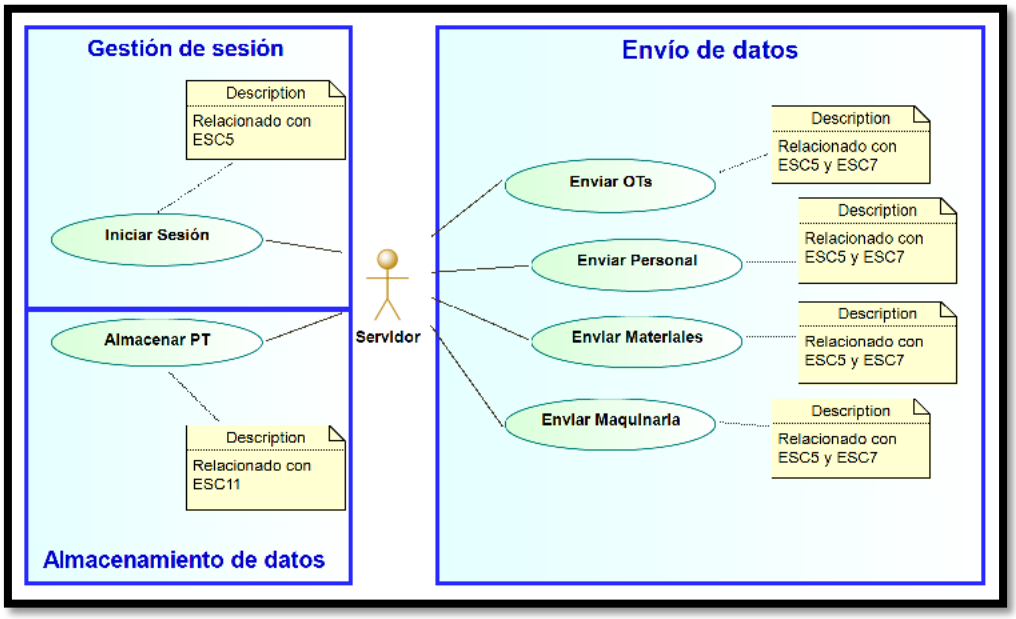


Figure 2: Diagrama de Casos de Uso del Servidor 'GestorTareas'

4.2. Diagramas de clases

4.2.1. Diagrama de clases del cliente

Explicaciones, aclaraciones y suposiciones:

- Las clases *OrdenTrabajo*, *ParteTrabajo* y *Maestro* heredan de la clase *ObjectWithMap* el parámetro *id* y las funciones *fromMap* y *toMap*. Debido a que las funciones se sobrescriben, se han representado de todas formas para cada tabla. En cambio, los ids no lo hacen, por lo que se han omitido.
- A diferencia del punto anterior, las clases *OrdenMaestro* y *ParteMaestro* no extienden de *ObjectWithMap* debido a que su clave primaria está compuesta por las claves ajenas que conforman al objeto. Así que no hereda el id, y las funciones no se sobrescriben.
- Las clases *OrdenTrabajoDao*, *ParteTrabajoDao* y *MaestroDao* heredan de *BaseDao* las funciones CRUD básicas y el atributo *tableName*.
- A diferencia del punto anterior, las clases *OrdenMaestroDao* y *ParteMaestroDao* no heredan de *BaseDao*. Por esto mismo, son los únicos DAOs donde se han representado los métodos CRUD, además de en *BaseDao*.
- Se entiende que todo parte de trabajo debe estar asociado a una orden de trabajo, es por ese motivo por el que se representa la relación como una agregación y no como una asociación, porque un parte no puede existir sin una orden.

Diagrama 1

4.2.1. Diagrama de clases del servidor

Explicaciones, aclaraciones y suposiciones:

- El diagrama de clases del servidor es únicamente orientativo y no debe seguir estrictamente, pues como se ha mencionado en la parte del análisis, es responsabilidad del equipo web encargado de diseñar la aplicación del servidor.

Diagrama 2

4.3. Diagramas de actividades

4.3.1. Diagrama de actividades Arranque e Inicio de Sesión

En este diagrama de actividades se contempla el arranque y el inicio de sesión. *ESC1*, *ESC5* y *ESC8*.

Diagrama 3

4.3.2. Diagrama de actividades Menú Lateral

El diagrama contempla las posibles acciones del menú lateral. *ESC3*, *ESC7* y *ESC8*.

Diagrama 4

4.3.3. Diagrama de actividades Listado de Partes

En este diagrama de actividades se parte de la pantalla con la lista de órdenes, debido a que es la pantalla principal que se muestra después de haber iniciado sesión exitosamente. Muestra las posibles actividades hasta llegar a mostrar el listado de partes. *ESC8*, *ESC9*, *ESC10* y *ESC11*.

Diagrama 5

4.3.4. Diagrama de actividades Creación de Parte

En este diagrama se parte del punto de salida al que se llega con el diagrama anterior, la pantalla que muestra el listado de partes. Se contemplan las posibles acciones hasta crear un parte. Con la acción “Editar campos del parte” no sólo se incluyen sus parámetros propios si no también los maestros asociados al parte. *ESC11*, *ESC12* y *ESC14*.

Diagrama 6

4.3.5. Diagrama de actividades Visualización/Edición/Cierre de Partes

En este diagrama también se parte de la pantalla con la lista de partes de trabajo. Se muestran las posibles acciones a realizar para visualizar en detalle un parte, o también editarlo y/o cerrarlo. Con la acción “Editar campos del parte” no sólo se incluyen sus parámetros propios si no también los maestros asociados al parte. *ESC4*, *ESC11*, *ESC13* y *ESC14*.

Diagrama 7

4.4. Diagramas de secuencia

Explicaciones, aclaraciones y suposiciones generales:

- Se debe suponer que todos los DTOs se parsean al obtenerlos del servidor y al introducirlos en la base de datos. Se han omitido estas operaciones para aportar mayor claridad.
- Cuando se selecciona una determinada orden o parte, no es necesario llamar a ninguna función para ver su obtener el objeto porque se supone que se debe utilizar el objeto ya existente en la lista.

4.4.1. Diagrama de secuencia Arranque e Inicio de Sesión

Este diagrama recoge la secuencia a realizar para el arranque de la aplicación y el inicio de sesión de un usuario hasta acabar en la lista de órdenes de trabajo. *ESC1*, *ESC5* y *ESC8*.

Explicaciones, aclaraciones y suposiciones:

- Se supone que hay conexión a internet.
- Se supone que las credenciales son correctas.

Diagrama 8

4.4.2. Diagrama de secuencia Refrescar

Este diagrama recoge la secuencia a realizar para refrescar los datos. *ESC7* y *ESC8*.

Explicaciones, aclaraciones y suposiciones:

- Se supone que hay conexión a internet.

Diagrama 9

4.4.1. Diagrama de secuencia Visualizar detalle de orden y maestros

Este diagrama recoge la secuencia a realizar para ver el detalle de una orden de trabajo seleccionada y sus maestros. *ESC8*, *ESC9* y *ESC10*.

Diagrama 10

4.4.2. Diagrama de secuencia Creación de Parte

Este diagrama recoge la secuencia a realizar para crear un parte, así como para editar este parte en su creación añadiendo los campos y los maestros. *ESC8*, *ESC9*, *ESC11*, *ESC12* y *ESC14*.

Diagrama 11

4.4.3. Diagrama de secuencia Cierre de Parte

Este diagrama recoge la secuencia a realizar para editar un parte y cerrarlo, incluyendo la edición de sus campos y maestros. *ESC8*, *ESC9*, *ESC11*, *ESC13* y *ESC14*.

Explicaciones, aclaraciones y suposiciones:

- Se supone que hay conexión a internet.

Diagrama 12

4.5. Pruebas que realizar

4.5.1. Pruebas de interfaz

- Se probará la usabilidad de la interfaz mediante técnicas de evaluación de usuario, como pruebas de tareas, para asegurar que los botones y elementos de la interfaz estén correctamente etiquetados y sean fáciles de entender para los usuarios.
- Se realizarán pruebas de compatibilidad para comprobar que la aplicación se adapte correctamente a diferentes tamaños de pantalla, resoluciones y dispositivos.
- Se realizarán pruebas de usabilidad con usuarios no involucrados en el desarrollo para recoger sus opiniones y sugerencias, se les dará una tarea específica y se observará cómo interactúan con la interfaz, esto ayudará a detectar problemas en la interfaz que los desarrolladores no han notado.

4.5.2. Pruebas de funcionalidad

- Se realizarán pruebas unitarias y de integración para asegurar que la aplicación cumple con todos los requisitos especificados en el documento de requisitos.
- Se probará cada una de las funcionalidades de la aplicación mediante casos de prueba automatizados y manuales para asegurar su correcto funcionamiento.
- Se comprobará que la aplicación maneja correctamente los errores y excepciones mediante pruebas de escenarios de fallo.

4.5.3. Pruebas de integración

- Se realizarán pruebas de integración entre los diferentes componentes y módulos de la aplicación para asegurar que todo funciona correctamente juntos.
- Se comprobará que la aplicación se integra correctamente con otros sistemas y servicios externos si es necesario mediante pruebas de integración.
- Se comprobará que la aplicación maneja correctamente las interacciones entre componentes y servicios externos.

4.5.4. Pruebas de rendimiento

- Se realizarán pruebas de rendimiento para medir el tiempo de respuesta de la aplicación y asegurar que cumpla con los requisitos de velocidad y capacidad.
- Se comprobará que la aplicación no se sobrecarga y se mantiene estable incluso bajo una alta carga de trabajo mediante pruebas de carga y stress.
- Se comprobará que la aplicación maneja correctamente la escalabilidad y el rendimiento en diferentes entornos y configuraciones.

4.5.5. Pruebas de seguridad

- Se realizarán pruebas de seguridad para comprobar que la aplicación está protegida contra posibles ataques y violaciones de seguridad mediante pruebas de penetración y escaneo de vulnerabilidades.
- Se comprobará que la aplicación cumple con los estándares y regulaciones de seguridad aplicables, incluyendo el cumplimiento de normativas como PCI-DSS y HIPAA.
- Se realizarán pruebas de seguridad para verificar la protección de la aplicación contra ataques comunes como inyección SQL, XSS y CSRF.

4.5.6. Pruebas de uso

- Se realizarán pruebas de uso para comprobar que la aplicación es fácil de usar y ofrece una experiencia de usuario satisfactoria para los usuarios mediante pruebas de aceptación y pruebas de usuario final.
- Se recogerán comentarios y sugerencias de los usuarios para mejorar la aplicación.

4.5.7. Pruebas de regresión

- Se realizarán pruebas de regresión para comprobar que las nuevas funcionalidades y cambios no afectan negativamente al funcionamiento de las funcionalidades existentes de la aplicación mediante pruebas automatizadas y manuales.
- Se asegurará de que la aplicación sigue funcionando correctamente después de cualquier cambio o actualización mediante pruebas de regresión.
- Se compararán los resultados de las pruebas de regresión con los resultados de las pruebas originales para detectar cualquier cambio o desviación mediante herramientas de comparación de resultados de pruebas.

5. Desarrollo

Notas previas:

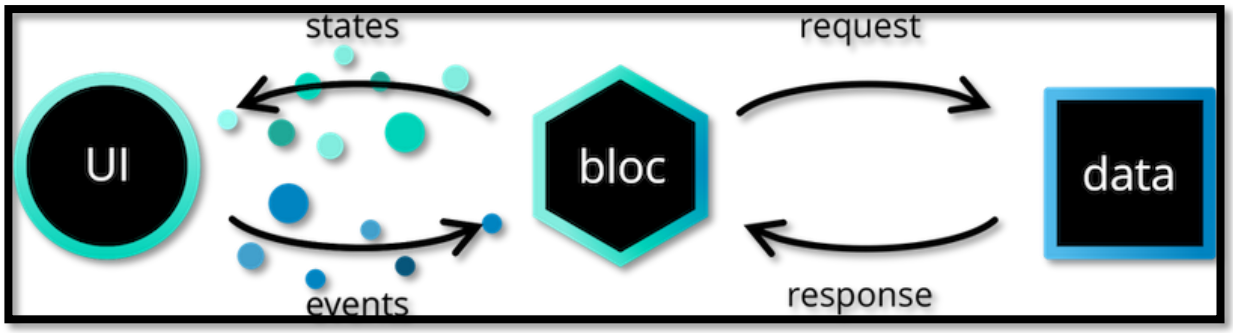
La aplicación se ha desarrollado sin servicios dado que se decidió que esto formaría parte de una segunda fase de desarrollo en conjunto con un equipo encargado de desarrollar el back-end, junto con otras nuevas funcionalidades.

Lo explicado en este apartado de desarrollo es sólo una pequeña muestra del código, dado que explicar todo sería imposible porque superaría el límite de páginas establecido para el documento. Se han elegido partes sencillas para facilitar la comprensión de cualquier lector que no haya trabajado con el framework Flutter. Además de lo explicado aquí, hay muchísimas características que tiene el proyecto. En este *enlace* se puede visualizar el proyecto entero. Se recomienda echar al menos un vistazo, principalmente a la carpeta *lib*, donde se encuentra la mayor parte del código de la aplicación, para valorar el trabajo del autor.

5.1. Estructura

En el *apartado 1.6* se ha mencionado el patrón arquitectónico BLoC. Éste es un patrón como cualquier otro utilizado en la programación: MVC, MVVM, MVP, etc. La diferencia es que BLoC es un patrón propio de Flutter, se creó por y para ser utilizado en este framework. Aunque es cierto que se podría utilizar el patrón sin hacer uso de librerías (escribiendo todo el código a mano) está ampliamente extendido el uso de las librerías *bloc* y *flutter_bloc* entre los programadores de Flutter. Esto es debido a que facilitan la implementación de clases Bloc sin perder recursos, como el tiempo de compilación.

Como es difícil entender el concepto de BLoC sin tener conocimiento previo en este campo, esta imagen del paquete oficial *bloc* ayudará a su entendimiento.



Imaginemos que tenemos una pantalla donde se desea mostrar una lista de personas. En nuestro UI (interfaz) tenemos elementos para esto, como un ListView para agrupar verticalmente y Cards para cada una de las personas en las que se mostrarán algunos de sus datos, como, por ejemplo: nombre, fecha de nacimiento y DNI. Esta lista en un principio está vacía, así que lo que se puede hacer es, que cuando se avance de la pantalla anterior (del login, por ejemplo), al principio del código, se llama mediante un evento al bloc. Dentro del manejador del evento en bloc se hace una *request* a la base de datos o al servidor (lo que tenga la aplicación) para obtener la lista de personas. La base de datos o el servidor envían la lista de personas al bloc. Una vez el bloc tiene esa lista de personas,

hace que cambie el estado de la interfaz, (que actualmente se veía vacía porque la variable con la lista de personas estaba vacía) haciendo que se vuelva a construir el ListView, esta vez con la lista llena. Por lo que el estado de la interfaz cambia y ahora se muestra la lista gráficamente con todas las personas que se han obtenido de la base de datos o del servidor.

Como se podrá suponer, esto no es tan simple como parece, hay que utilizar múltiples elementos para poder realizar este proceso. Primeramente, el bloc se divide en tres elementos: el estado (BlocState), el evento (BlocEvent) y el propio bloc (Bloc). El BlocState es una clase que contiene las variables que pueden cambiar en una vista a lo largo del tiempo, en el ejemplo anterior sería la lista de personas. El BlocEvent es una clase que define las funciones (eventos) que se llaman desde la vista. Y, por último, el Bloc, es una clase que maneja los eventos definidos en el BlocEvent, es la que implementa esas funciones. Es la que se encargaría de llamar a la base de datos o al servidor para obtener la lista de personas, y emitir el nuevo estado (cambiar los valores del BlocState). Todos estos elementos forman parte de la librería *bloc*, aunque, como se ha mencionado anteriormente, se pueden implementar a mano, pero el resultado sería el mismo.

Además de estos elementos, también es conveniente utilizar los de la librería *flutter_bloc*. Todo lo mencionado está muy bien, pero, cómo es posible que cambiando el estado de una variable del BlocState, automáticamente cambie también en la interfaz. Esto se hace mediante varios elementos, en el ejemplo anterior se utilizaría el BlocBuilder. Este elemento sirve para que se vuelva a construir un determinado widget cuando se cambie el estado del BlocState que escucha. En este caso se envolvería el ListView en un BlocBuilder, también se utilizaría una función llamada *buildWhen* para que sólo se volviera a construir el ListView cuando cambiase el valor de la variable que contiene la lista de personas. Esto es debido a que, si hubiera múltiples variables en el BlocState, lo cual es lo más corriente, cada vez que se cambiase una de ellas el widget ListView se volvería a construir, sin ningún cambio, y esto no es nada óptimo. Además del BlocBuilder existen otros elementos utilizados para otras finalidades, como BlocListener, BlocProvider o BlocSelector. Estos elementos se verán más adelante ejemplificados cuando se explique el código.

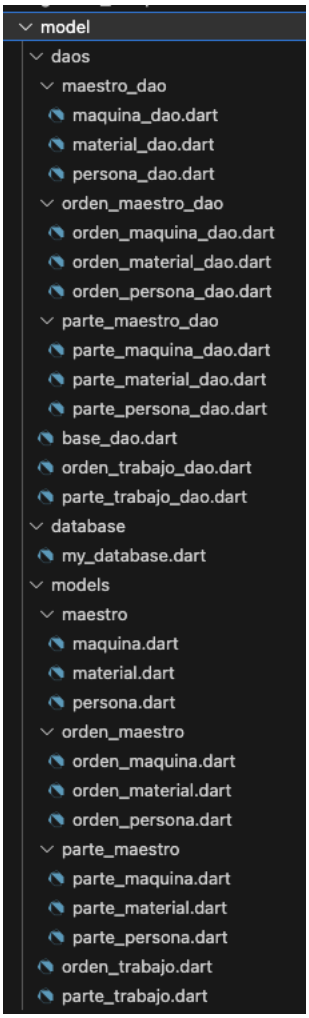
Entendiendo todo lo anterior, ahora se puede explicar la estructura del código. Se divide en tres partes: la vista, los blocs y el modelo. En Flutter los elementos gráficos de la interfaz se denominan Widgets, son todos aquellos elementos como los ListView, Container, Text, Button, TextField, y un largo etcétera. La vista se compone de todos estos widgets, y se divide en dos partes: las páginas y los componentes genéricos. Después están los blocs, explicados en este mismo apartado. Y por último el modelo, básicamente se compone de la base de datos, los DTOs y los DAOs. Tanto la vista, como los blocs, como el modelo se explicarán aisladamente más adelante.

En este *enlace* se puede ver la estructura del proyecto.

5.2. Modelo

El modelo se compone de tres partes: la base de datos, las clases DTO y las clases DAO. La base de datos es el componente donde se almacenan los datos persistentes de la aplicación. Los objetos DTO (*Data Transfer Object*) son estructuras de datos utilizadas para transferir información. Los DAO (*Data Access Object*) son clases que facilitan la manipulación de los objetos DTO en la base de datos.

El modelo entero se encuentra en este *enlace*.



A continuación, se detallarán estos componentes, que en conjunto forman el modelo de la aplicación.

5.2.1. Base de datos

La base de datos se compone por una única clase: la clase *MyDatabase*. Es la que se encarga de gestionar la base de datos utilizando la biblioteca *sqflite*. Su estructura es la siguiente.

```dart
class MyDatabase {
  static final MyDatabase instance = MyDatabase._init();

  Database? _database;

  MyDatabase._init() {
    _initDB('my_database.db').then((db) {
      _database = db;
      _createDB(_database!, 1);
    });
  }

  Future<Database> get database async {
    if (_database != null) return _database!;

    _database = await _initDB('my_database.db');
    return _database!;
  }

  Future<Database> _initDB(String filePath) async {
    final documentsDirectory = await getApplicationDocumentsDirectory();
    final path = join(documentsDirectory.path, filePath);

    return await openDatabase(
      path,
      version: 1,
      onCreate: _createDB,
    );
  }

  Future<void> _createDB(Database db, int version) async {
    await db.execute('''
      CREATE TABLE IF NOT EXISTS ordenesTrabajo (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        fechaInicio TEXT NOT NULL,
        fechaFin TEXT,
```

El constructor privado *MyDatabase._init()* es el punto de entrada para inicializar la base de datos. Este constructor privado (indicado con el guion bajo) llama al método *_initDB* que abre o crea la base de datos y se asigna a la variable privada *_database* una vez que esté disponible. Además, se llama al método *_createDB* para crear las tablas necesarias en caso de que no existan.

El método privado *_initDB* se encarga de abrir o crear la base de datos en una ubicación específica del dispositivo. Utiliza la función *getApplicationDocumentsDirectory* para obtener el directorio de documentos de la aplicación y luego se une al nombre del archivo de la base de datos para formar la ruta completa. Luego, se utiliza la función *openDatabase* para abrir la base de datos en la ruta especificada. Si la base de datos no existe, se ejecuta el método *_createDB* para crear las tablas.

El método privado *_createDB* se encarga de crear las tablas en la base de datos utilizando el objeto *db* pasado como argumento. Cada declaración 'CREATE TABLE IF NOT EXISTS' verifica si la tabla ya existe y, si no, la crea.

Además de los métodos anteriores, existen otros métodos básicos para controlar la base de datos, como *clearDatabase* o *close* que se encargan de limpiar las tablas de la base de datos o cerrarla, respectivamente. En este *enlace* se puede ver la clase completa si se desea.

Para ver la estructura de la base de datos y de la clase entera, pulsar en el *enlace*.

5.2.2. DTOs

Todas las clases DTO tienen el siguiente aspecto.

```dart
class OrdenTrabajo extends ObjectWithMap {
  final DateTime fechaInicio;
  final DateTime? fechaFin;
  final String? tipo;
  final String? observaciones;
  final String? trabajoARealizar;
  final String codigoOrdenCliente;
  final String? instalacion;

  OrdenTrabajo({
    required this.fechaInicio,
    this.fechaFin,
    this.tipo,
    this.observaciones,
    this.trabajoARealizar,
    required this.codigoOrdenCliente,
    this.instalacion,
  }) : super(id: null);

  factory OrdenTrabajo.fromMap(Map<String, dynamic> map) {
    return OrdenTrabajo(
      fechaInicio: DateTime.parse(map['fechaInicio']),
      fechaFin:
          map['fechaFin'] != null ? DateTime.parse(map['fechaFin']) : null,
      tipo: map['tipo'],
      observaciones: map['observaciones'],
      trabajoARealizar: map['trabajoARealizar'],
      codigoOrdenCliente: map['codigoOrdenCliente'],
      instalacion: map['instalacion'],
    )..id = map['id'];
  }

  @override
  Map<String, dynamic> toMap() {
    final map = {
      'fechaInicio': fechaInicio.toIso8601String(),
      'fechaFin': fechaFin?.toIso8601String(),
      'tipo': tipo,
      'observaciones': observaciones,
      'trabajoARealizar': trabajoARealizar,
      'codigoOrdenCliente': codigoOrdenCliente,
      'instalacion': instalacion,
    };

    if (id != null) {
      map['id'] = id.toString();
    }

    return map;
  }
}
```

Para explicar la estructura de un DTO se va a utilizar *OrdenTrabajo* como ejemplo. Primero de todo la clase contiene una serie de variables, que pueden ser *nullables* o no *nullables* (nulas o no nulas), indicado con el símbolo de interrogación de cierre seguidamente del tipo de dato.

Después está el constructor del objeto, se utiliza cada vez que se crea una nueva instancia. Cuando se crea una instancia se han de especificar los parámetros que van precedidos por la palabra clave *required*. Como se puede observar, concuerda con los que no son *nullables*, esto suele ser así excepto en alguna excepción que se verá en este mismo punto en la explicación de la clase *ObjectWithMap*. Las llaves del constructor envolviendo a los parámetros son una forma de hacer que al crear una nueva instancia del objeto utilizando

el constructor, se indiquen los nombres de los parámetros seguidos de dos puntos. Si no se pusieran estas llaves los parámetros serían indicados por posición, y no por nombre.

Por último, existen dos métodos: *fromMap* y *toMap*. Estos métodos se utilizan para parsear la información. El método *fromMap* construye una instancia de un objeto (indicado con la palabra clave *factory*) desde un mapa. El método *toMap* hace lo contrario, convierte una instancia del objeto en un mapa. Son necesarios cuando se quiere introducir un objeto en la base de datos o cuando se desea componer un objeto con los campos provenientes de ella. Actualmente se utilizan para la base de datos, pero también se puede utilizar estos métodos para recibir o enviar datos a través de una API.

Además, todos los DTOs menos los surgidos por una relación entre otras dos entidades, extienden de *ObjectWithMap*. Es una clase que contiene un *id* y los métodos *fromMap* y *toMap*.

En cuanto al uso del *id* en la clase *OrdenTrabajo*, hay varias consideraciones:

1. El *id* es un atributo que actúa como identificador para cada instancia de *OrdenTrabajo* en la base de datos y se hereda como una propiedad de *ObjectWithMap*.
2. Se inicializa con el valor *null*. Esto se debe a que, cuando se crea una nueva instancia de *OrdenTrabajo*, aún no se ha asignado un valor al *id* porque se genera automáticamente cuando se inserta la instancia en la base de datos.
3. En el método *fromMap* de *OrdenTrabajo*, se le asigna el valor del *id* a la instancia de *OrdenTrabajo* creada a partir del mapa. La asignación se realiza mediante el operador de cascada (`..`) seguido de la asignación *id = map['id'].* Esto se hace después de haber creado la instancia, ya que se asume que el valor del *id* se encuentra en el mapa de datos proporcionado. De esta manera, se actualiza la propiedad *id* de la instancia de *OrdenTrabajo* con el valor correspondiente del mapa.
4. En el método *toMap* de *OrdenTrabajo*, si el *id* no es nulo, se agrega al mapa. Sin embargo, si el *id* es *null*, no se incluirá en el mapa, lo que es útil cuando se desea omitir el *id*, en los casos donde se genera automáticamente.

Además de los anteriores, la clase *ParteTrabajo* tiene otros dos métodos, que no tienen el resto de las clases:

```
static ParteTrabajo initial() {
  return ParteTrabajo(
    ordenTrabajoId: -1,
    fechaInicio: DateTime.now(),
    observaciones: '',
    trabajoRealizado: '',
    identificadorDispositivo: '',
  ); // ParteTrabajo
}

ParteTrabajo copyWith({
  int? ordenTrabajoId,
  DateTime? fechaInicio,
  DateTime? fechaFin,
  String? observaciones,
  String? trabajoRealizado,
  String? identificadorDispositivo,
  String? coordenadas,
  int? id,
}) {
  return ParteTrabajo(
    ordenTrabajoId: ordenTrabajoId ?? this.ordenTrabajoId,
    fechaInicio: fechaInicio ?? this.fechaInicio,
    fechaFin: fechaFin ?? this.fechaFin,
    observaciones: observaciones ?? this.observaciones,
    trabajoRealizado: trabajoRealizado ?? this.trabajoRealizado,
    identificadorDispositivo:
        identificadorDispositivo ?? this.identificadorDispositivo,
    coordenadas: coordenadas ?? this.coordenadas,
    id: id ?? this.id,
  );
}
```

El método *initial* devuelve una instancia de *ParteTrabajo*. Es una forma de, al declarar una variable de tipo *ParteTrabajo*, inicializarla con atributos predeterminados. Es por eso por lo que en este caso se inicializa siempre con la fecha actual y con un *id* asociado de orden de -1 (*id* no asociado a ninguna orden real).

Después, a medida que se van editando los campos del parte, se utiliza el método *copyWith*. Este método funciona como un setter compuesto para todos los parámetros del objeto. Se pueden cambiar los valores individualmente o en conjunto. Por lo que, teniendo un parte de trabajo determinado, al editar las observaciones se utiliza este método cambiando los valores de la instancia para, posteriormente, actualizar el objeto entero usando un *update* en la base de datos.

En el *apartado 5.5* se verá el uso conjunto de ambos métodos.

Antes se ha mencionado que las entidades surgidas de la relación de otras entidades no heredan de *ObjectWithMap*. Esto es debido a que esas entidades surgidas utilizan como clave primaria la clave compuesta de las otras dos entidades. Por eso no requieren tener una clave *id* identificatoria. Los métodos *toMap* y *fromMap* se implementan de la misma forma, la única diferencia es que no se sobrescriben y que no tratan con el parámetro *id*.

```dart
class OrdenPersona {
  final int ordenTrabajoId;
  final int personaId;
  final double horas;

  OrdenPersona({
    required this.ordenTrabajoId,
    required this.personaId,
    required this.horas,
  });

  factory OrdenPersona.fromMap(Map<String, dynamic> map) {
    return OrdenPersona(
      ordenTrabajoId: map['ordenTrabajoId'],
      personaId: map['personaId'],
      horas: map['horas'],
    );
  }

  Map<String, dynamic> toMap() {
    return {
      'ordenTrabajoId': ordenTrabajoId,
      'personaId': personaId,
      'horas': horas,
    };
  }
}
```

La clase *ObjectWithMap* tiene el siguiente aspecto:

```dart
class ObjectWithMap {
  int? id;

  ObjectWithMap({required this.id});

  fromMap<T extends ObjectWithMap>(Map<String, dynamic> map) {
    throw UnimplementedError('objectFromMap() must be implemented');
  }

  Map<String, dynamic> toMap() {
    throw UnimplementedError('toMap() must be implemented');
  }
}
```

Aunque en este caso el *id* es un atributo nullable, es obligatorio al construir el objeto. Esto se hace para que las clases que extiendan de *ObjectWithMap* obligatoriamente tengan un *id*, pero que inicialmente sea nulo, hasta que cuando se introduzca la instancia en la base de datos, adquiera un valor.

En el siguiente *enlace* se pueden ver todos los DTOs. Y en este *enlace* la clase *ObjectWithMap* (la clase se encuentra en la parte inferior).

5.2.3. DAOs

Los DAOs se utilizan para manipular mediante las operaciones CRUD (creación, lectura, actualización y eliminación) conjuntos de datos, en este caso, los DTOs.

Únicamente se utilizan en los Bloc, donde se abstraen e introducen los DTOs de la base de datos. No se utilizan en la vista debido a que rompería el patrón arquitectónico BLoC.

En el desarrollo de este código se ha utilizado una clase abstracta llamada *BaseDao* que define e implementa las operaciones CRUD, para que, al extenderla otra clase, no sea necesario implementar estos métodos de nuevo. Esto funciona con todos los métodos que no necesitan devolver un objeto específico, pero en el caso de los métodos *get* (lectura), hay que implementarlos en cada DAO.

```dart
abstract class BaseDao<T extends ObjectWithMap> {
  final String tableName;

  BaseDao({required this.tableName});

  Future<int> create(T obj) async {
    final db = await MyDatabase.instance.database;
    final id = await db.insert(tableName, obj.toMap());
    return id;
  }

  Future<T?> get(int id);

  Future<List<T>> getAll();

  // Devuelve el número de filas afectadas
  Future<int> update(T obj) async {
    final db = await MyDatabase.instance.database;
    return await db.update(
      tableName,
      obj.toMap(),
      where: 'id = ?',
      whereArgs: [obj.id],
    );
  }

  Future<int> delete(int id) async {
    final db = await MyDatabase.instance.database;
    return await db.delete(
      tableName,
      where: 'id = ?',
      whereArgs: [id],
    );
  }

  Future<int> deleteAll() async {
    final db = await MyDatabase.instance.database;
    return await db.delete(tableName);
  }
}
```

Como se puede observar la clase abstracta *BaseDao* toma como parámetro el objeto T, este objeto ha de ser una subclase de *ObjectWithMap*. Esto permite al DAO trabajar con diferentes tipos de objetos que implementan esa clase.

El atributo *tableName* es un String que representa el nombre de la tabla de la base de datos con la que el DAO interactúa.

A continuación, se explica la estructura de una clase DAO utilizando el ejemplo de *ParteTrabajoDao*.

```dart
class ParteTrabajoDao extends BaseDao<ParteTrabajo> {
  static final ParteTrabajoDao _instance = ParteTrabajoDao._internal();
  static ParteTrabajoDao get instance => _instance;

  ParteTrabajoDao._internal() : super(tableName: 'partesTrabajo');

  @override
  Future<ParteTrabajo?> get(int id) async {
    final db = await MyDatabase.instance.database;
    final maps = await db.query(
      tableName,
      where: 'id = ?',
      whereArgs: [id],
    );

    if (maps.isNotEmpty) {
      return ParteTrabajo.fromMap(maps.first);
    } else {
      return null;
    }
  }

  @override
  Future<List<ParteTrabajo>> getAll() async {
    final db = await MyDatabase.instance.database;
    final List<Map<String, dynamic>> maps = await db.query(tableName);

    return List.generate(maps.length, (i) {
      return ParteTrabajo.fromMap(maps[i]);
    }); // List.generate
  }

  Future<List<ParteTrabajo>> getAllPartesDeOrden(
      {required int ordenTrabajoId}) async {
    final db = await MyDatabase.instance.database;
    final List<Map<String, dynamic>> maps = await db.query(
      tableName,
      where: 'ordenTrabajoId = ?',
      whereArgs: [ordenTrabajoId],
    );

    return List.generate(maps.length, (i) {
      return ParteTrabajo.fromMap(maps[i]);
    }); // List.generate
  }

```

ParteTrabajoDao contiene un constructor interno que crea una única instancia de la clase cuando se llama desde el método *get instance* utilizando la variable privada *instance*. Esto es exactamente igual en todos los DAOs.

Se puede observar que los métodos de creación, actualización y eliminación no hace falta sobrescribirlos, como se menciona al principio del apartado. Únicamente es preciso sobrescribir los métodos de lectura (*get*). Además de los dos métodos *get* genéricos que han de estar obligatoriamente en cada DAO: *get* y *getAll*; existen otros métodos específicos en cada DAO. En la imagen anterior se puede ver uno llamado *getAllPartesDeOrden* que devuelve todos los partes de trabajo asociados a un *id* de orden determinado.

Si se desea ver en profundidad, en este *enlace* se encuentra la clase *ParteTrabajoDao*.

Como se puede observar, los métodos realizan consultas utilizando cláusulas de SQL para obtener los datos deseados.

En el caso de las clases DAO que trabajan con DTOs compuestos por dos clases, es decir, las clases Orden-Maestro DAO y Parte-Maestro DAO (para cada uno de los tres maestros), como no extienden de *BaseDao*, al utilizar ésta la clase *ObjectWithMap*, tienen implementadas todas las funciones CRUD. Es decir, no sólo se implementan las de lectura, sino también las de creación, actualización y eliminación.

En el siguiente *enlace* se pueden ver los DAOs completos.

5.3. Blocs

Para la implementación de los blocs se ha utilizado un paquete llamado *freezed* que se utiliza para generar clases inmutables en Dart de manera sencilla. Las clases inmutables son aquellas cuyos objetos no pueden cambiar una vez creados. Son útiles cuando se desea garantizar que los objetos sean inmutables para evitar cambios inesperados en el estado de la aplicación. No importa si se utiliza el paquete *freezed* o no, los Bloc tienen la misma funcionalidad, pero cambia el aspecto de las tres clases, siendo más sencillo de implementar. Como el autor tiene experiencia creando Bloc de ambas maneras, ha optado por utilizar el paquete, dado que facilita el trabajo. En el siguiente *enlace* se muestra la diferencia de un Bloc sin *freezed* y con *freezed*.

En este proyecto existen dos Bloc: el que trabaja con las órdenes de trabajo y el que lo hace con los partes de trabajo. Ambos Bloc no trabajan únicamente con las órdenes y los partes, si no con todo lo relacionado a las mismas. Es decir, el Bloc que controla la lista de órdenes también es el que controla los maestros de una orden.

Como el Bloc relacionado con las órdenes de trabajo es similar al relacionado con los partes de trabajo, pero con menor funcionalidad dada la estructura de la aplicación, se va a utilizar como ejemplo el Bloc que gestiona los partes de trabajo.

Dada la explicación en el *apartado 5.1* (el cual se recomienda ir revisando mientras se ven los ejemplos), a continuación, se va a ver un ejemplo real de un Bloc (de los tres componentes que lo conforman). Notar que: en la primera línea del BlocState y del BlocEvent se puede observar que forman parte de otra clase, dado que el estado y el evento son dos de los tres elementos que lo componen: BlocState, BlocEvent y el propio Bloc. A diferencia de en estas clases, en el propio Bloc aparecen las partes que lo componen, y no que forma parte de otra clase. Estas partes son: el estado, el evento y el Bloc (la clase autogenerada por *freezed)*.

En este *enlace* se pueden ver los dos Bloc implementados en la aplicación.

5.3.1. BlocState

El BlocState es el encargado de definir los parámetros que escucha la vista.

```dart
part of 'listado_partes_bloc.dart';

@freezed
class ListadoPartesState with _$ListadoPartesState {
  const factory ListadoPartesState({
    required bool isLoading,
    required bool isError,
    required List<ParteTrabajo> listPartesTrabajo,
    required ParteTrabajo lastParteCreated,
    required ParteTrabajo? lastParteModified,
    required bool isButtonEnabled,
    required bool isParteClosed,

    /// Personas
    required List<PartePersona> listPartePersonas,
    required List<Persona> listPersonas,

    /// Materiales
    required List<ParteMaterial> listParteMateriales,
    required List<Materiall> listMateriales,

    /// Máquinas
    required List<ParteMaquina> listParteMaquinas,
    required List<Maquina> listMaquinas,
  }) = _ListadoPartesState;

  factory ListadoPartesState.initial() => ListadoPartesState(
        isLoading: false,
        isError: false,
        listPartesTrabajo: [],
        lastParteCreated: ParteTrabajo.initial(),
        lastParteModified: null,
        isButtonEnabled: false,
        isParteClosed: false,
        listPartePersonas: [],
        listPersonas: [],
        listParteMateriales: [],
        listMateriales: [],
        listParteMaquinas: [],
        listMaquinas: [],
      );
}
```

Como se puede observar existen dos constructores, el primero sirve para definir las variables que va a contener el BlocState (en este caso llamado *ListadoPartesState*), el segundo es otro constructor adicional que llama al primero asignando un valor a todas las variables, es el utilizado desde fuera.

Todos estos parámetros que contiene el BlocState son parámetros que pueden ser cambiados de valor al ser manipulados en un evento.

- Por ejemplo, el parámetro *isLoading* se establece a verdadero cuando se está realizando algún evento donde el usuario debe esperar (como mientras carga la lista de partes de trabajo). Después, en una pantalla se escucha este parámetro mediante un BlocListener, para que cuando se detecte que está en verdadero, se muestre un widget circular en mitad de la pantalla para indicar que está cargando.

Por ejemplo, la lista de partes se inicializaría vacía (como el resto de los parámetros), y cuando el usuario se dirigiese a la pantalla que contiene la lista de partes, un evento cambiaría el valor de la variable *listPartesTrabajo* en el BlocState, y después en la pantalla se utilizaría un BlocBuilder para escuchar la variable y volver a pintar la lista llena.

5.3.2. BlocEvent

El BlocEvent es la clase que define las funciones que se van a utilizar desde la vista. Estas funciones se llaman eventos. En el BlocEvent únicamente se definen, no se implementan. La implementación se realiza en el Bloc, utilizando un controlador para cada evento.

```dart
part of 'listado_partes_bloc.dart';

@freezed
class ListadoPartesEvent with _$ListadoPartesEvent {
  const factory ListadoPartesEvent.onLoadPartes() = OnLoadPartes;

  const factory ListadoPartesEvent.onLoadPartesDeOrden(
      {required int ordenTrabajoId}) = OnLoadPartesDeOrden;

  const factory ListadoPartesEvent.onSearch(
      {required int ordenTrabajoId, required String search}) = OnSearch;

  const factory ListadoPartesEvent.onCreateParte(
      {required ParteTrabajo parteTrabajo}) = OnCreateParte;

  const factory ListadoPartesEvent.onUpdateParte(
      {required ParteTrabajo parteTrabajo}) = OnUpdateParte;

  const factory ListadoPartesEvent.onChangeButtonState({
    required bool buttonState,
  }) = OnChangeButtonState;

  /// Personas
  const factory ListadoPartesEvent.onLoadPersonasDeParte(
      {required int parteTrabajoId}) = OnLoadPersonasDeParte;

  const factory ListadoPartesEvent.onLoadPersonasDePartePersona() =
      OnLoadPersonasDePartePersona;

  const factory ListadoPartesEvent.onSearchPersona(
      {required int parteTrabajoId, required String search}) = OnSearchPersona;

  const factory ListadoPartesEvent.onUpdateHoursPartePersona(
      {required int parteTrabajoId,
      required int personaId,
      String? hours,
      String? mins}) = OnUpdateHoursPartePersona;
```

Para definir una función primero se pone el nombre del BlocEvent, en este caso *ListadoPartesEvent*, y después de un punto, el nombre de la función. Después se iguala al nombre que se desea dar al evento. Así después, cuando se quiera definir el funcionamiento utilizando un controlador del evento en el Bloc, se puede hacer mediante el nombre. Al llamar a un evento desde la vista, se puede también hacer con el nombre del evento, pero es preferible hacerlo con el nombre del BlocEvent y de la función. Así, si existiesen dos Blocs que tuvieran un evento con el mismo nombre (como *OnSearch*), al llamarlo desde la vista no habría confusión, porque el nombre del BlocEvent sería distinto.

Cuando se especifica un parámetro en la función, después desde el controlador del evento en el Bloc se puede acceder a ese parámetro. Esto se ve en el siguiente *apartado*.

En este siguiente *enlace* está la clase entera.

5.3.3. Bloc

El Bloc es el encargado de controlar los eventos a los que llama la vista y también de cambiar el estado de los parámetros que escucha la vista.

La vista escucha el BlocState. Dadas determinadas acciones, como pulsar un botón o inicializar una pantalla, la vista llama a un evento definido en el BlocEvent. El Bloc, desde donde se controlan los eventos llamados por la vista, realiza las operaciones pertinentes, como una consulta a la base de datos, y cambia el estado del BlocState. Al cambiar el estado del BlocState y la vista estar escuchando ese BlocState, la vista cambia. Es importante aclarar que la vista no tiene por qué cambiar siempre, en el ejemplo que hemos visto anteriormente donde se utilizaba la variable *isLoading* del BlocState, la vista no se reconstruye, sólo llama a una función. Esto es porque en la vista se usa un BlocListener y no un BlocBuilder. En el *apartado 5.4* se ven ambos casos.

Los controladores del Bloc donde se implementa la funcionalidad del evento son funciones llamadas *on*, es por eso por lo que a todos los eventos se les suele llamar 'On…', aunque esto es preferencia del programador que desarrolle el código.

```dart
part 'listado_partes_event.dart';
part 'listado_partes_state.dart';
part 'listado_partes_bloc.freezed.dart';

class ListadoPartesBloc extends Bloc<ListadoPartesEvent, ListadoPartesState> {
  final ParteTrabajoDao _parteTrabajoDao = ParteTrabajoDao.instance;

  final PartePersonaDao _partePersonaDao = PartePersonaDao.instance;
  final PersonaDao _personaDao = PersonaDao.instance;

  final ParteMaterialDao _parteMaterialDao = ParteMaterialDao.instance;
  final MaterialDao _materialDao = MaterialDao.instance;

  final ParteMaquinaDao _parteMaquinaDao = ParteMaquinaDao.instance;
  final MaquinaDao _maquinaDao = MaquinaDao.instance;

  ListadoPartesBloc() : super(ListadoPartesState.initial()) {
    on<ListadoPartesEvent>((event, emit) {});

    on<OnLoadPartesDeOrden>((event, emit) async {
      emit(state.copyWith(isLoading: true, isParteClosed: false));

      List<ParteTrabajo> partesTrabajo = await _parteTrabajoDao
          .getAllPartesDeOrden(ordenTrabajoId: event.ordenTrabajoId);

      emit(state.copyWith(isLoading: false, listPartesTrabajo: partesTrabajo));
    });
```

Teniendo en cuenta lo anterior, aquí se puede ver qué ocurre cuando la vista llama al evento *OnLoadPartes* para obtener la lista de partes y mostrarla en una pantalla. Primero se emite un estado haciendo uso de la función *emit*, la variable *isLoading* se pone a *true*. Después, haciendo uso del DAO de los partes de trabajo, se guarda en una variable la lista de todos los partes asociados a una determinada orden. Como se puede observar, al llamar a la función del DAO que obtiene todos los partes de una orden, se le pasa como parámetro el *id* de la orden de la que se desean obtener los partes. Esto se logra indicando el parámetro *ordenTrabajoId* en la definición del evento en el BlocEvent. Cuando la vista llama a este evento, le pasa el *id* de la orden (debido a que, al venir anteriormente de una orden, tiene el objeto), y después se usa en el controlador. Después de haber obtenido todos los partes, se vuelve a emitir otro estado que cambia el valor del parámetro de *isLoading* a *false* y cambia el parámetro *listPartesTrabajo* con la lista llena de todos los partes asociados a la orden del evento. En este *enlace* está la clase entera.

5.4. Vista

La vista se compone de dos partes: las páginas y los componentes genéricos. Las páginas son las pantallas que el usuario ve navegando por la interfaz. Los componentes genéricos son elementos gráficos reutilizados en las pantallas.

Tanto las pantallas, como los componentes genéricos, son formadas mediante widgets. Los widgets son cualquier elemento visualizable en Flutter. Es por eso por lo que desde un botón hasta una pantalla entera están hechos mediante la combinación de uno o varios widgets.

Es recomendable ir visualizando las pantallas de la aplicación, situadas en el *anexo 1*, al mismo tiempo que se mencionan.

5.4.1. Páginas

Principalmente hay tres tipos de pantalla en este proyecto: las que agrupan una lista de órdenes y partes, las que muestran el detalle de una orden y parte, y las que muestran el detalle de los maestros en una orden y en un parte. Además de estas pantallas hay otras como el login. En el siguiente *enlace* se pueden ver todas.

Dado que para las órdenes y para los partes los tres tipos de pantalla están construidos de forma similar, a excepción de las que muestran el detalle de los maestros, las cuales son más complejas las de los partes, se ha decidido utilizar como ejemplo para los tres tipos de pantalla las relacionadas con los partes, y no con las relacionadas con las órdenes.

Primera pantalla: lista de partes dada una determinada orden.

Dada que la vista es muy larga para insertar una captura, se recomienda verla en el siguiente *enlace*.

Como se observa en el código, la página de la lista de partes se compone de tres clases. La primera clase es un widget estático, es decir, que no puede variar o volver a construirse a lo largo del tiempo. Esta clase obtiene el parámetro *ordenTrabajo* de la pantalla anterior. Luego obtiene el Bloc de la lista de partes, creado en el *main* de la aplicación, mediante el uso de un BlocProvider, especificando el Bloc que se quiere obtener y pasándole el contexto. Después, una vez ha recuperado el Bloc, llama a un evento del Bloc para obtener la lista de partes de esa orden pasándole el id de la orden como argumento. Por último, devuelve la segunda clase. La segunda clase es un widget con estado. El método *createState* es obligatorio y devuelve una instancia de *_PartesTrabajoViewState*, que es una clase que extiende *State* y maneja el estado interno de *PartesTrabajoView*. La tercera clase *_PartesTrabajoViewState* es la implementación del estado interno para *PartesTrabajoView*. En la tercera clase es donde se encuentra el contenido que ve en la pantalla el usuario.

La columna de esta pantalla es un Scaffold customizado, básicamente un Scaffold (andamio en inglés) es un widget que permite crear distintos elementos en la pantalla como una AppBar (barra superior con el título y el id de la orden), BottomBar (barra inferior, en este caso no hay), FloatingActionButton (botón flotante, en este caso para crear un parte), etc; además del cuerpo. El cuerpo del Scaffold (*body*) es un BlocConsumer. El BlocConsumer se compone de dos partes, la que escucha y la que construye (*listener* y

builder). El *listener* está escuchando cambios en las variables cargando y de la lista de partes. Como se puede observar cuando la variable *isLoading* está en verdadero llama a una función que muestra un círculo indicando que está cargando en mitad de la pantalla. El *builder* no escucha ninguna variable en este caso, porque se desea que se vuelva a construir cada vez que cambie el estado del Bloc. El *builder* devuelve una vista deslizable que contiene un cuadro de búsqueda en la parte superior, para buscar el parte que se desee. Cada vez que se escribe un número en este cuadro de búsqueda, se llama al Bloc del listado de partes para que actualice la lista de partes, mostrando sólo los que encajan con la búsqueda. Como ha cambiado una de las variables del BlocState y no se ha especificado con qué variables se tiene que volver a construir, se construye automáticamente. Así que el *listener* deja de mostrar el cargando porque ya está en false y el *builder* vuelve a construir la pantalla con la nueva lista de partes que cumplen con la búsqueda. Esta lista de partes se muestra con un *ListView.buidler*, este widget es como un ListView pero autogenera el número de elementos que indique el valor del parámetro *itemCount*. Cada parte se muestra en una card, donde al clickar en ella redirige al detalle de ese parte.

Segunda pantalla: detalle de un parte.

Dada que la vista es muy larga para insertar una captura, se recomienda verla en el siguiente *enlace*.

Como ya se ha explicado en la primera pantalla la estructura que tiene una página, ahora se omite esta información, y se explica únicamente el contenido.

En la función *initState* (la cual se ejecuta al inicializar el widget una sola vez) se inicializa la variable local *_parteTrabajo* con el parte de trabajo recibido de la pantalla anterior. Se puede observar que esta vez no se ha hecho con una ruta. En la pantalla anterior se ha utilizado una ruta y en esta se ha pasado como parámetro a la clase para que el lector vea las dos posibilidades, ambas igual de eficaces.

Este widget se compone encima de todo por un BlocListener. Con el parámetro *listenWhen* se indica que va a escuchar a la variable *isParteClosed* del BlocState de la lista de partes. Cuando el *listener* detecta que el parte está cerrado, viendo que la variable *isParteClosed* está en verdadero, se llama a una función que muestra un diálogo notificando al usuario que se ha cerrado el parte correctamente.

Esta pantalla también es una pantalla compuesta por un widget deslizable. Aparecen todos los parámetros de un parte como textos, y las observaciones y el trabajo realizado como campos de texto editables. Después de esto se encuentran los tres botones que llevan a la pantalla donde se encuentran los maestros del parte. Por último, está el botón de cerrar parte.

En el caso de esta pantalla hay tres llamadas a eventos, cuando se cambia el contenido de las observaciones, cuando se cambia el contenido del trabajo realizado y cuando se pulsa el botón cerrar parte.

Cuando se edita un campo, se utiliza el método *copyWith* de *ParteTrabajo*, mencionado en el *apartado 5.2.2*, para *setear* el parámetro pertinente en la instancia del parte. Una vez se ha hecho esto, se llama mediante un evento al Bloc. En el controlador de ese evento, utilizando el DAO de parte de trabajo, se hace un *update* en la base de datos.

Después se obtiene el parte de la base de datos utilizando el id del parte, para asegurarse de que se tiene el elemento sincronizado. Más tarde, todo dentro del evento del Bloc, se comprueba si tiene fecha de fin, en este caso, al editar un campo, no ha de tener. Sin embargo, cuando se pulsa el botón de cerrar parte, ocurre exactamente lo mismo, excepto con la diferencia de que se le añade la fecha actual como fecha fin usando el método *copyWith*. Entonces en la lógica del evento sí entra en la condición donde tiene una fecha fin y cambia la variable *isParteClosed* del BlocState. El *listener* antes mencionado escucha la variable y muestra el diálogo.

Tercera pantalla: personal de un parte.

Dada que la vista es muy larga para insertar una captura, se recomienda verla en el siguiente *enlace*.

Como ya se ha explicado en la primera pantalla la estructura que tiene una página, ahora se omite esta información, y se explica únicamente el contenido.

Se ha utilizado como ejemplo el personal, pero es muy similar la de los tres maestros.

Esta pantalla es la más compleja de las mencionadas, la vista tiene varias clases (aunque se podría hacer todo en una, pero en estos casos donde hay partes muy diferenciadas con muchas líneas de código, lo recomendable es modularizar), también tiene funciones y múltiples llamadas a eventos; además de cierta complejidad en la lógica. Además de esto, la lógica de los eventos también es larga y compleja. Por estos motivos la siguiente pantalla se comenta a grandes rasgos (en mayor medida que el resto del código), sin explicaciones exhaustivas que puedan entorpecer la comprensión del lector.

Lo primero que ocurre en esta pantalla es que se llama a un evento del Bloc, llamado *OnLoadPersonasDeParte*. En este evento se obtienen todos los objetos *PartePersona*. El DTO *PartePersona* está compuesto por las claves del parte y de la persona, junto con la cantidad de horas, pero no con la descripción de la persona. Es por esto por lo que después se obtienen todas las personas de la base de datos. Luego se almacenan en una lista todos los *ids* de persona de la lista de tipo *PartePersona*. Se hace lo mismo con los *ids* de la lista de tipo Persona. Se ordenan ambas listas. Se comparan ambas listas. Si en la lista de las personas de un parte falta algún id de la lista de personas, se crea un nuevo objeto *PartePersona* y se introduce en la base de datos. Esto se hace utilizando el id del parte del evento y el id de la persona faltante. Se repite todas las veces necesarias. Esto se hace básicamente para que la primera vez que se entre a la vista de los maestros de un parte, se creen todas las combinaciones posibles, y para que cuando se entre la segunda vez, únicamente se obtengan y no se creen de nuevo. Cuando se tienen ambas listas de objetos *PartePersona* y de objetos Persona se emiten en un estado cambiando las dos variables.

Después de esto en la vista, cuando se escucha el nuevo estado emitido se pasan al siguiente widget y se construye. Además del widget que muestra la lista de personas para añadir al parte, hay un buscar en la parte superior de la pantalla, que funciona como el de la primera pantalla.

En el segundo widget se crea una lista de mapas donde se agrupan los parámetros de las dos listas:

```
// Se crea una lista combinando la descripción de Persona y las horas de PartePersona.
// Se formatea las horas para pasar de double a String --> 2.5 == '2h 30m'
_dataList = widget.partePersonas
    .map((partePersona) => {
          'parteTrabajoId': partePersona.parteTrabajoId.toString(),
          'personaId': partePersona.personaId.toString(),
          'descripcion': widget.personas.isNotEmpty
              ? widget.personas
                  .firstWhere((p) => p.id == partePersona.personaId,
                      orElse: () => Persona(descripcion: 'Sin descripción'))
                  .descripcion
              : 'Sin descripción',
          'horas': partePersona.horas.toString(),
        })
    .toList();
```

Después se ordena esta lista. El resultado de esto es una lista ordenada por horas con los *ids* de persona y del parte, con la descripción por parte de persona y con las horas por parte de la entidad surgida de la relación entre una persona y un parte (*PartePersona*). Hay que considerar que más tarde cada vez que se actualiza la cantidad de horas que ha trabajado una persona en un parte, las listas cambian de valor, y se vuelve a construir el widget y la lista de mapas. Por eso, aunque al principio todos los elementos de la lista de mapas tengan un valor de horas igual a 0, después no es así, por eso se ordena para que el usuario pueda ver las personas con mayor número de horas primero. También es importante considerar que, aunque para la lógica no fuese necesario traer ambas listas a la vista mediante un estado, sí lo es para el usuario, dado que necesita leer la descripción del personal.

El segundo widget se compone de dos botones en la parte superior y de un ListView en la parte inferior. Uno de los botones sirve para deshacer los cambios no guardados al editar las horas del personal. El otro botón sirve para guardar los cambios. El primer botón limpia los controladores de las horas y los minutos de todas las personas, además llama a un evento que pone a false el estado de los botones, para que aparezcan desactivados al haber eliminado los cambios. El segundo botón llama a un evento que actualiza la lista de personas en un parte cambiando las horas, además también llama al evento que desactiva los botones. El ListView está formado por todas las personas. Para cada elemento se muestra la descripción de la persona y dos campos de texto, uno para escribir las horas y otro para los minutos. Cabe mencionar que detrás de esto existe mucha lógica, como por ejemplo adaptaciones en el formato de las horas, dado que el usuario las escribe por separado y en la base de datos entran como un double, o validaciones de formato para no poder poner más de 59 minutos, y un largo etcétera.

5.4.2. Componentes genéricos

Se puede ver el código de cada componente en este *enlace*, acompañado de las ilustraciones de la interfaz, ubicadas en el *anexo 1*.

5.5. Otros

Además de todo lo mencionado anteriormente, la aplicación cuenta con más pantallas, blocs y componentes. Además, cuenta con un *main*, *scripts* y *elementos* para la traducción de idiomas, *funciones* utilizadas para los formatos, *splash screen* personalizada, un *login*, *estilos y fuentes* personalizados, *extensiones*, *recursos*, *paquetes*, *listas de literales*, etc. Es una aplicación muy completa de más de 15.000 líneas de código. Se recomienda echar un vistazo general a las diferentes carpetas y archivos del proyecto, dado que explicar todo esto en este documento, sería inviable.

6. Conclusiones

El Trabajo de Fin de Grado ha sido una experiencia enriquecedora que ha abarcado todas las etapas de un proyecto, desde un estudio inicial, pasando por el desarrollo de la aplicación, y hasta la documentación del proceso.

Durante su desarrollo, se ha puesto especial énfasis en la arquitectura y la claridad del código fuente, lo que lo convierte en una sólida base para futuros desarrollos tanto para mí como para otros interesados.

En resumen, este proyecto ha logrado implementar con éxito una aplicación móvil que optimiza el proceso de creación y gestión de partes de trabajo en el ámbito de los proyectos de construcción. Su arquitectura estructurada y su código fuente claro proporcionan una base sólida para futuros desarrollos y la convierten en una herramienta valiosa para empresas privadas que buscan soluciones eficientes de gestión de trabajo.

7. Referencias

- IEEE Recommended Practice for Software Requirements Specifications," in *IEEE Std 830-1998*, vol., no., pp.1-40, 20 Oct. 1998
- ISO/IEC/IEEE International Standard - Systems and software engineering -- Life cycle processes -- Requirements engineering," in *ISO/IEC/IEEE 29148:2018(E)*, vol., no., pp.1-104, 30 Nov. 2018
- https://docs.flutter.dev/
- https://bloclibrary.dev/
- https://learn.microsoft.com/es-es/appcenter/
- https://git-fork.com/
- https://docs.github.com/en

8. Anexo 1 – Ilustraciones de la interfaz

Ilustración 1: Login

Ilustración 2: Login con teclado

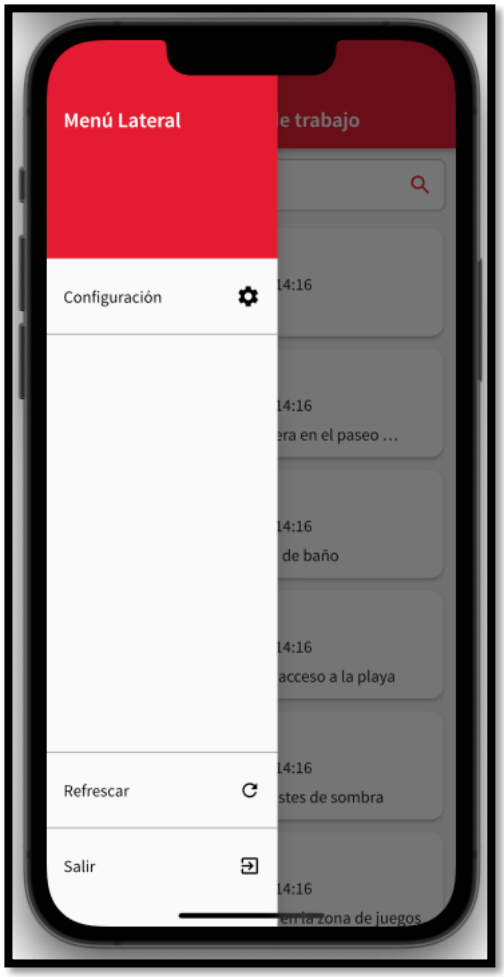


Ilustración 3: Menú lateral

Ilustración 4: Lista de órdenes de trabajo

Ilustración 5: Lista de órdenes de trabajo - Búsqueda de orden

Ilustración 6: Detalle de orden de trabajo

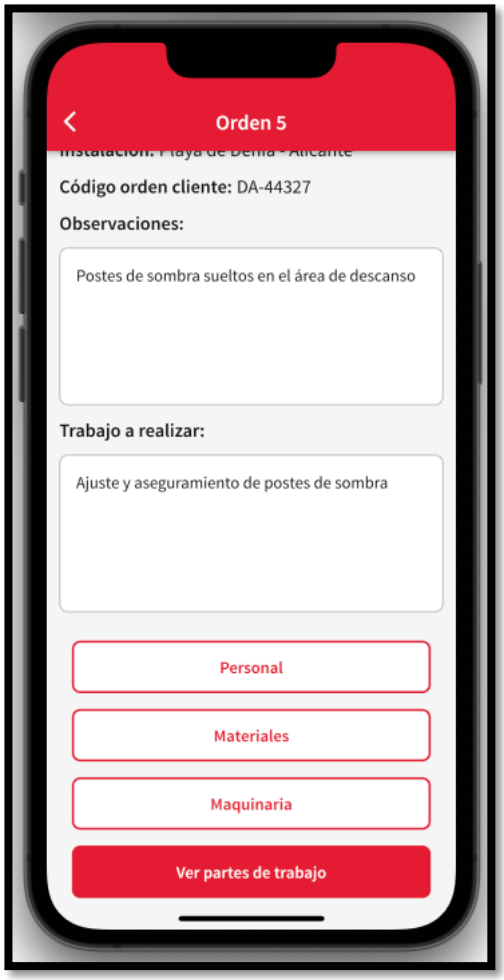


Ilustración 7: Detalle de orden de trabajo - Parte inferior

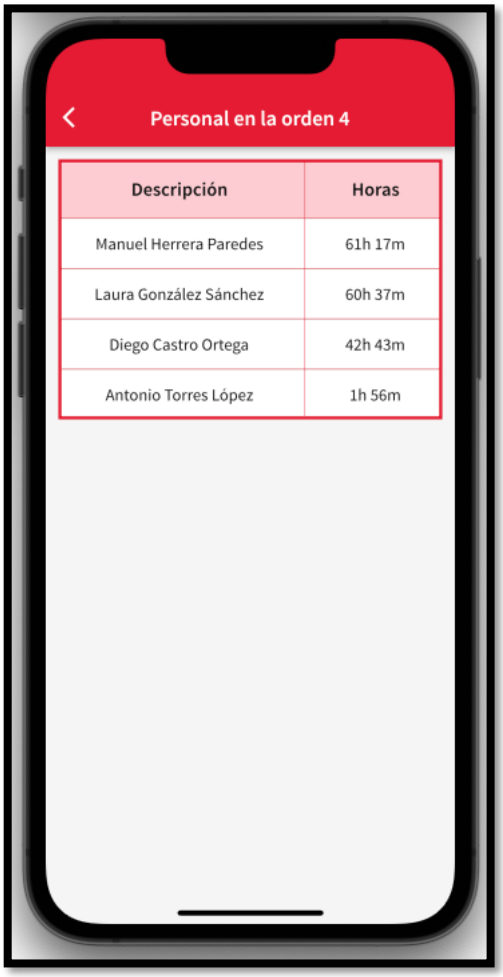


Ilustración 8: Detalle de orden de trabajo - Detalle del personal

Ilustración 9: Detalle de orden de trabajo - Detalle de los materiales

Ilustración 10: Detalle de orden de trabajo - Detalle de la maquinaria

Ilustración 11: Lista de partes de trabajo - Vacía

Ilustración 12: Lista de partes de trabajo - Llena

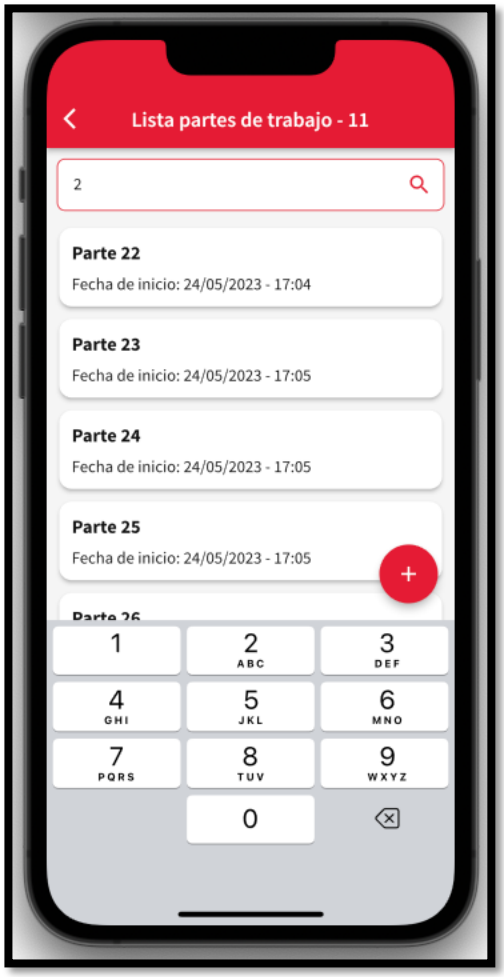


Ilustración 13: Lista de partes de trabajo - Búsqueda de parte

Ilustración 14: Creación de parte de trabajo

Ilustración 15: Edición de parte de trabajo

Ilustración 16: Detalle personal de parte de trabajo – Por defecto

Ilustración 17: Detalle personal de parte de trabajo - Con cambios

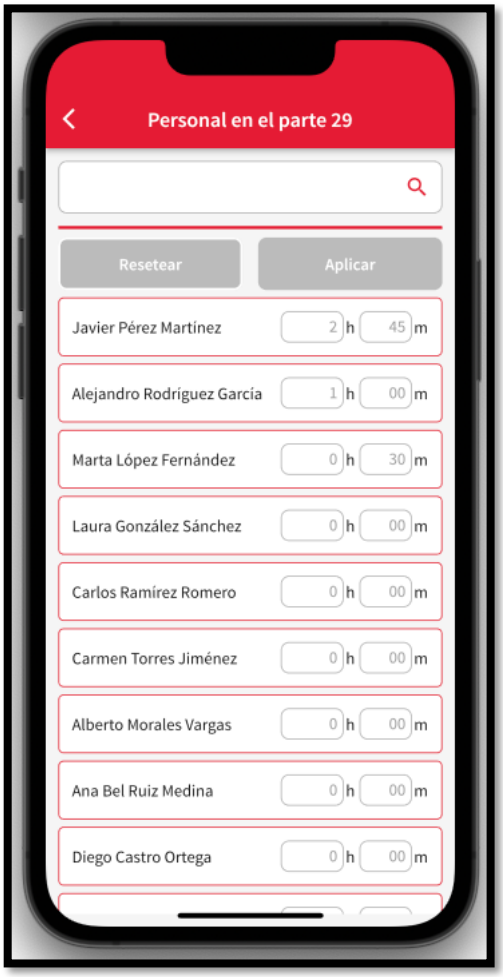


Ilustración 18: Detalle personal de parte de trabajo - Cambios aplicados

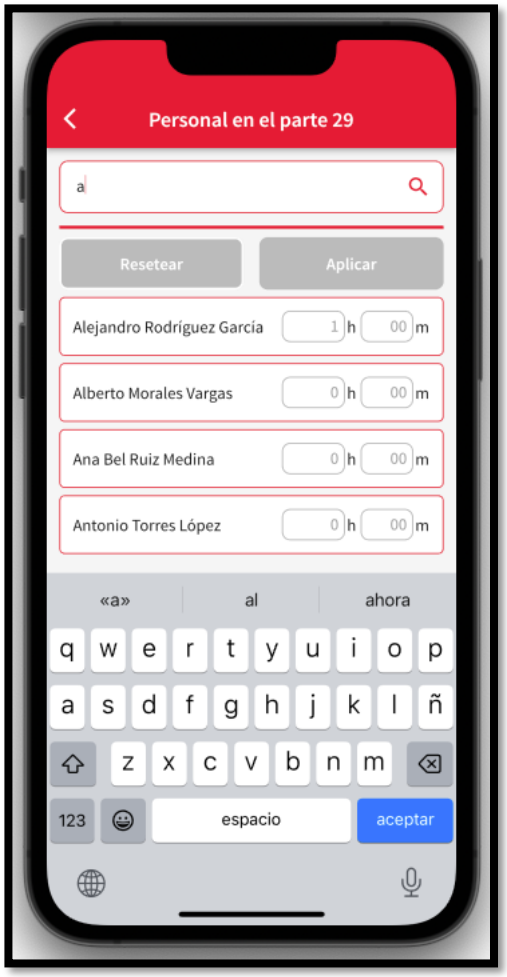


Ilustración 19: Detalle personal de parte de trabajo - Búsqueda de persona

Ilustración 20: Detalle materiales de parte de trabajo - Por defecto

Ilustración 21: Detalle materiales de parte de trabajo - Con cambios

Ilustración 22: Detalle materiales de parte de trabajo – Cambios aplicados

Ilustración 23: Detalle materiales de parte de trabajo - Búsqueda de material

Ilustración 24: Detalle maquinaria de parte de trabajo - Por defecto

Ilustración 25: Detalle maquinaria de parte de trabajo - Con cambios

Ilustración 26: Detalle maquinaria de parte de trabajo - Cambios aplicados

Ilustración 27: Detalle maquinaria de parte de trabajo - Búsqueda de máquina

9. Anexo 2 – Diagramas

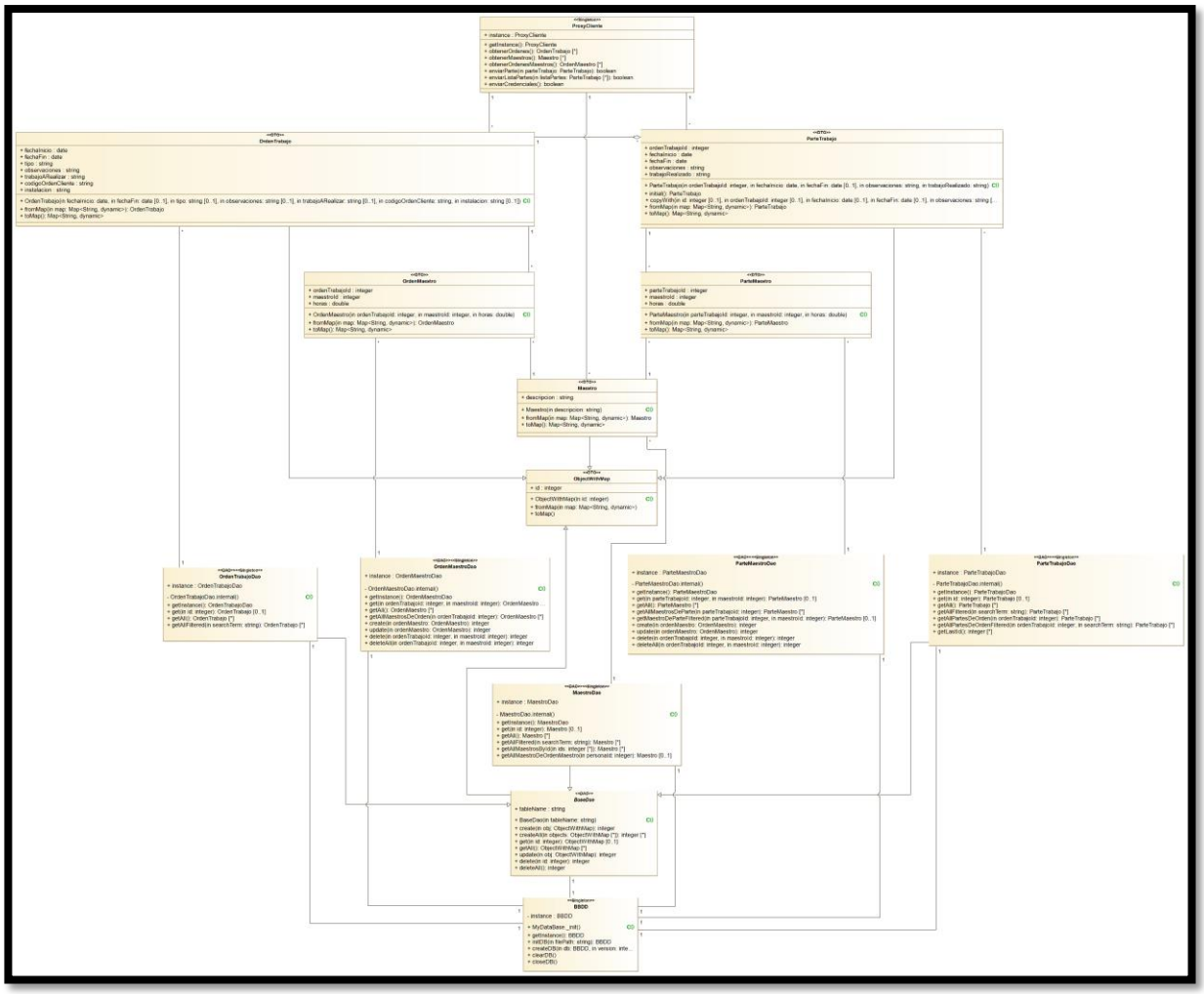


Diagrama 1: Diagrama Clases Cliente

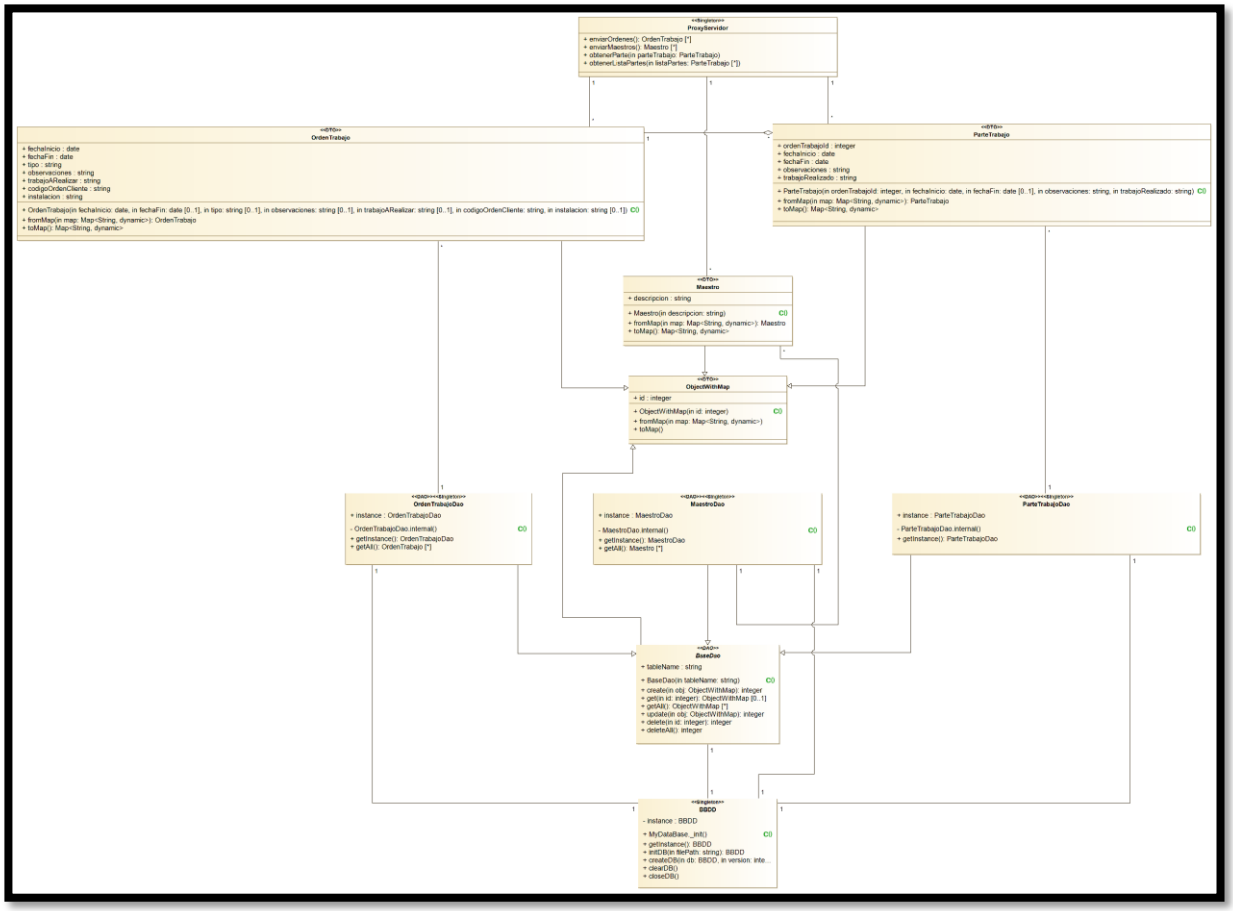


Diagrama 2: Diagrama Clases Servidor

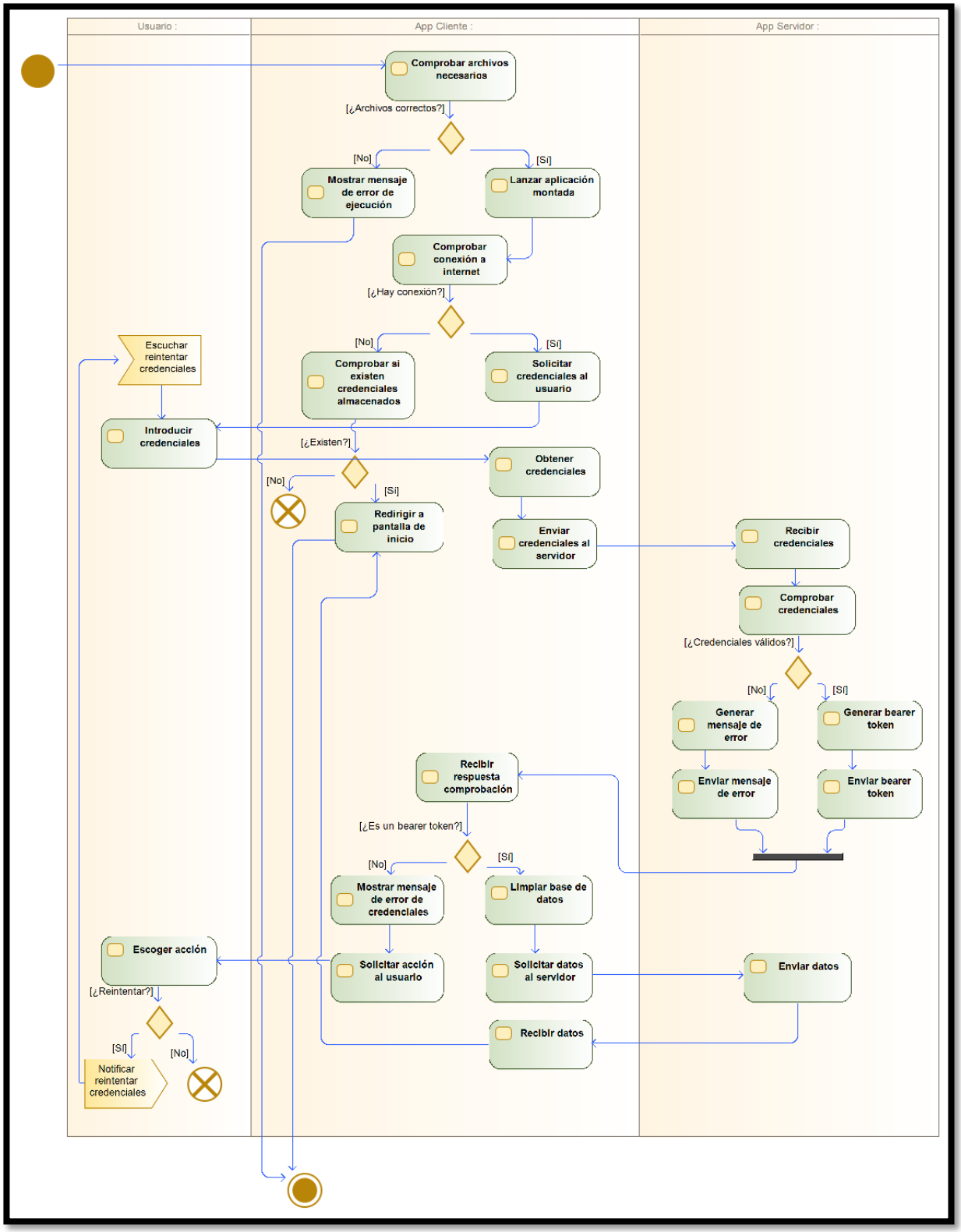


Diagrama 3: Diagrama Actividades Inicio de Sesión

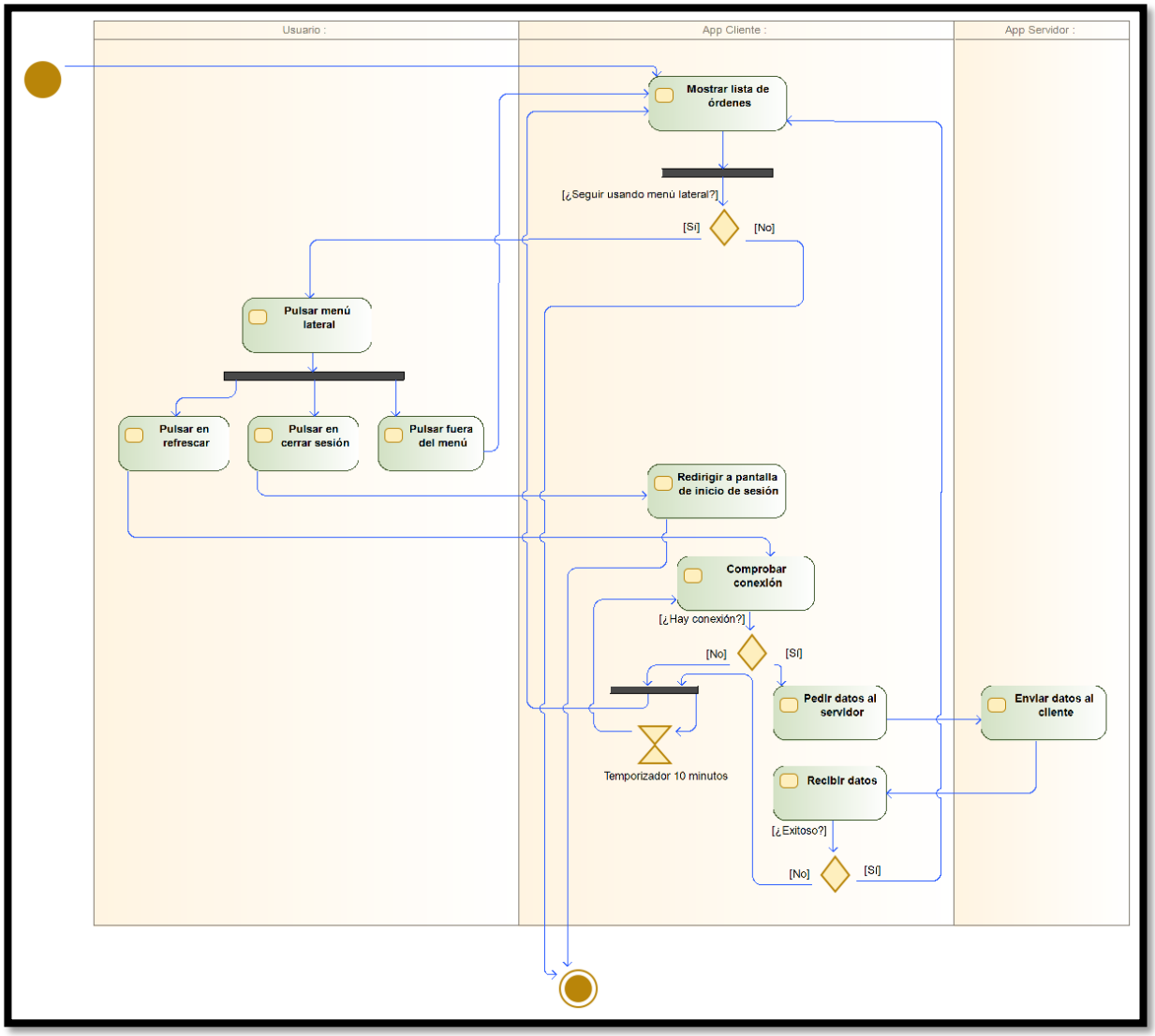


Diagrama 4: Diagrama Actividades Menú Lateral

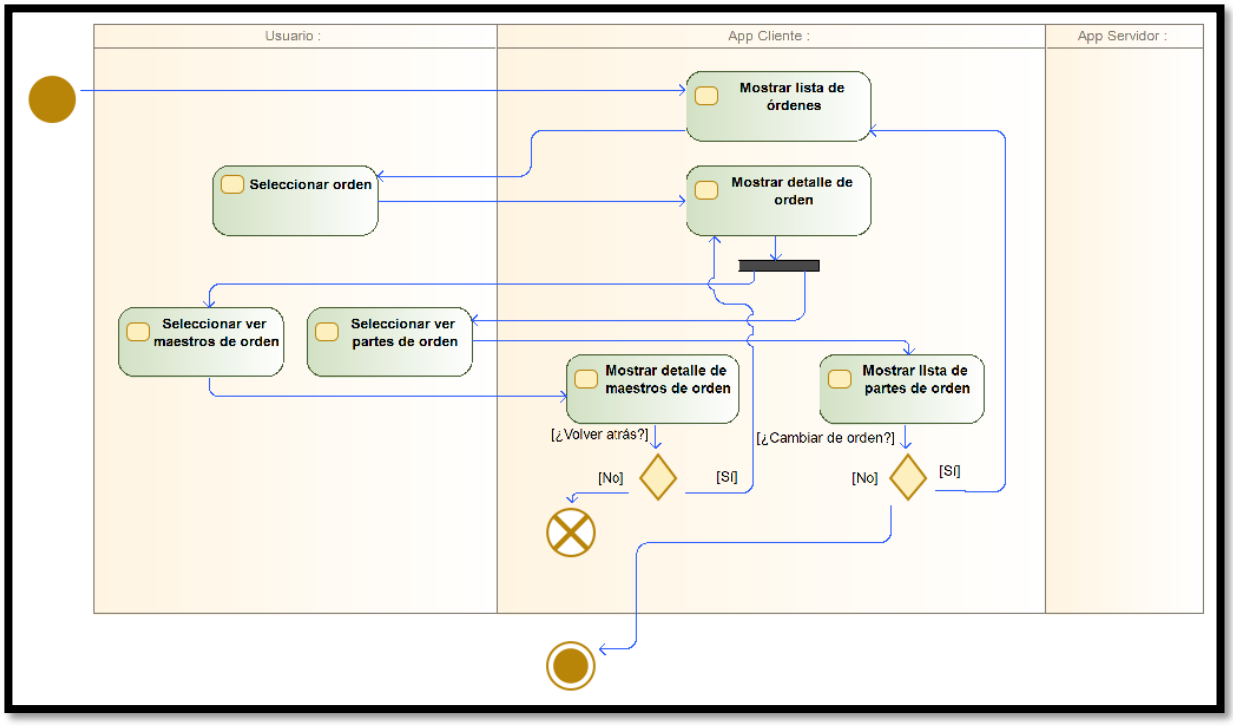


Diagrama 5: Diagrama Actividades Visualizar Lista Partes

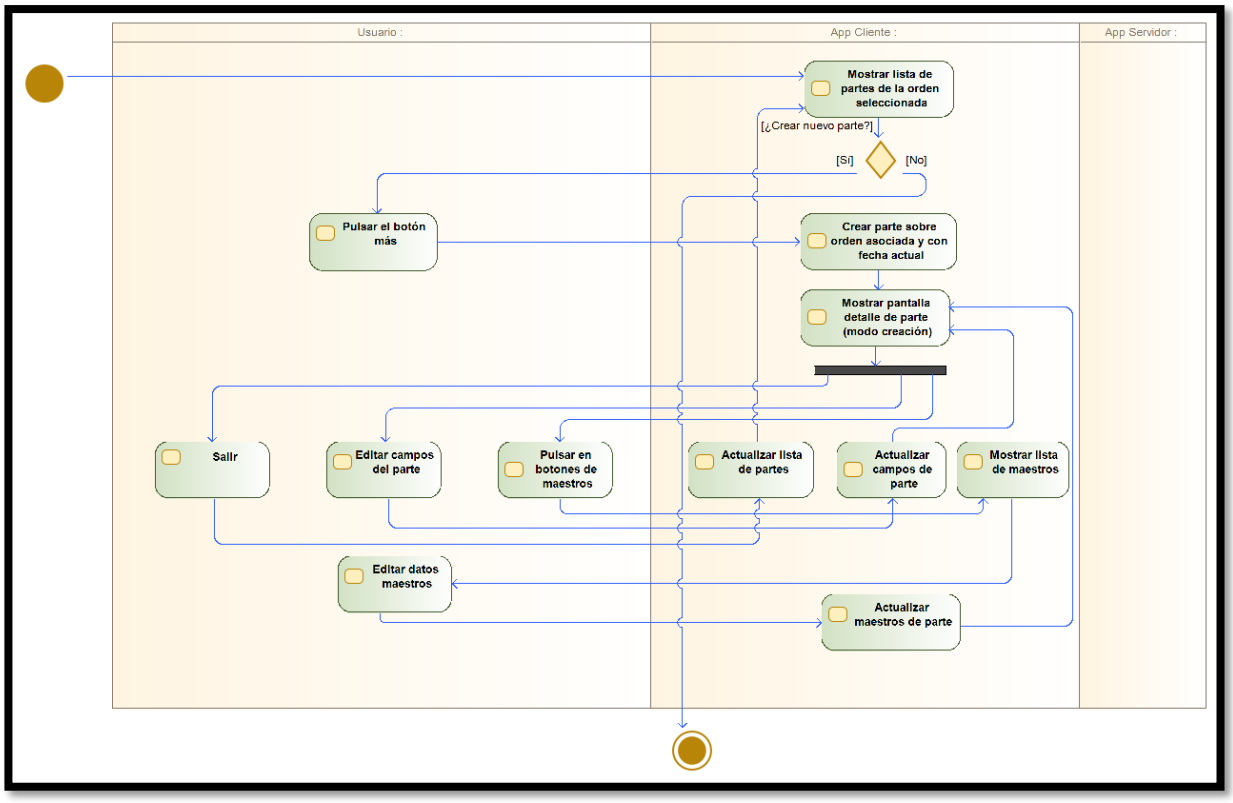


Diagrama 6: Diagrama Actividades Creación de Parte

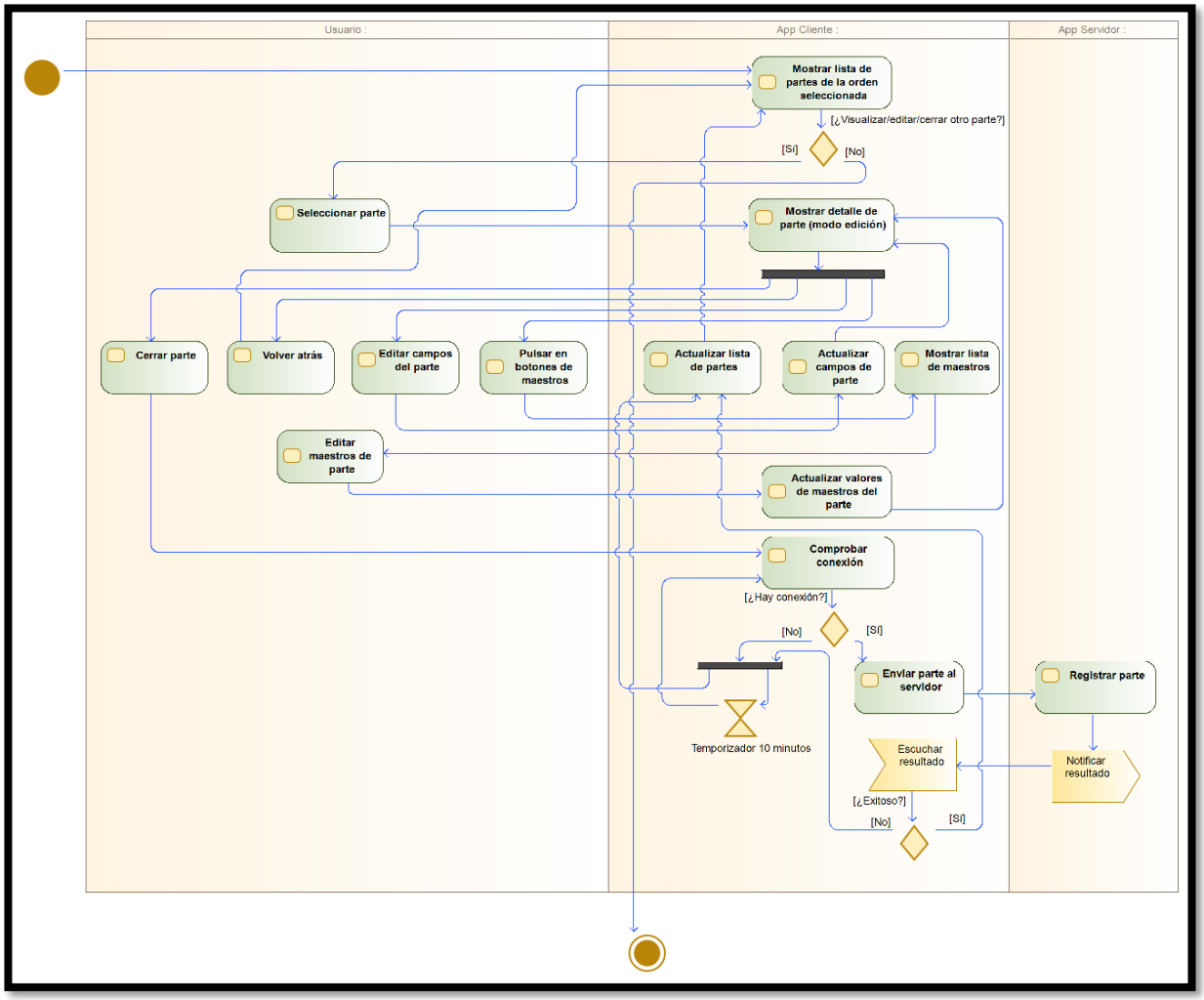


Diagrama 7: Diagrama Actividades Visualización/Edición/Cierre de Parte

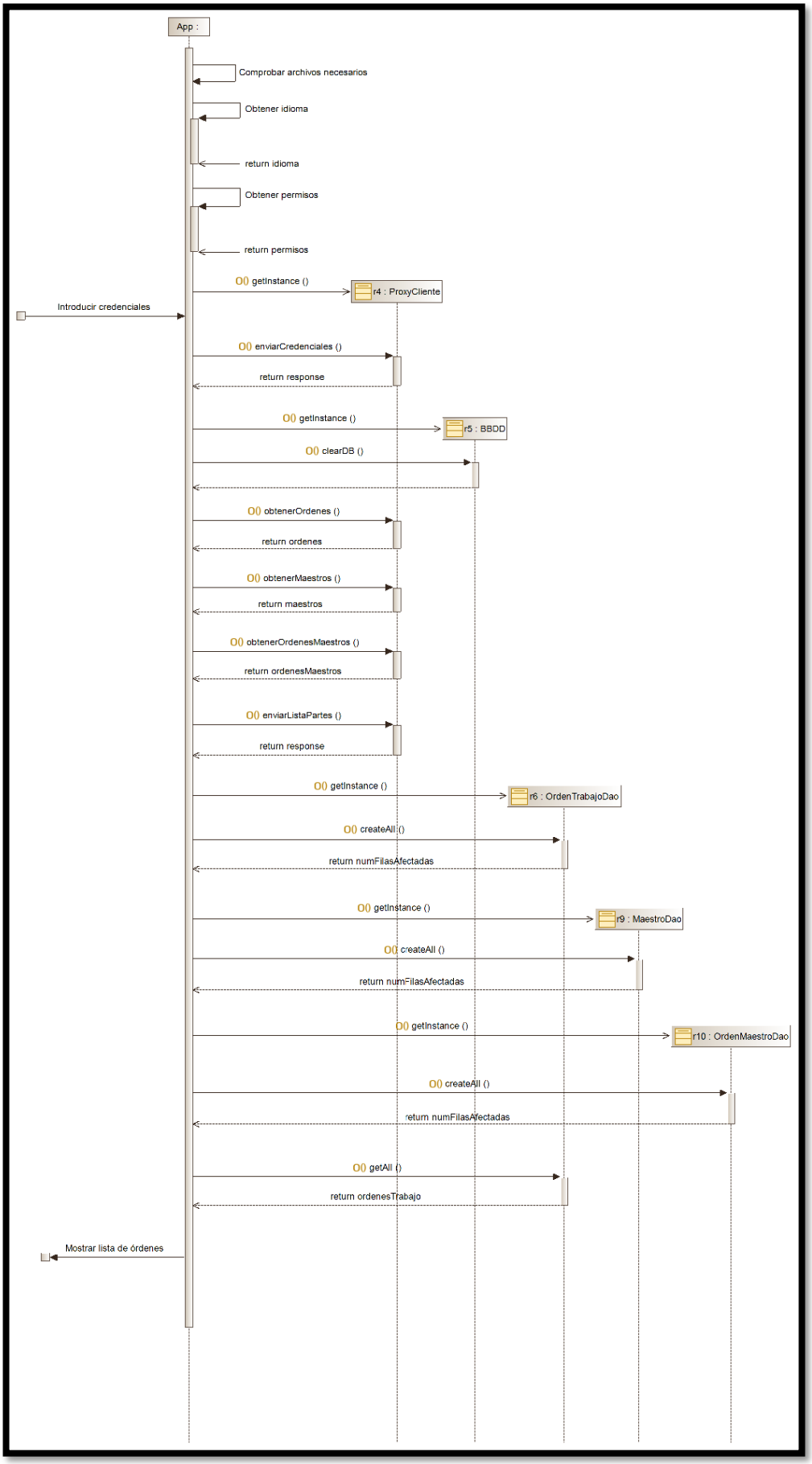


Diagrama 8: Diagrama Secuencias Inicio de Sesión

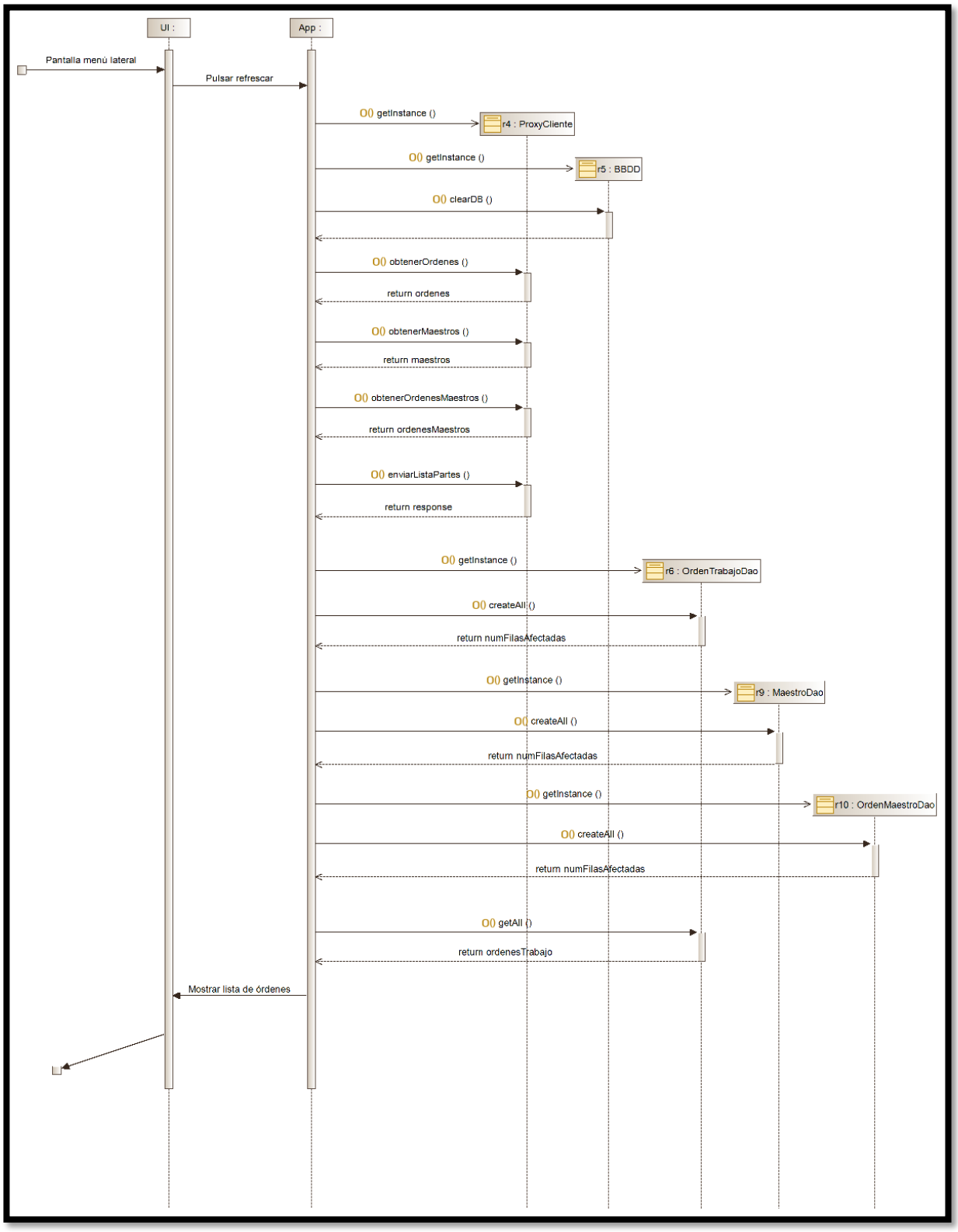


Diagrama 9: Diagrama Secuencias Refrescar

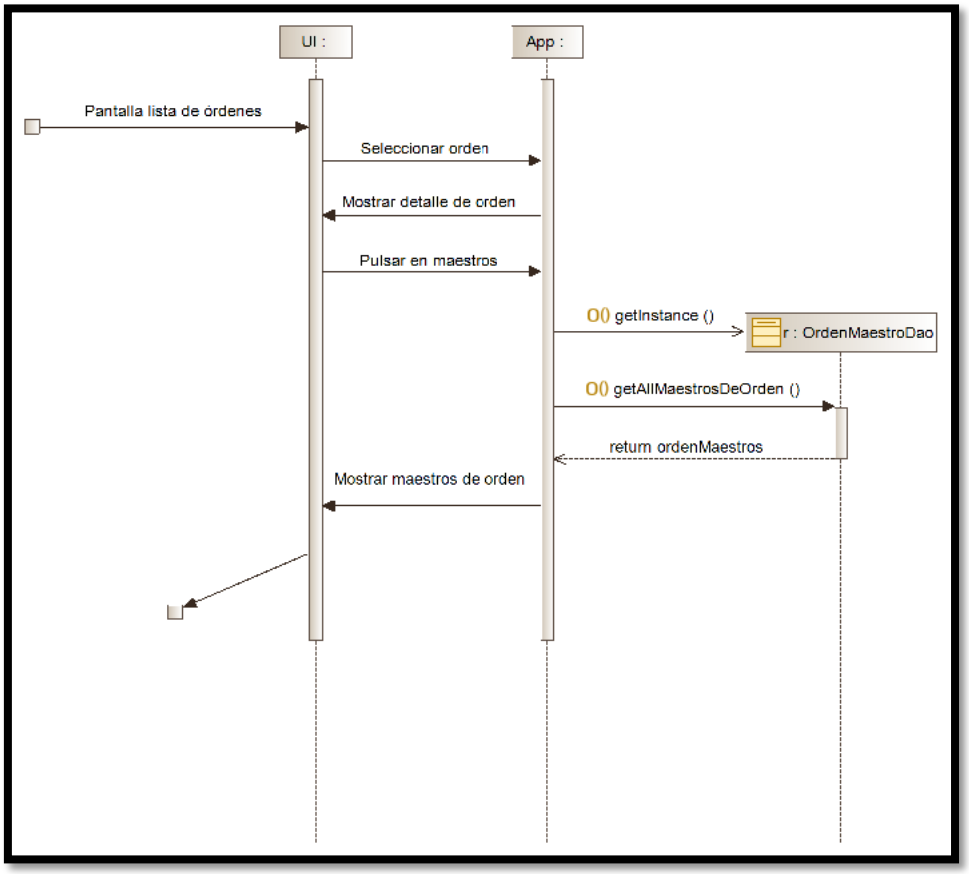


Diagrama 10: Diagrama Secuencias Visualizar Maestros de Orden

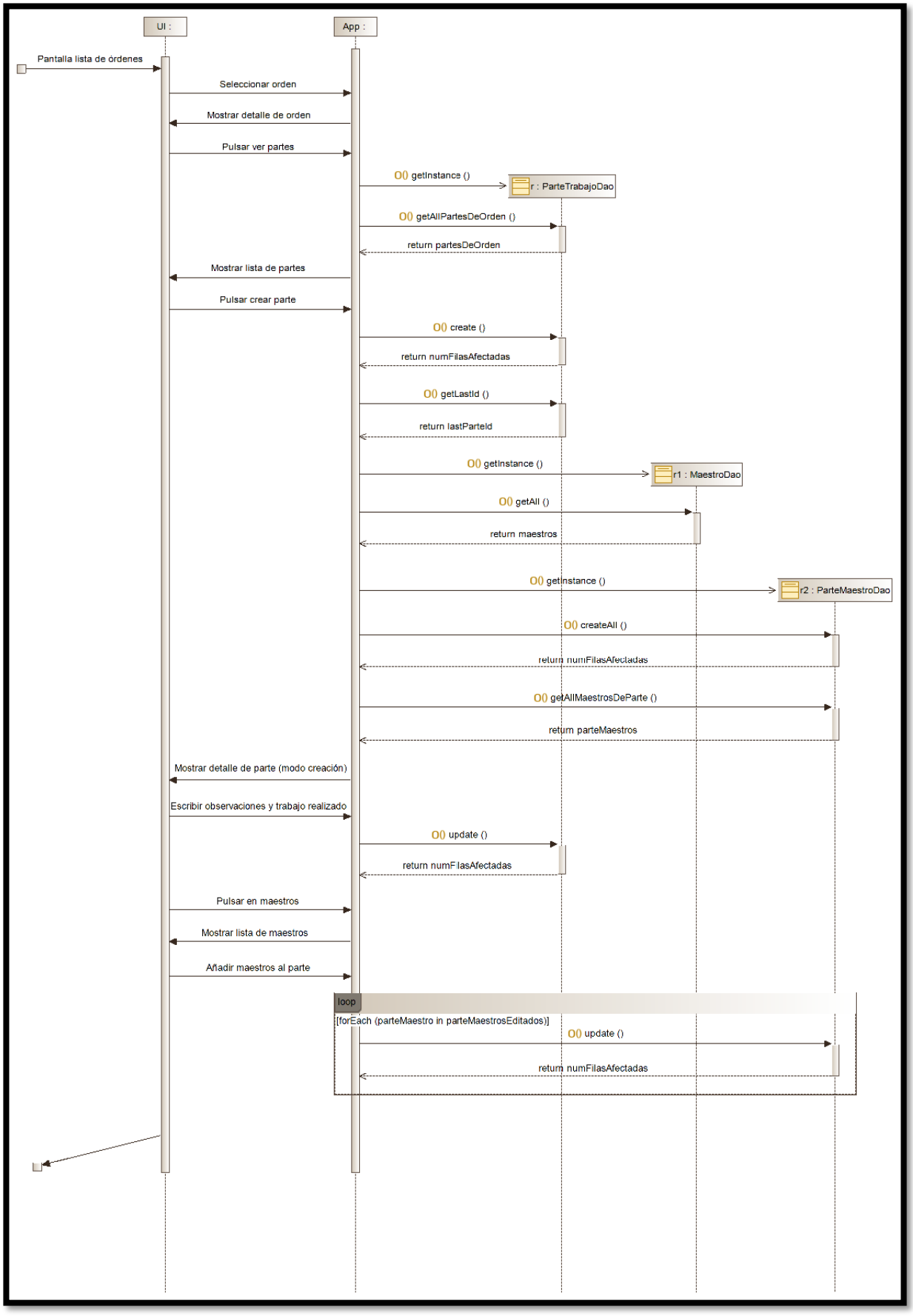


Diagrama 11: Diagrama Secuencias Creación de Parte

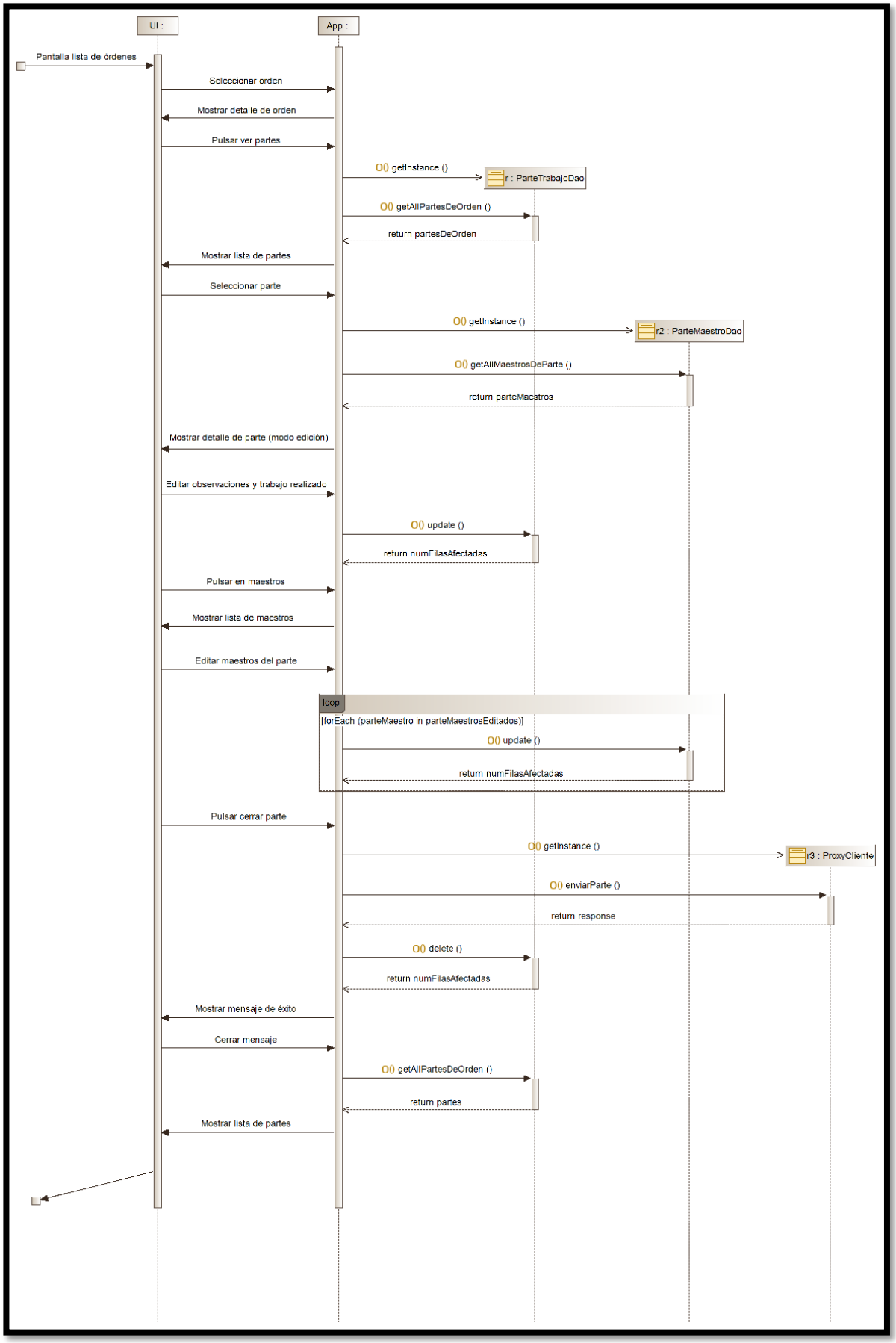


Diagrama 12: Diagrama Secuencias Cierre de Parte