—Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Memoria de Seguimiento

Desarrollo de una Aplicación para la Planificación y Gestión de Viajes en Grupo (6603)

Autor: Andrés García Solórzano

Tutor(a): Antonio Jesús Diaz Honrubia

Madrid, <<Mayo 2023>>

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: Desarrollo de una Aplicación para Ayuda y Cuidados de las Personas Dependientes

Enero 2023

Autor: Jorge Arroyo Martinez

Tutor: Sergio Paraíso Medina Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

ETSI Informáticos

Universidad Politécnica de Madrid

Agradecimiento

1 Resumen del trabajo realizado

El trabajo realizado hasta la fecha puede dividirse en dos categorías:

- Formación: Para el desarrollo de la aplicación en la que consiste este TFG ha sido necesario aprender a manejar una amplia variedad de herramientas que nunca había utilizado y refrescar las que ya conocía. Estas herramientas son:

    - Flutter: Un SDK creado por Google para el desarrollo de aplicaciones móviles multiplataforma.
    - Dart: Un lenguaje de programación desarrollado por Google, utilizado principalmente en el desarrollo de aplicaciones móviles con Flutter. Es un lenguaje orientado a objetos y de alto rendimiento.
    - Node.js: Un entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome, diseñado para crear aplicaciones de red escalables.
    - MySQL: Un sistema de gestión de bases de datos relacionales utilizado en aplicaciones web y de servidor que permite gestionar y organizar datos en tablas relacionadas.
    - Socket.IO: Una biblioteca de JavaScript que permite la comunicación bidireccional en tiempo real entre el cliente y el servidor, utilizando WebSockets y eventos.

- Desarrollo: En cuanto al desarrollo, la estructura base de la aplicación está prácticamente terminada, a falta de algunas funcionalidades sencillas de implementar como puedan ser poder crear viajes nuevos e invitar a usuarios nuevos a un viaje existente, que a pesar de ser funcionalidades clave para la aplicación, su desarrollo es muy sencillo y no tiene prioridad ya que con los usuarios y viajes que tengo de prueba me es más que suficiente para desarrollar y probar las tareas más complejas que tengo por delante.

  La aplicación ya cuenta con una primera versión del sistema de Login, tras iniciar sesión el usuario se encuentra con una pantalla en la que puede ver su próximo viaje, un calendario en el que se juntan las fechas de todos sus planes y que marca errores en caso de que dos o más planes compartan fechas, y un carrusel con los viajes que tiene más adelante. Estos viajes están representados con tarjetas con una foto, el nombre y el estado de planificación del viaje.

  Al tocar en un viaje el usuario es llevado a una pantalla en la que se muestran los distintos componentes que forman el viaje, así como la opción de añadir componentes nuevos. Estos componentes son pequeñas tarjetas en filas de 3 que muestran un resumen de la información que contienen. Tocar cualquiera de estos componentes nos llevara a una pantalla que será distinta según el tipo de componente, en esta pantalla veremos la información completa de dicho componente y podremos editarla según sea necesario. Por ejemplo, un componente de confirmaciones mostrara información relativa a la asistencia o no asistencia de los miembros invitados al viaje y un componente de tipo habitaciones mostrara información relativa a las habitaciones y camas que ocuparan los usuarios.

Actualmente me encuentro desarrollando más componentes ya que solo tengo 2 completamente funcionales, pero cabe destacar que el desarrollo de distintos componentes es muy similar por lo que la velocidad de desarrollo ha aumentado drásticamente.

2 Explicación y justificación de los cambios en el Plan de Trabajo

2.1 Revisión de la lista de objetivos del trabajo

La lista de objetivos no ha sufrido ningún cambio desde la ultima entrega.

2.2 Revisión de la lista de tareas

En cuanto a la lista de tareas originalmente proporcionada no he tenido que realizar ningún cambio al planteamiento original.

1. Elección de las tecnologías más adecuada para este proyecto
2. Estudio del estado del técnico de la cuestión
3. Familiarización con las tecnologías a emplear
4. Desarrollo del sistema principal
5. Desarrollo de diferentes módulos opcionales para los usuarios
6. Pruebas con usuarios
7. Desarrollo de la memoria
8. Preparación de la presentación

2.3 Revisión del Diagrama de Gantt

El Diagrama de Gantt si que ha sido ligeramente modificado, inicialmente tenía pensado desarrollar al completo el sistema principal y al terminar, empezar con el desarrollo de los componentes pero para poder asegurarme de que el sistema funciona correctamente ha sido necesario que los viajes tuviesen componentes que se actualizasen en tiempo real y solo apareciesen en el viaje al que pertenecía, por lo que primero he desarrollado las funcionalidades necesarias para hacer las pruebas, después he desarrollado completamente dos componentes distintos que pueden ser añadidos a cualquier viaje y ahora me encuentro añadiendo funcionalidades al sistema y creando nuevos componentes al mismo tiempo.

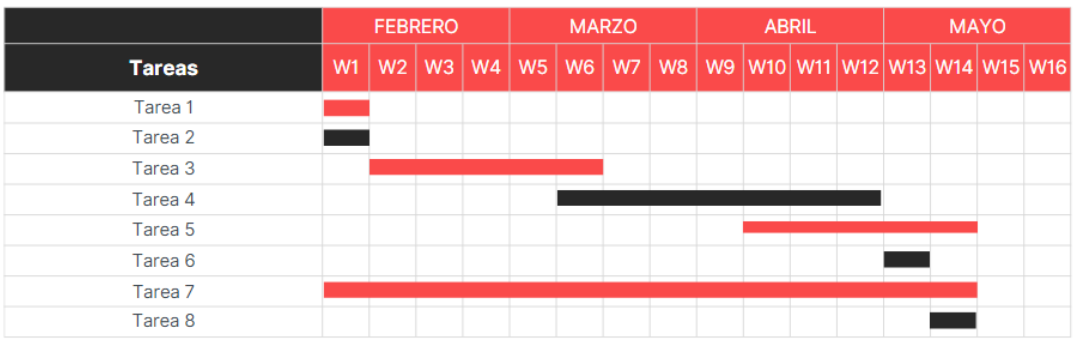


Tabla de contenidos

1 Introducción

1.1 Motivación y necesidad del proyecto

La coordinación de viaje grupal puede ser todo un desafío. Requiere de capacidad de planificación, organización, trabajo en grupo y coordinación entre otras. La dificultad de esta tarea aumenta junto al número de integrantes del grupo, cada vez es más difícil coincidir tanto en el momento como en el lugar para tomar decisiones. Esto nos lleva a buscar apoyo en las herramientas TIC, pero la falta de herramientas para este problema nos obliga a utilizar herramientas ya conocidas como chats o documentos online, que no fueron diseñados para este fin por lo que la experiencia de usuario deja mucho que desear.

1.2 Objetivos

El objetivo principal del trabajo será el desarrollo de una aplicación multiplataforma completamente funcional que ayude a sus usuarios a gestionar sus viajes, acompañándolos desde su planificación hasta el post viaje.

Este objetivo puede ser dividido en 4 objetivos específicos:

- Desarrollar una base de datos relacional con MYSQL para persistir los datos gestionar la autenticación de usuarios.
- Desarrollar la lógica y el Front-End de la aplicación que permita a los usuarios acceder esos datos de forma intuitiva.
- Desarrollar un servidor que conecte la base de datos con la aplicación.
- Desarrollar los distintos componentes que los usuarios podrán añadir a su viaje si así lo desean.

La aplicación se basa en un enfoque modular, para cada viaje podrán añadirse componentes como calendarios en los que ver la disponibilidad de cada persona, una sección de pagos en los que apuntar quién paga qué para cuadrar las cuentas, votaciones para elegir los destinos del viaje o votaciones para tomar decisiones sobre el viaje.

1.3 Planificación

Para llevar a cabo este TFG cuento con una lista de las 8 tareas que debo realizar:

1. Elección de las tecnologías más adecuadas para el proyecto
2. Estudio del estado del técnico de la cuestión
3. Familiarización con las tecnologías a emplear
4. Desarrollo del sistema principal
5. Desarrollo de diferentes módulos opcionales para los usuarios
6. Pruebas con usuarios
7. Desarrollo de la memoria
8. Preparación de la presentación

Cuento también con un plan que establece los tiempos para desarrollar cada una de las tareas previamente mencionadas.

2 Estado del arte

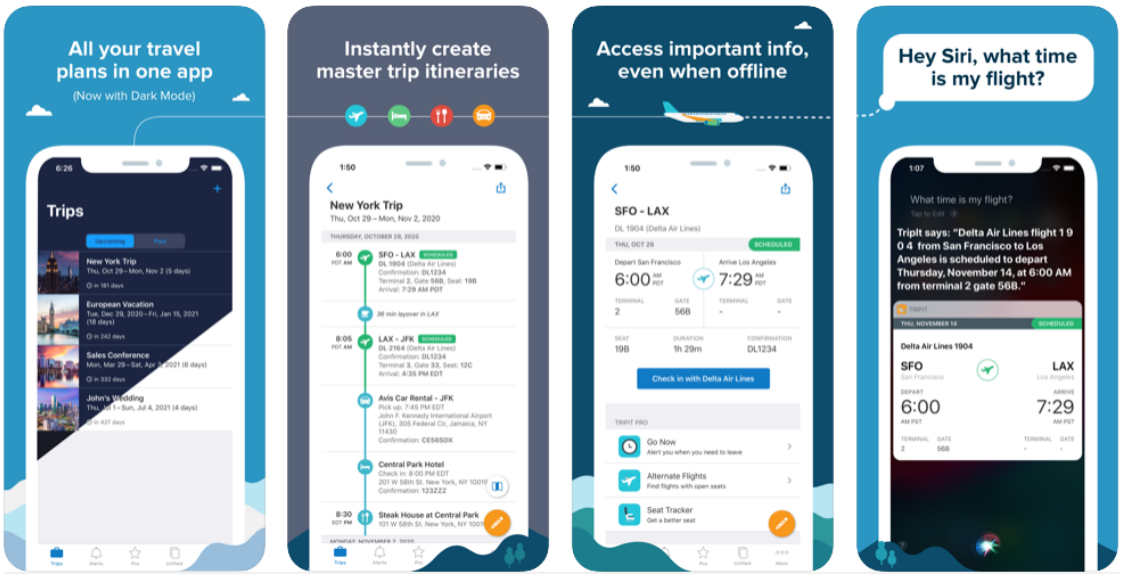
Para analizar el estado del arte en el ámbito de aplicaciones y herramientas para planificar viajes, es importante hacer una separación entre las herramientas específicamente diseñadas para este propósito y aquellas que, aunque no fueron diseñadas con ese fin, también pueden ser utilizadas en este contexto.

2.1 Herramientas específicamente diseñadas para planificar viajes:

2.1.1 TripIt

TripIt es una herramienta que funciona a modo de carpeta en la que juntar información relevante para el viaje. Cuando un usuario recibe confirmaciones de reservas de vuelos, hoteles, alquiler de coches o actividades, puede reenviar estos correos electrónicos a una dirección específica de TripIt. La aplicación extrae y organiza la información relevante, creando un itinerario maestro accesible desde cualquier dispositivo.

A diferencia de la app de este TFG, TripIt busca juntar información relevante sobre aspectos del viaje que ya han sido decididos, es decir, no es una herramienta para planear viajes sino más bien una carpeta donde juntar los resultados de la planificación de estos.

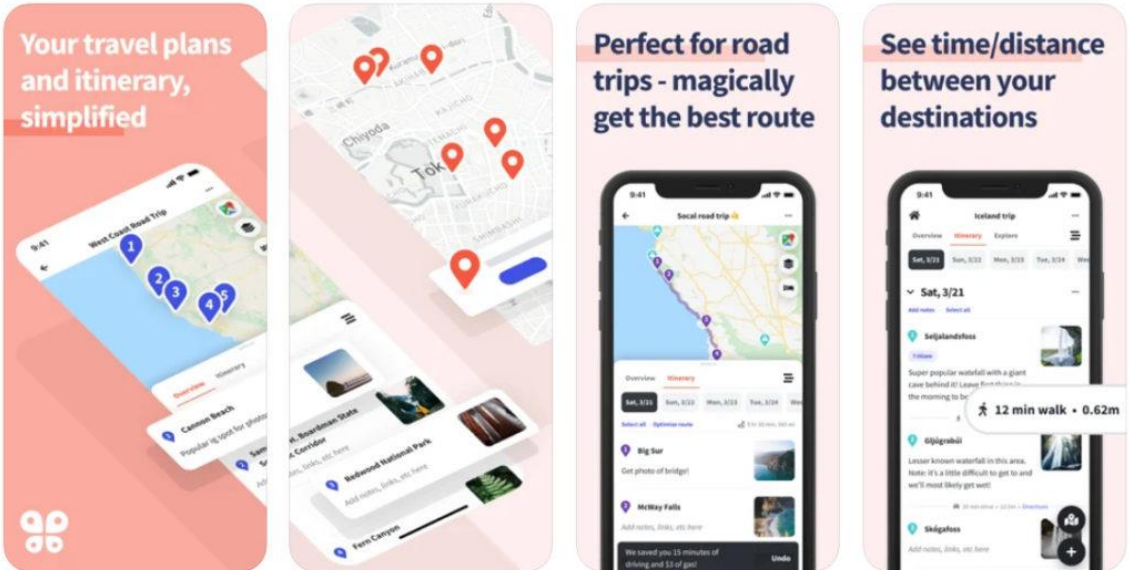


2.1.2 Wanderlog

Wanderlog es una aplicación y sitio web de planificación de viajes que permite a los usuarios crear, organizar y compartir itinerarios de viaje. Los usuarios pueden agregar hoteles, actividades, restaurantes y otros lugares de interés a su itinerario y ver todo en un mapa interactivo, todo esto en tiempo real, lo que facilita la planificación de viajes en grupo.

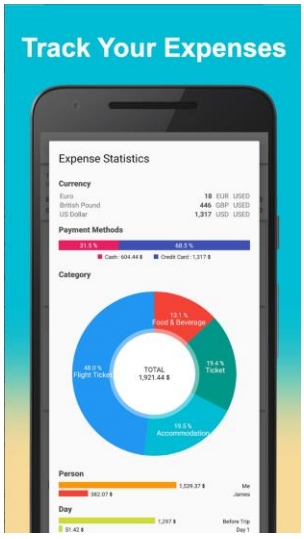
Esta es la aplicación más similar a este TFG, la diferencia clave es que esta aplicación busca ayudar a los usuarios a planear un viaje, pero al margen de

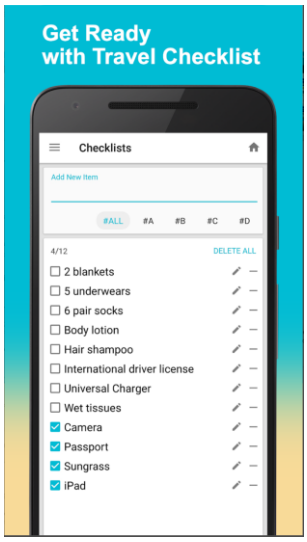
los itinerarios, no cuenta con herramientas para ayudar a los usuarios durante el mismo. De la misma forma que en Wanderlog un usuario puede añadir a su viaje notas o planes, pierde la oportunidad de añadir otras funcionalidades como gestiones de pagos durante el viaje, distribución de tareas o votaciones entre otras.

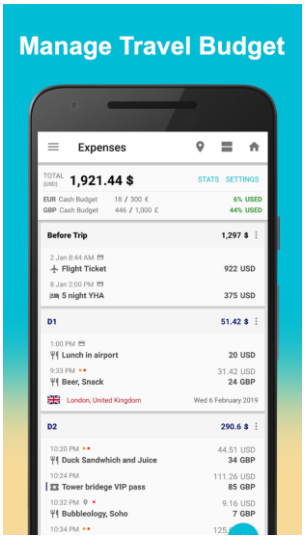


2.1.3 SaveTrip

SaveTrip es muy similar a Wanderlog y aunque cuenta con varias de las herramientas que he mencionado que Wanderlog no tiene, su principal fallo es que SaveTrip no tiene funcionalidad cooperativa por lo que solo es útil para planear viajes en solitario.



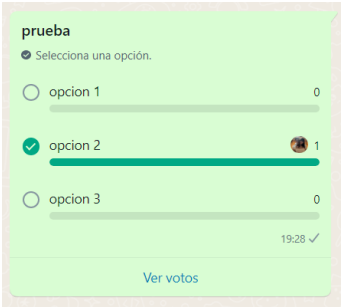


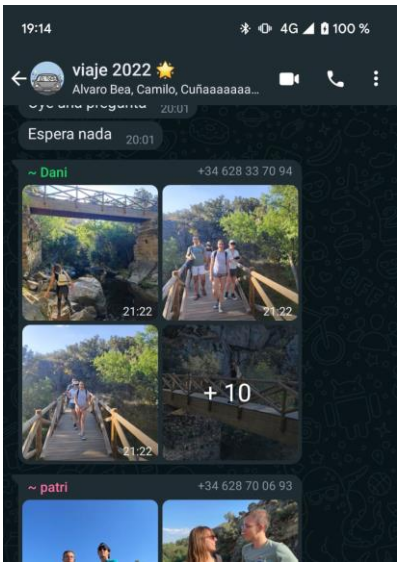


2.2 Herramientas no diseñadas específicamente para planificar viajes, pero útiles en este contexto:

2.2.1 WhatsApp

Aunque es una aplicación de mensajería, puede ser útil para planificar viajes en grupo. Los usuarios pueden crear grupos para discutir detalles del viaje, compartir documentos, ubicaciones, usando el chat a modo de lista fijando mensajes, o a modo de carpeta compartida para guardar fotos. Desde hace poco es posible incluso enviar votaciones por WhatsApp en las que cada usuario puede marcar una o más opciones.



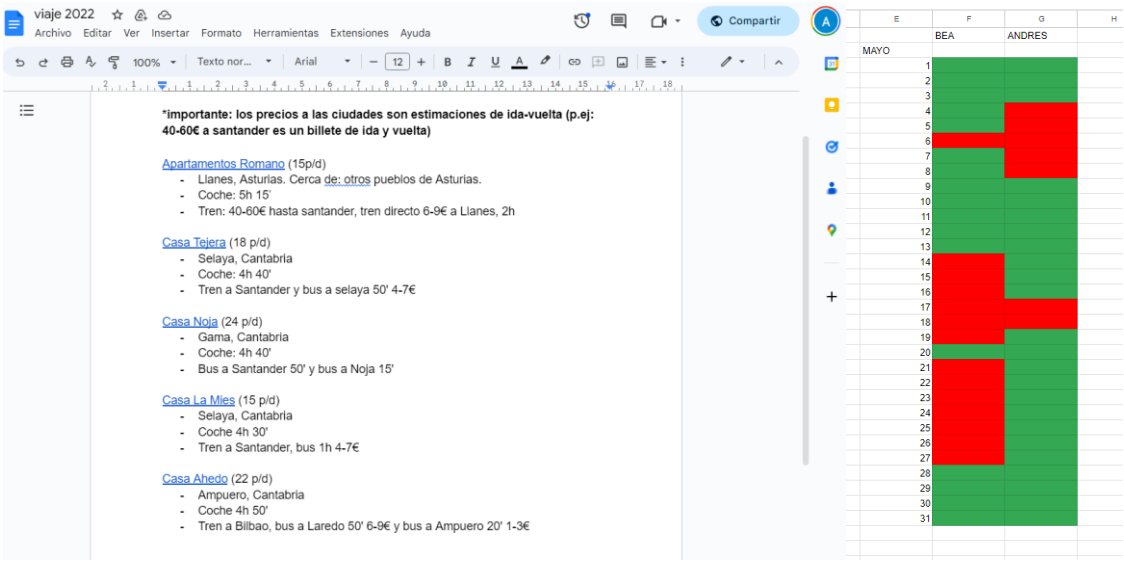


2.2.2 Google Drive

Los documentos creados y editados en Google drive se actualizan en tiempo real para los usuarios con acceso a estos lo que convierte Google drive en una plataforma muy útil para la planificación de viajes.

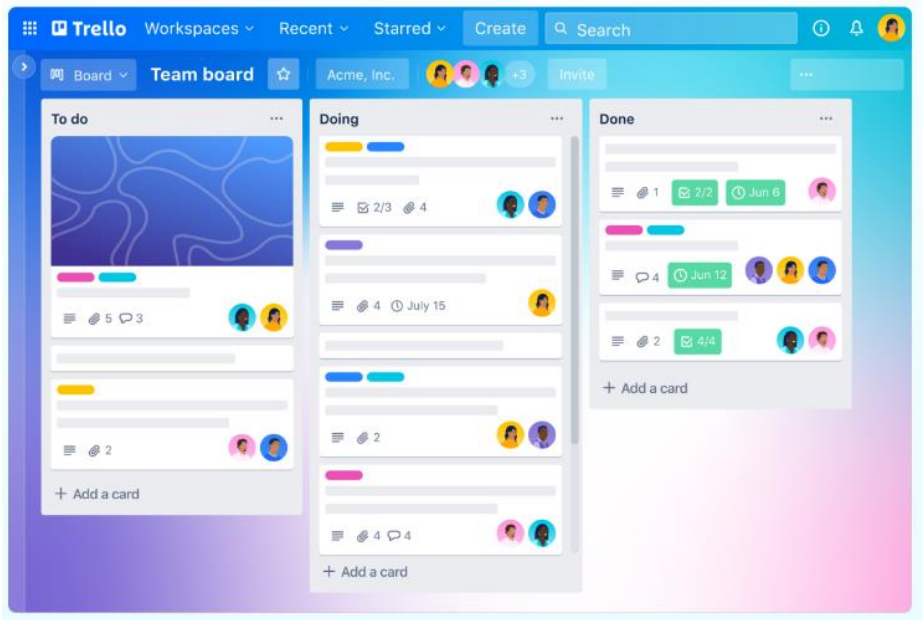
Por ejemplo, Google Docs nos permite tener un documento de texto compartido con varios usuarios por lo que resulta muy útil como de lista, por ejemplo, para juntar todos los links relevantes para un viaje como puedan ser links a los billetes de avión, a las opciones de apartamentos que alquilar o a guías turísticas entre otros.

Google Sheets además de su habitual uso como herramienta de contabilidad, puede ser usada también para planificar otros aspectos de los viajes como por ejemplo las fechas libres de cada usuario y aunque no sea la solución más elegante, es una de las opciones más simples que hay.



2.2.3 Trello

Trello es una herramienta de gestión de proyectos basada en tableros, está orientada a proyectos empresariales, pero algunas de sus funcionalidades pueden ser útiles para gestionar un viaje en grupo. Trello cuenta con herramientas para gestión de chas y eventos, además los usuarios pueden crear listas y tarjetas para organizar diferentes aspectos del viaje, como reservas, actividades y gastos. Trello cuenta con una opción gratuita que aun estando bastante limitada puede sigue siendo útil para la gestión de viajes.

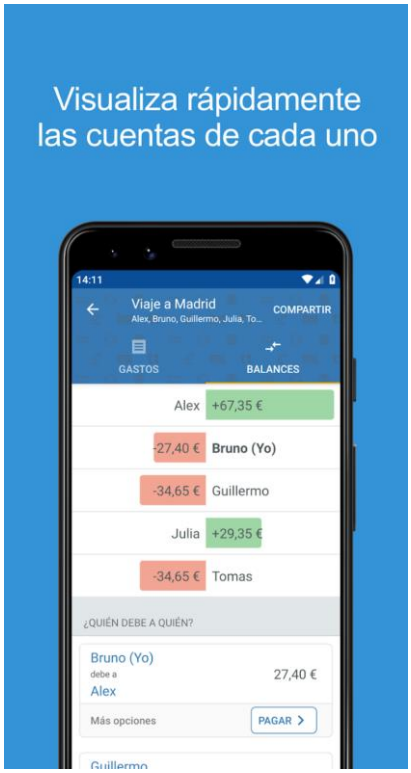


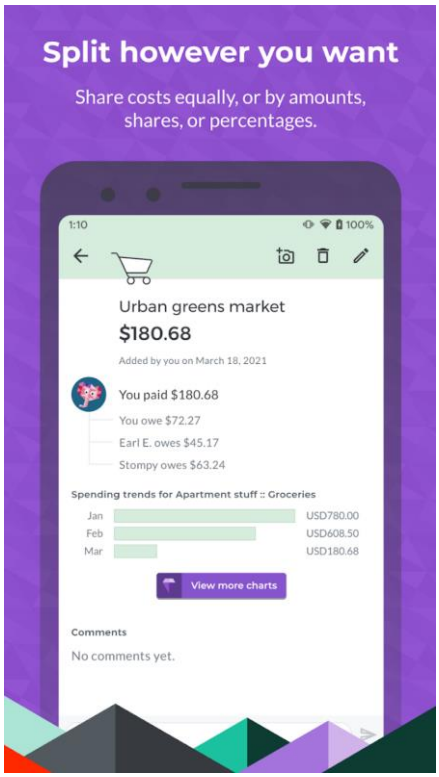
2.2.4 Tricount y Splitwise

Estas aplicaciones permiten a los usuarios gestionar los pagos en un viaje en grupo, cuando un usuario paga algo que debe ser dividido entre varias personas añade a la aplicación el gasto, quien lo ha pagado y entre que personas debe ser dividido.

La principal ventaja de estas aplicaciones es que, al acabar el viaje, simplifican el proceso de pago ya que tienen un sistema que reduce la cantidad de transacciones necesarias, por ejemplo, si A debe 5 euros a B, B debe 5 euros a C y C debe 6 euros a A, solo sería necesario que C pague 1 euro a A.

Esta simplificación es muy útil en viajes en grupo ya que es común que una sola persona pague por todos y luego se hagan cuentas mas adelante pero cuando el número de miembros del viaje aumenta esta tarea se vuelve muy compleja.



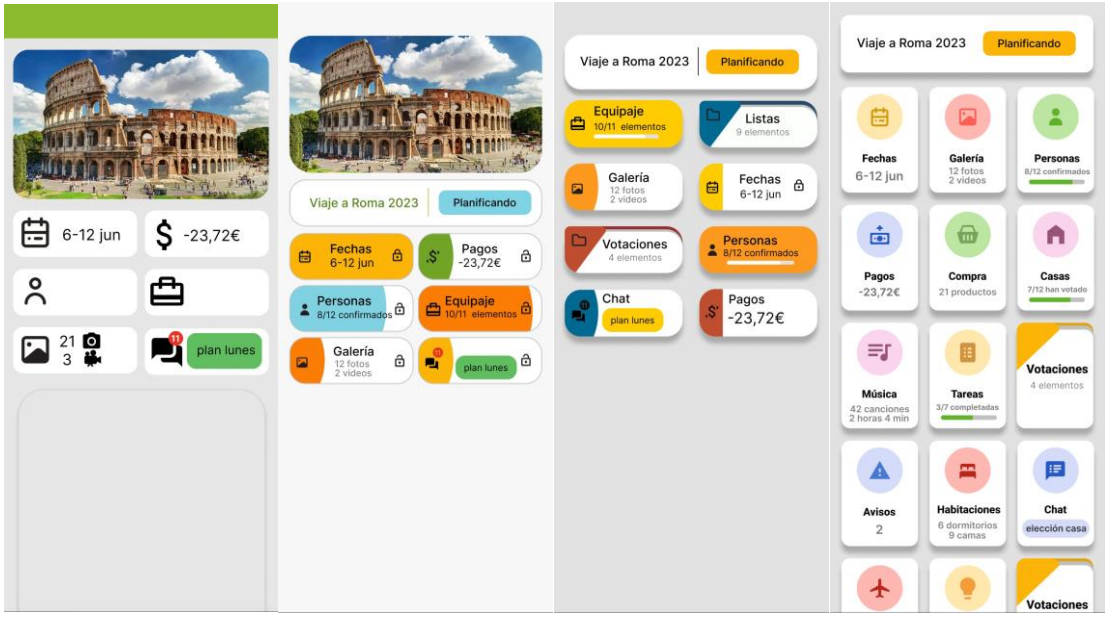


3 Tecnologías empleadas

3.1 Figma

Para los primeros diseños de la interfaz de usuario he utilizado Figma, una herramienta basada en la nube que permite diseñar interfaces de usuario de forma rápida y sencilla. El uso de componentes predefinidos como cards con altura y sombras me permitió hacer varios diseños en poco tiempo para hacerme una idea de como quería que fuera mi aplicación incluso antes de empezarla.

Además Figma es una herramienta multiplataforma y con funcionalidades cloud por lo que podía diseñar mi apliacion desde mi ordenador y visualizarla he interactuar con ella al momento en mi móvil, esto me fue de gran ayuda ya que me fnadsifuansfiaus para el tamaño que tienen las cosas en el móvil.



3.2 Dart

La decisión más importante para el desarrollo de este trabajo fue elegir un lenguaje de programación para la aplicación cliente. Tras mucho investigar decidí desarrollar el proyecto utilizando Dart, un lenguaje de programación desarrollado por Google que esta ganando popularidad en los últimos años. Algunas de sus principales ventajas son:

Orientado a objetos:

Dart es un lenguaje de programación orientado a objetos lo que facilita la creación de estructuras de datos complejas y reutilización de código mediante herencia y polimorfismo.

Compilación Just in Time (JIT):

El código escrito en Dart puede ser compilado a código maquina en tiempo real, es decir, no es necesario detener la ejecución del programa para ver en tiempo real los nuevos cambios.

Null Safety:

La null safety es una característica de Dart que distingue entre variables que pueden ser nulas y las que no pueden serlo, ayudando a prevenir errores de null pointer. Existen tipos distintos para las variables que puedan ser nulas, por ejemplo, un String debe tener un valor, pero si existe la posibilidad de que no se le asigne ninguno, Dart obliga a cambiar el tipo a "String?" y obliga a manejar explícitamente los casos nulos, lo que resulta en código más seguro y predecible, y reduce la cantidad de errores en las aplicaciones.

Fácil de aprender:

La sintaxis de Dart es sencilla y similar a otros lenguajes de programación orientados a objetos que he utilizado previamente como Java o C#.

Flutter:

Un framework de Dart creado por Google para el desarrollo de aplicaciones móviles multiplataforma que fue la principal la razón para elegir Dart. Más adelante hablare en detalle de esta herramienta.

3.3 Flutter

Flutter es un framework para Dart creado por Google. Se utiliza para desarrollar aplicaciones para Android, iOS, Linux, Mac, Windows, y la web desde una única base de código. Algunas de sus principales ventajas y razones por las que lo he elegido son:

Rendimiento:

Al compilar a código nativo y tener su propio motor de renderizado, Flutter puede proporcionar un rendimiento superior en comparación con otros marcos de desarrollo multiplataforma.

Widgets:

Los componentes que forman la interfaz de la aplicación Flutter se denomian Widgets. Los Widgets se organizan en una estructura de árbol, cada Widget puede tener uno o mas Widgets hijo dependiendo del componente. Estos Widgets vienen predefinidos y son personalizables lo que facilitan el diseño de la interfaz y permite crear aplicaciones muy similares a las programadas de forma nativa ya que pueden importarse Widgets con diseño Material Design (El utilizado en los dispositivos Android) y Cupertino (Para dispositivos IOS) y mostrar unos u otros dependiendo de la plataforma en la que se ejecute la aplicación.

Hot Reload:

Flutter utiliza la compilación JIT de Dart para ofrecer la funcionalidad de Hot Reload, que permite ver los cambios en el código en tiempo real sin necesidad de volver a compilar la aplicación y sin perder el estado de las variables. Esta funcionalidad es particularmente útil ya que además de permitir ver la aplicación mientras se desarrolla, lo que facilita el diseño de la interfaz y la resolución de errores ya que el programa no termina su ejecución cuando falla y puede ser arreglado en el momento sin necesidad de recompilarlo.

3.4 Node.Js

Node.js es un entorno de ejecución de JavaScript basado en el motor V8 de Chrome. Fue diseñado para construir aplicaciones de red escalables y eficientes, especialmente servidores de entrada/salida orientados a eventos, lo que lo hace ligero y eficiente, perfecto para aplicaciones de datos en tiempo real.

3.5 MYSQL

Un sistema de gestión de bases de datos relacionales utilizado en aplicaciones web y de servidor que permite gestionar y organizar datos en tablas relacionadas.

.

.

.

3.6 Socket.IO

Socket.IO es una biblioteca de JavaScript que permite la comunicación bidireccional en tiempo real entre el servidor y los clientes web o móviles. Permite que el servidor envíe datos a los clientes cuando tiene nuevos datos disponibles, sin necesidad de que los clientes hagan una solicitud HTTP específica. Este patrón de comunicación es muy útil para las aplicaciones que necesitan actualizaciones en tiempo real.

3.7 No IP

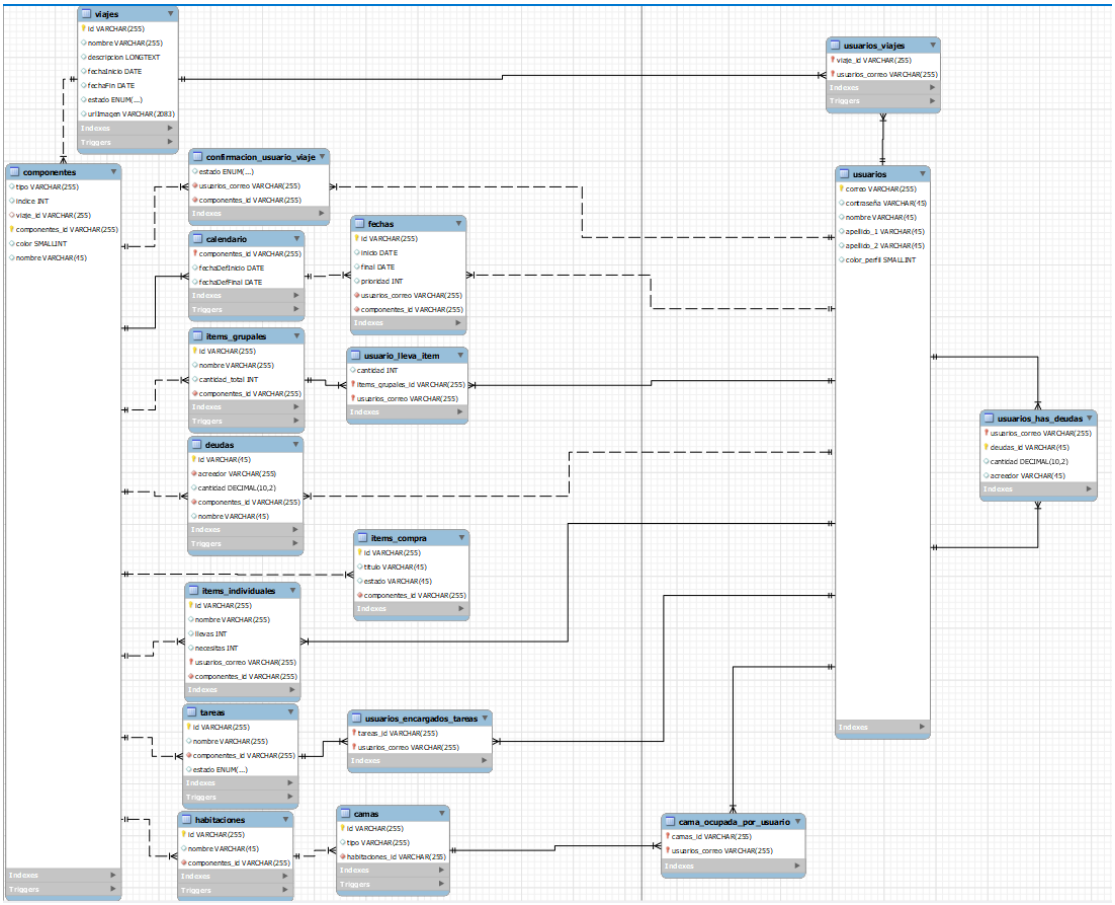
.

.

.

4 Desarrollo de la aplicación

4.1 Base de datos



Usuarios:

Los usuarios se crean en la base de datos con un identificador que es su correo, una contraseña dos apellidos y color del perfil ya que en algunos componentes se muestra una miniatura de los usuarios que participan en ndsfjknsdfj usando la primera letra de su nombre y la primera letra de su apellido sobre un fondo del color de su perfil.

Viajes:

Los viajes tienen un identificador autogenerado, un nombre, una descripción que actualmente se crea vacía porque aun no se usa, una fecha de inicio, una fecha de final, un URL para la imagen del viaje y un estado que puede tomar los valores 'Planificando', 'Planificado', 'En curso', 'Finalizado' y 'Cancelado'.

Componentes:

Los componentes estas diseñados de forma que implementan la llamada estructura de los modelos Entidad relación extendidos. Todos los componentes cuentan con un identificador autogenerado, un tipo, un índice para su posición en la lista de componentes de la aplicación, un id del viaje al que pertenece el componente que es clave foránea de la tabla viajes, un int para el color de forma similar a el color del usuario, y un nombre.

En la aplicación y el servidor, además de estas propiedades todos los objetos componente tienen otra llamada subcomponente que consiste en una lista de objetos con propiedades distintas según el tipo del componente. Por ejemplo,

un componente de tipo Tareas tendrá en su subcomponente una lista de objetos tarea, cada una con su nombre, su estado, su id autogenerado y la lista de usuarios encargados. Por otro lado, un componente de tipo distribución de habitaciones tendrá en su subcomponente una lista de objetos habitación, cada habitación de la lista tiene sus propiedades y una lista con las camas que contiene, cada cama de nuevo con sus propiedades y una lista de los usuarios que la ocupan.

Las filas de las 7 tablas que están conectadas a la tabla componentes sin contar la tabla de viajes y que almacenan la información necesaria para la propiedad subcomponente, reciben como Foreign de la tabla componentes el id del componente al que pertenecen. En una misma tabla puede haber varias filas con el mismo valor en esta columna, por ejemplo, puede haber varias tareas dentro de un mismo componente, pero nunca puede ocurrir que dos tablas distintas comportan este valor, por ejemplo, es imposible que un mismo componente contenga tareas y deudas.

Esta forma de diseñar la base de datos para que distintos componentes puedan tener distintas propiedades se conoce como modelo entidad relación extendido ys e utiliza para jdsnfjksdfnsdk y polimorfimos,

Confirmaciones:

La tabla confirmaciones tiene en cada row el id de un usuario, tomado como clave foránea de la tabla 'usuarios', el id del componente correspondiente, también como clave foránea, y un estado, que es un campo enumerado que puede tener los valores "pendiente", "confirmado" o "no asistirá".

Cuando se crea un nuevo componente de asistencia en un viaje, este genera automáticamente una fila en la tabla 'confirmación_usuario_viaje' para cada usuario invitado al viaje. Esta fila contiene el id del usuario, el id del componente y el estado inicial "pendiente".

Además, si un usuario se añade a un viaje posteriormente, un disparador (trigger) se encargará de insertar la nueva fila correspondiente a dicho usuario en la tabla. De igual manera, si se elimina un componente de asistencia, otro disparador se activará para eliminar todas las filas asociadas a ese componente en la tabla.

Calendario:

La tabla calendario tiene 3 propiedades, la primera es el id del componente al que pertenece que además de ser una clave foránea es la clave primaria de la fila, las dos propiedades restantes son una fecha de inicio y una fecha de finalización, ya que el objetivo del componente 'calendario' es coordinar una fecha común entre todos los invitados al viaje.

Fechas:

Además, el componente 'calendario' puede tener asociados varios eventos. Cada usuario puede marcar en el calendario común los días en los que no está disponible o los días en los que preferiría no programar el viaje. Al hacerlo se añade a la tabla fechas una nueva fila, esta fila tendrá 6 propiedades que son: Un identificador autogenerado, una fecha de inicio, una fecha fin, un int que

indica la fnajnfasjk del evento, el id del componente al que pertenece y el id del usuario que ha añadido el evento.

Items grupales:

.

.

.

.

Asignacion ítems

.

.

.

.

.

Deudas

.

.

.

Dgfhfg gggg

.

.

.

.

Tareas

.

.

.

.

Asignación tareas

.

.

.

Habitaciones

.

.

.

Camas

.

.

.

Asignaciones cama

.

.

.

4.2 Conexión App-Servidor

4.2.1 Inicio de la conexión

La comunicación entre la aplicación y el servidor se inicia cuando un usuario abre la aplicación. Al iniciar, se crea en esta una instancia de la clase SocketService que intenta establecer una conexión con el servidor con la URL 'http://viajesapp.ddns.net:3000/'. Esta dirección se mantiene constante gracias a NO IP, que permite mantener un nombre de dominio fijo incluso si la dirección IP del servidor cambia adsdad.

Cuando la conexión se establece correctamente, se dispara en la aplicación el evento 'onConnect'. En respuesta, la aplicación cambia el estado del servidor a 'Online' y notifica a todos los listeners de este cambio. En caso de que la conexión se pierda por cualquier motivo, se dispara el evento 'onDisconnect'. Esto provoca que el estado del servidor cambie a 'Offline' y se notifique a todos los listeners.

Más adelante se explicará en profundidad la necesidad de mantener en todo momento info de cuanto esta updated nfdsfnsdjf

Este proceso estable una comunicación bidireccional entre el cliente y el servidor. El servidor almacena la información de la conexión con el cliente en la variable 'client', lo que permite identificar la conexión particular. A partir de este punto, el servidor escucha activamente el resto de eventos que el cliente pueda emitir.

4.2.2 Gestión de la comunicación cliente-servidor

Envío al usuario que disparó el evento

Cuando un cliente realiza una acción en la aplicación, como solicitar información de un viaje, dispara un evento específico en el servidor. El servidor, al recibir este evento, realiza las operaciones necesarias y devuelve la respuesta directamente al cliente que disparó el evento. Para ilustrar, en el caso del evento 'solicitud-viajes', el servidor obtiene la información del viaje y la devuelve al cliente que realizó la solicitud.

```
client.on('solicitud-componentes-viaje', async (payload) => {
    try {
        client.join(payload);
        const UsuariosViaje = await usuarios.getUsuariosViaje(payload);
        const listaComponentes = await componentes.getComponentes(payload);
        client.emit('lista-usuarios', UsuariosViaje);
        client.emit('lista-componentes', listaComponentes);
    } catch (err) {
        console.error('Error enviando la lista de usuarios o componentes:', err);
    }
})
```

Envío a Rooms

La biblioteca Socket.IO facilita la implementación de rooms o salas. Estos son canales a los que un cliente puede suscribirse y recibir mensajes enviados a ese canal específico.

Cuando un usuario inicia sesión, y pasa a la pantalla de inicio, dispara el evento “solicitud-viajes” para obtener una lista con todos los viajes a los que está invitado. Además de enviar al usuario la información solicitada, el servidor añade al cliente a los rooms asociados a todos sus viajes. Así, el cliente recibe en tiempo real todas las actualizaciones que otros usuarios realizan sobre el viaje mientras está utilizando la aplicación.

Cuando un usuario cierra la aplicación, se le elimina de todos los rooms a los que esté suscrito. Si algún room queda vacío tras esto, se elimina. Además, si se intenta añadir a un usuario a un room que no existe, el servidor crea automáticamente un nuevo room.

```
client.on('creacion-nuevo-componente', async (payload) => {
    try {
        const Componente = await componentes.addComponente(payload.tipo, payload.viaje_id, payload.color, payload.nombre);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
    } catch (err) {
        console.error('Error creando componente:', err);
    }
})
```

Envío a un cliente distinto al que disparó el evento

Para poder invitar usuarios a un viaje y que estos vean el nuevo viaje al instante es necesario poder encontrar el canal de conexión entre dicho usuario y el servidor. Para desarrollar esta funcionalidad he implementado un mapa 'userToSocket'. Cuando el cliente se registra con un correo electrónico, la aplicación del usuario emite el evento 'register-user', pasando el correo electrónico como dato. El servidor que conoce el identificador de la conexión por la que ha llegado este evento, agrega una nueva entrada al mapa 'userToSocket', con el correo electrónico del usuario y la identificación del cliente.

Cuando un usuario invita a otro a un viaje introduciendo el correo del segundo en la aplicación, este correo se envía al servidor junto con el resto de información necesaria para añadir la invitación del usuario en la base de datos y para enviar al usuario invitado un evento que provoque que la información de su pantalla se actualice.

```
client.on('añadir-usuario-a-viaje', async (payload) => {
    try {
        console.log('añadir usuario a viaje solicitado')
        const UsuarioViaje = await usuarios.addUsuarioViaje(payload.correo, payload.viaje_id);
        const UsuariosViaje = await usuarios.getUsuariosViaje(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-usuarios-viaje', UsuariosViaje);
        io.to(userToSocket.get(payload.correo)).emit('añadido-a-viaje');
    } catch (err) {
        console.error('Error añadiendo usuario a viaje:', err);
    }
})
```

4.3 Flujo de datos

4.3.1 Persistencia de datos y Actualización en tiempo real

Todas las pantallas de la aplicación contienen un método llamado initState cuya función es mantener actualizada la información que se muestra en la pantalla.

Toda la información relativa a viajes y componentes esta almacenada únicamente en la base de datos por lo que antes de movernos a cualquier pantalla antes hay que enviar al servidor una petición con la información necesaria para que pueda devolver los datos con los que dibujar la pantalla.

4.3.2 Información que se conserva en local

El diseño de la aplicación se basa en un enfoque de "conservación mínima de datos", manteniendo localmente solo la información estrictamente necesaria para las interacciones con el servidor. Este enfoque ayuda a garantizar que la información que se muestra a los usuarios esta siempre actualizada.

La información que se conserva localmente son los parámetros necesarios para las peticiones. Por ejemplo, cando un usuario abre la pantalla de un viaje lo primero que hará la aplicación antes de mostrar nada es enviar una petición al servidor con el Id del viaje que se habrá guardado en local en el momento en el que el usuario pulsó el botón que lo llevó a esta pantalla. Si el Id del viaje no se conserva entre pantallas es imposible obtener el resto de la información.

Esta información esencial se mantiene en cuatro campos específicos dentro de la clase SocketService, permitiendo que cualquier pantalla pueda acceder y modificarlos según sea necesario.Estos campos son:

- Usuario: Cuando el usuario inicia sesión en la aplicación, el servidor envía junto a la confirmación todas las propiedades del usuario excepto su contraseña. Antes de llevar al usuario a la pantalla de inicio se guardan estas propiedades del usuario en el SocketService. Esta información es útil para saber que viajes mostrar en la aplicación y para los componentes que muestran información personalizada a cada usuario (como el componente de equipaje grupal que ademas de la lista común muestra una lista de los ítems que el usuario tiene asignados) entre otros casos.

- Viaje: Cuando el usuario toca cualquier elemento de la aplicación que le vaya a llevar a la pantalla de un viaje, como la lista de viajes o el calendario, se guarda el id del viaje en el SocketService y después se mueve al usuario a la nueva pantalla.

- Componente: Cuando el usuario toca cualquier elemento de la aplicación que le vaya a llevar a una pantalla de componente, se guarda en el SocketService el id y el tipo de dicho componente y después se mueve al usuario a la nueva pantalla. Esto permite al servidor saber cual es el componente que esta siendo modificado ya que el id se adjunta a todos los payload relacionados con las modificaciones de componentes.

- Estado de la conexión: En el momento en el que la aplicación establece conexión con el servidor este campo tendrá valor true, la utilidad de esta variable es que si se pierde la conexión el valor vuelve a false e inutiliza todos los botones que envíen peticiones al servidor, además los vuelve grises y no interactivos para que el usuario lo entienda fácilmente.

4.3.3 Peticiones de la aplicación al servidor

4.3.3.1 Init state

Todas las pantallas de la aplicación contienen un método llamado initState cuya función es obtener la información actualizada del servidor y una vez recibida comienza a ejecutar el código para renderizar la pantalla con la información actualizada.

Lo primero que hace esta función es obtener una instancia del servicio SocketService que como ya se ha explicado, además de manejar la conexión con el servidor, contiene la información necesaria para realizar las peticiones a este.

A través de SocketService enviamos un evento “solicitud-componentes-viaje” junto al id del viaje en el que nos encontramos, que como mencionamos antes quedo guardado en el SocketService antes de enviar al usuario a esta pantalla.

4.3.3.2 Eventos

4.3.4 Mapeo de la base de datos a objetos JS ejemplo componentes

Cuando se dispara un evento, el servidor recibe y procesa esa información. En muchos casos, esto implica interactuar con la base de datos para agregar, eliminar o modificar la información según sea necesario. Después de hacer los cambios correspondientes en la base de datos, el servidor recoge la nueva información y la guarda en objetos definidos en el código del servidor.

Las clases que contienen la definicion de estos objetos y los metodos que ejecutan querys para modificar su infromacion en la base de datos y o mapearla e dichos objetos se encuentran definidas en la carpeta models del servidor.

Además de usuarios viajes y componentes, hay definidas clases para el contenido de los componentes.

```
client.on('añadir-item-compra', async (payload) => {
    try {
        console.log('añadir item compra solicitado')
        const Item = await itemsCompra.addItemCompra(payload.id_componente, payload.titulo);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-this-componente', await componentes.getComponenteCompra(payload.id_componente));
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
        console.log("item añadido");
        console.log(Item);
    } catch (err) {
        console.error('Error añadiendo item compra:', err);
    }
})
```

Cuando el evento llega al servidor este ejecuta el codigo necesario para devolver la infromacion necesaria a los clientes que se consideren necesarios.

Para cada evento que la aplicación puede disparar existe en la clase socket.js un fhbdsjsfbh que escucha ese evento y define los pasos a seguir en caso de que se dispare. El objetivo de estos pasos es

- Modificar la informacion de la base de datos: el primer paso tras escichar el evento es llamar a las funciones necesarias para que ejecuten los cambios asociados al evento usando la informacion del payload. Por ejemplo si el evento disparado es “añadir-item.compra” se llama a la funcion addItemCompra de la clase itemsCompra. Todos los paraemtros que necesite la funcion estaran en el payload.
- Mapeo de la informacion:

Una vez se ha modificado la informacion de la base de datos, se llama a una funcion encargada de leer esa informacion y mapearla en objetos para enviarla a los clientes.

Para poder mapear estos objetos hay definidas una serie de clases. Las clases en singular sirven para definir el objeto como por ejemplo la clase tarea, mientras que la clase en plural define todas las funciones de lectura escritura actualizacion y borrado necesarias para dicho objeto.

Cuando el servidor recive un evento de la aplicación,

En la clase viajes se definen las funciones necesarias para la creacion, lectura, actualizacion y borrado de viajes que seran llamadas desde los cursos de accion de los eventos de la clase socket. Los parametros que reciben vienen del payload que se envia junto a los eventos desde la aplicación.

En la clase componentes se define las funciones de creacion, lectura y borrado de componentes,

Esta solicitud llega al servidor NodeJs que llama a getComponentes, una función que llamara a las funciones necesarias según el tipo de componente, funciones que ejecutan querys para obtener información de la base de datos, aplican la lógica que sea necesaria para cada caso (por ejemplo contar la cantidad de tareas marcadas como completadas) y crea un objeto Componente con esa información.

Un objeto componente sigue la siguiente estructura:

```
Componente {
   id: 'f97b50b0-7330-4595-9490-465aea11c4c4',
   tipo: 'habitaciones',
   color: 2,
   subcomponente: [ [Habitacion] , [Habitacion] , [Habitacion] ],
   indice: 1,
   nombre: 'Prueba',
   propiedad_1: 0,
   propiedad_2: 0
 }
```

4.3.5 Respuestas del servidor a la aplicación

4.3.6 actualización de la informacion

4.3.7 Actualización de la Pantalla de Viaje en Tiempo Real

Para asegurarnos de que la información de viaje que se muestra a los usuarios siempre está actualizada, cada pantalla de la aplicación contiene un método llamado initState. Este método tiene como función principal obtener la información actualizada del servidor y, una vez que se ha recibido, comienza a ejecutar el resto del código que mostrará dicha información en la pantalla.

En este proceso, la aplicación envía un evento "solicitud-componentes-viaje" al servidor junto con el ID del viaje en cuestión. Esta solicitud es procesada por el servidor, que llama a la función getComponentes. Esta función lleva a cabo las consultas necesarias para obtener la información de la base de datos, aplica la lógica necesaria para cada caso y crea un objeto Componente con esa información.

Los objetos Componente siguen una estructura específica y pueden incluir una variedad de tipos de subcomponentes, como listas de deudas, tareas, eventos, etc. Cada subcomponente tiene sus propias características y propiedades. Cuando la lista de objetos Componente está completa, se guarda en la variable listaComponentes y se envía al cliente que hizo la solicitud original.

Por último, cuando la aplicación recibe el mensaje, resetea la variable _componentes y mapea el payload para convertirlo en un objeto Componente en Dart.

Este sistema puede suponer una mayor carga para el servidor, pero garantiza que la información que se muestra a los usuarios esté siempre actualizada.

4.3.8 Clases

4.3.9 Eventos

```
client.on('añadir-item-compra', async (payload) => {
    try {
        console.log('añadir item compra solicitado')
        const Item = await itemsCompra.addItemCompra(payload.id_componente, payload.titulo);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-this-componente', await componentes.getComponenteCompra(payload.id_componente));
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
        console.log("item añadido");
        console.log(Item);
    } catch (err) {
        console.error('Error añadiendo item compra:', err);
    }
})
```

```
client.on('borrar-item-compra', async (payload) => {
    try {
        console.log('borrar item compra solicitado')
        const Item = await itemsCompra.deleteItemCompra(payload.id);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-this-componente', await componentes.getComponenteCompra(payload.id_componente));
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
        console.log("item borrado");
        console.log(Item);
    } catch (err) {
        console.error('Error borrando item compra:', err);
    }
})
```

```
client.on('actualizar-item-compra', async (payload) => {
    try {
        console.log('actualizar item compra solicitado')
        const Item = await itemsCompra.changeEstadoItemCompra(payload.id);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-this-componente', await componentes.getComponenteCompra(payload.id_componente));
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
        console.log("item actualizado");
        console.log(Item);
    } catch (err) {
        console.error('Error actualizando item compra:', err);
    }
})
```

4.3.10 Flujo de actualización en tiempo real:

Para mantener los datos en tiempo real entre todos los usuarios, cada vez que se realiza una operación que modifica los datos, el servidor envía una actualización a todos los clientes relevantes con los nuevos datos.

Esta arquitectura de back-end proporciona una manera efectiva y eficiente de gestionar operaciones en tiempo real, manteniendo a todos los usuarios sincronizados con los datos más recientes.

4.3.11 Actualización de la pantalla de viaje

Todas las pantallas de la aplicación contienen un método llamado initState cuya función es obtener la información actualizada del servidor y una vez recibida comenzar la ejecución del resto del código para mostrar esa información en la pantalla.

Lo primero que hace esta función es obtener una instancia del servicio SocketService que como ya se ha explicado, ademas de manejar la conexión con el serivodro, contiene la información necesaria para realizar las peticiones a este, además y su nombre y estado para mostrarlos en la parte superior de la pantalla.

A través de SocketService enviamos un evento “solicitud-componentes-viaje” junto al id del viaje en el que nos encontramos, que como mencionamos antes quedo guardado en el SocketService antes de enviar al usuario a esta pantalla.

Esta solicitud llega al servidor NodeJs que llama a getComponentes, una función que llamara a las funciones necesarias según el tipo de componente, funciones que ejecutan querys para obtener información de la base de datos, aplican la lógica que sea necesaria para cada caso (por ejemplo contar la cantidad de tareas marcadas como completadas) y crea un objeto Componente con esa información.

Un objeto componente sigue la siguiente estructura:

```
Componente {
   id: 'f97b50b0-7330-4595-9490-465aea11c4c4',
   tipo: 'habitaciones',
   color: 2,
   subcomponente: [ [Habitacion] , [Habitacion] , [Habitacion] ],
   indice: 1,
   nombre: 'Prueba',
   propiedad_1: 0,
   propiedad_2: 0
 }
```

Donde subcomponente es una lista de los elementos del componente. Un componente puede ser una lista de deudas, una lista de tareas, de eventos …

Los elementos de la lista subcomponente son muy distintos entre tipos, por ejemplo una habitación además de un nombre y un id tiene una lista de objetos cama que a su vez tienen un nombre un id y una lista de asignaciones que indica que usuarios ocupan esa cama;

Aunque las pantallas de los distintos componentes sean muy diferentes, todos llegan a la aplicación como un objeto de tipo componente. Para casos particulares en los que necesitamos más información como por ejemplo en un componente calendario que es una lista de eventos que tienen asociados unas fechas y usuarios, pero que también tiene una fecha inicial y final propia del componente, tenemos en cada objeto componente dos propiedades propiedad_1 y propiedad_2 que no tienen un tipo asociado y que se utilizan para pasar mas información a los componentes que lo necesiten ya sea una cadena de caracteres un numero o una fecha entre otros. En caso de no recibir un valor especifico estas propiedades se inicializan con valor 0.

A continuación la lista de objetos componente es guardada en la variable listaComponentes y enviada al mismo cliente que envio la solicitud, esta vez con el nombre ”lista-componentes” y la lista como payload

```dart
void initState() {
  final socketService = Provider.of<SocketService>(context, listen: false);
  socketService.socket.emit('solicitud-componentes-viaje', socketService.viaje.id);
  socketService.socket.on('lista-componentes',(payload) => {
            if (mounted)
              {
                setState(() {
                  _componentes = List.empty();
                  _componentes = (payload as List)
                      .map((componente) => Componente.fromMap(componente))
                      .toList();
                })
              }
          });
  super.initState();
}
```

```javascript
client.on('solicitud-componentes-viaje', async (payload) => {
    try {
        client.join(payload);
        console.log('usuario añadido a viaje' + payload)
        console.log('obtener usuarios de viaje solicitado')
        const UsuariosViaje = await usuarios.getUsuariosViaje(payload);
        console.log('componentes para viaje ' + payload + ' solicitados')
        const listaComponentes = await componentes.getComponentes(payload);
        console.log("usuarios: " + UsuariosViaje)
        console.log("componentes: " + listaComponentes)
        client.emit('lista-usuarios', UsuariosViaje);
        client.emit('lista-componentes', listaComponentes);
        console.log("usuarios de viaje enviados");
        console.log("componentes de viaje enviados");
    } catch (err) {
        console.error('Error enviando la lista de usuarios o componentes:', err);
    }
})
```

Cuando la aplicación recibe el mensaje resetea la variable _componentes (lista-componentes se utiliza para actualizar la lista por lo que en algunas ocasiones recibimos la lista para actualizar la actual por lo que hay que vaciarla primero) y mapea el payload para covertirlo en un objeto Componente en dart.

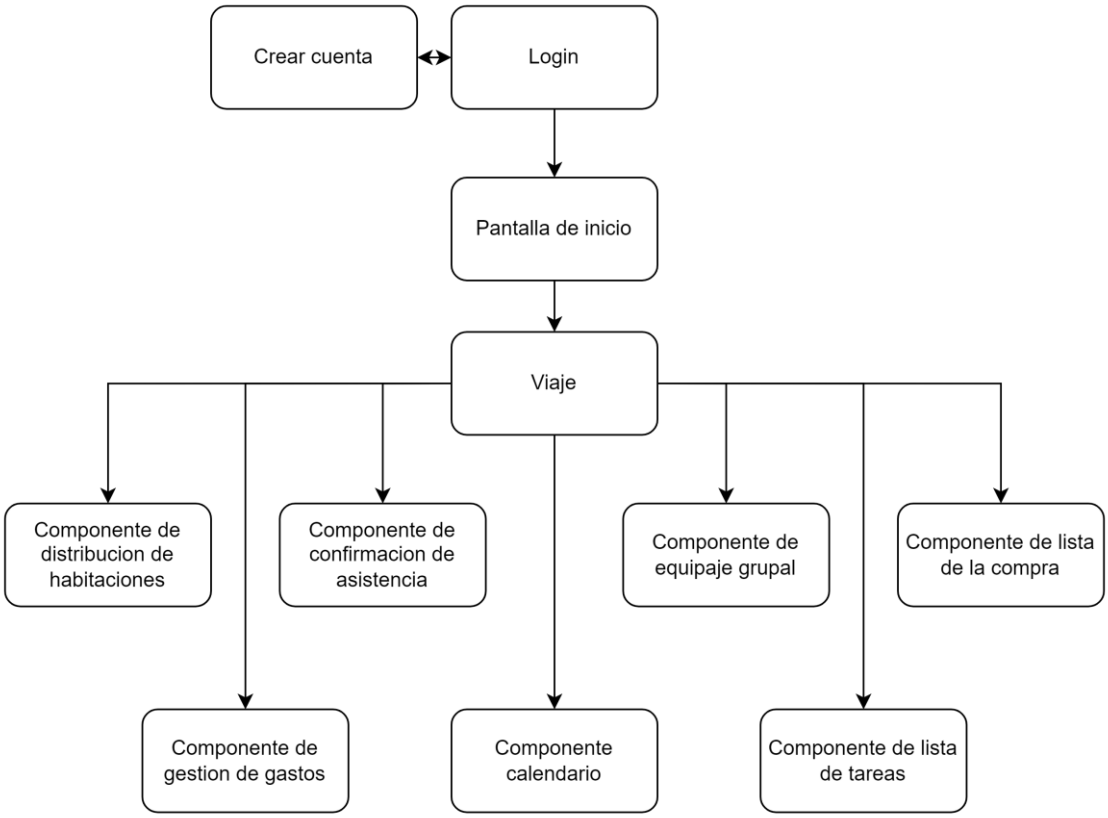
```javascript
client.on('añadir-item-compra', async (payload) => {
    try {
        console.log('añadir item compra solicitado')
        const Item = await itemsCompra.addItemCompra(payload.id_componente, payload.titulo);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-this-componente', await componentes.getComponenteCompra(payload.id_componente));
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
        console.log("item añadido");
        console.log(Item);
    } catch (err) {
        console.error('Error añadiendo item compra:', err);
    }
})
```

```javascript
client.on('borrar-item-compra', async (payload) => {
    try {
        console.log('borrar item compra solicitado')
        const Item = await itemsCompra.deleteItemCompra(payload.id);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-this-componente', await componentes.getComponenteCompra(payload.id_componente));
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
        console.log("item borrado");
        console.log(Item);
    } catch (err) {
        console.error('Error borrando item compra:', err);
    }
})
```

```javascript
client.on('actualizar-item-compra', async (payload) => {
    try {
        console.log('actualizar item compra solicitado')
        const Item = await itemsCompra.changeEstadoItemCompra(payload.id);
        const listaComponentes = await componentes.getComponentes(payload.viaje_id);
        io.to(payload.viaje_id).emit('envio-this-componente', await componentes.getComponenteCompra(payload.id_componente));
        io.to(payload.viaje_id).emit('lista-componentes', listaComponentes);
        console.log("item actualizado");
        console.log(Item);
    } catch (err) {
        console.error('Error actualizando item compra:', err);
    }
})
```

4.4 Aplicación Flutter

4.4.1 Flujo de la aplicación



La estructura de la aplicación es sencilla, el usuario abre la aplicación por la pantalla de Login, desde ahí puede o dirigirse a la pantalla de creación de cuenta que después de crear la cuenta lo devolverá a la primera o puede iniciar sesión y llegar a la pantalla de inicio. Desde la pantalla de inicio tiene acceso a la pantalla de viaje de todos los viajes a los que este invitado, además, el usuario puede crear nuevos viajes de se esta misma pantalla con un pop up que veremos más adelante.

Desde la pantalla de viaje el usuario puede acceder con un solo toque a la pantalla especifica de cualquiera de los componentes del viaje que puede ser de 7 tipos según el tipo de componente. Además, al igual que con los viajes, el usuario puede crear nuevos componentes y modificar la información del viaje sin salir de la pantalla gracias al uso de pop ups.

4.4.2 Login

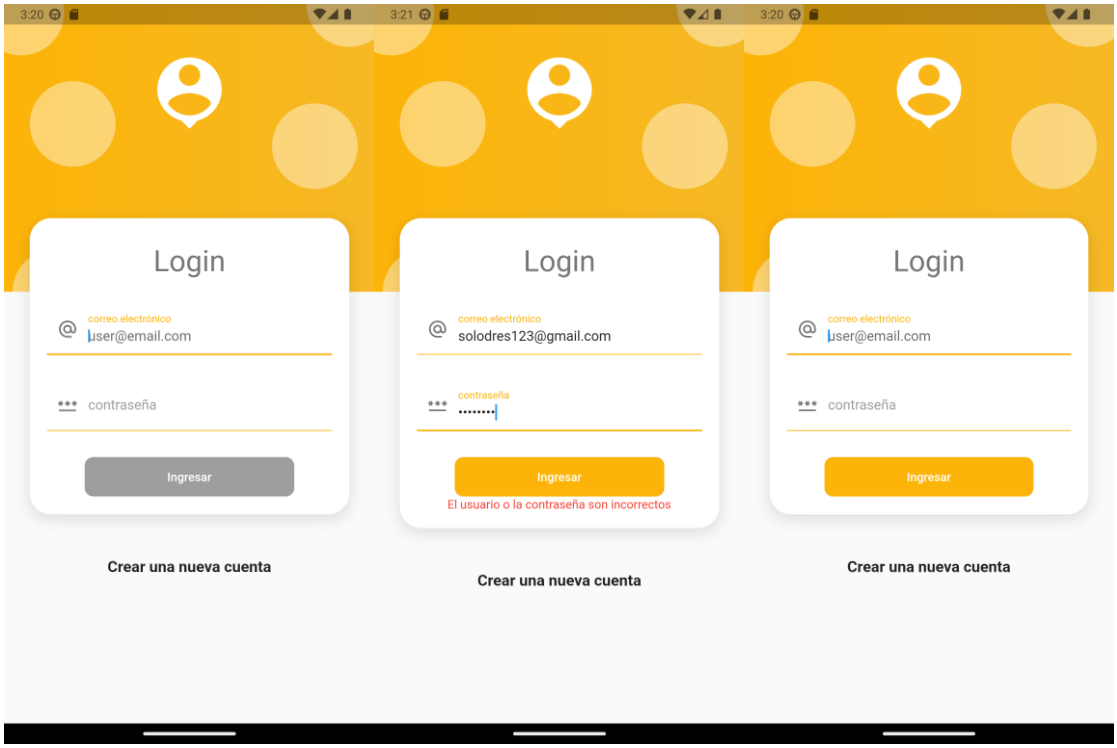
Al abrir la aplicación el usuario se encuentra con la pantalla de Login en la que debe rellenar los campos de correo y contraseña para poder iniciar sesión.

Hasta que la aplicación pueda conectarse con el servidor el botón de Ingresar se muestra de color gris y no reacciona al ser pulsado, cuando la aplicación logra conectarse se vuelve del color naranja que se muestra en las imágenes y al ser pulsado enviara al servidor el usuario y la contraseña hasheada para comprobar en la base de datos si la información es correcta. Si la información

enviada es correcta llevará al usuario a la pantalla de inicio y en caso de error en la contraseña/correo se muestra al usuario un mensaje en rojo informándole de que ha introducido mal sus datos.

Además, el campo de correo marcara un mensaje en rojo si el correo introducido no es un correo valido es decir que su estructura no es texto + arroba + texto + texto + al menos dos letras. De igual forma la contraseña mostrara un mensaje de error si no se han introducido al menos 6 caracteres que es la longitud mínima de una contraseña en la aplicación.

Debajo del botón de Ingresar hay un texto que puede ser tocado para llevar al usuario a la pantalla de creación de cuenta.

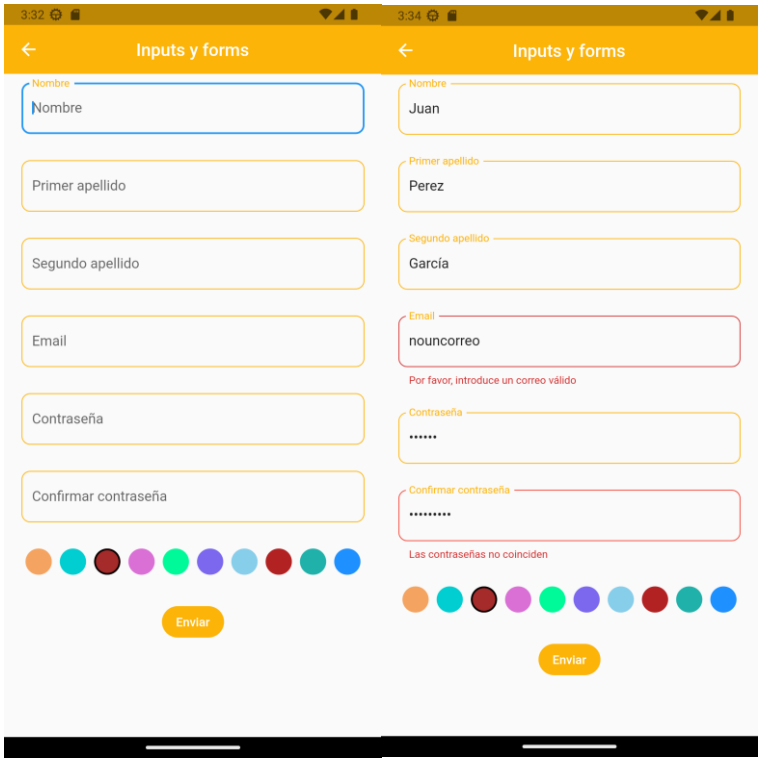


4.4.3 Pantalla de creación de cuenta

La pantalla de creación de cuentas muestra una serie de campos de texto a rellenar por el usuario seguidos de una lista de colores entre los cuales el usuario debe elegir uno que será el color de su perfil.

Los campos de nombre, primer apellido y segundo apellido muestran el teclado con la primera letra en mayúsculas para facilitar la entrada de datos, el campo de email abre el teclado en minúsculas y muestra también la fila de números en la parte superior de este y el arroba en la parte inferior junto a la tecla de espacio.

Al pulsar el botón “Crear” el usuario es llevado de nuevo a la pantalla de Login para que pueda introducir sus datos e iniciar sesión.

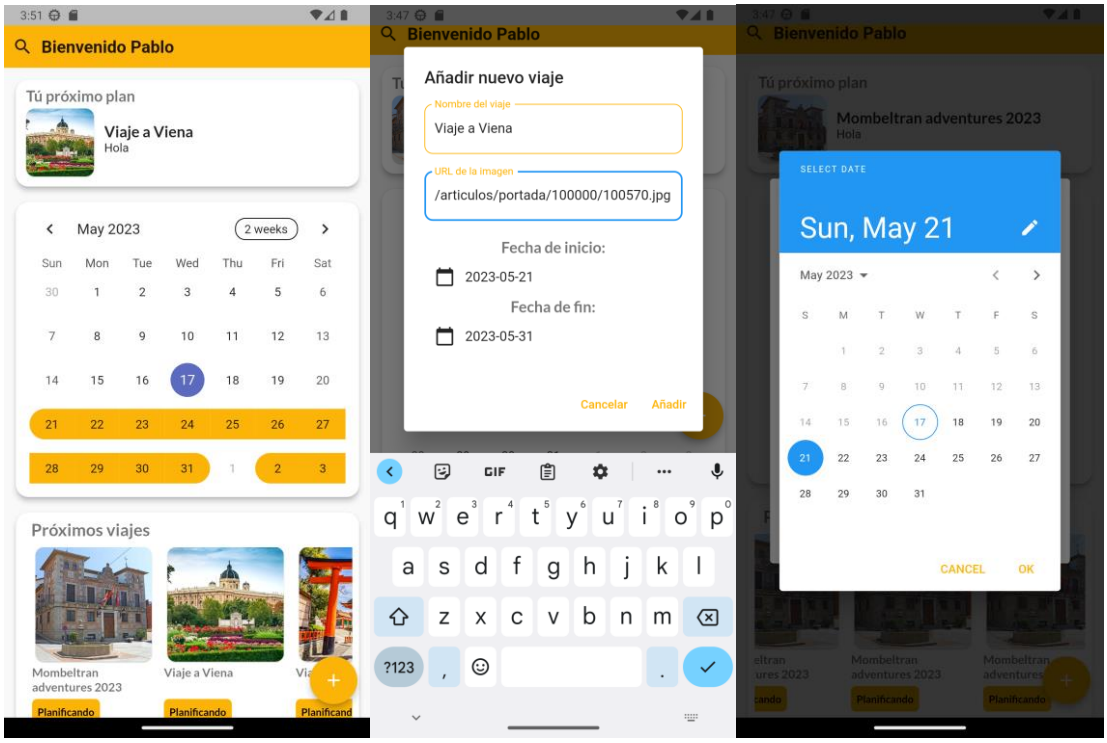


4.4.4 Pantalla de inicio

Después de iniciar sesión los usuarios son enviados a la pantalla de inicio. Esta pantalla muestra al usuario una lista de todos sus viajes y le permite crear viajes nuevos pulsando el botón flotante de la esquina inferior derecha y dando un nombre para el viaje, una URL a una imagen y un rango de fechas. Al crear el viaje este se añadirá al momento a la pantalla de inicio.

Cuando un usuario es invitado a un viaje se envía a través del servidor un mensaje al usuario invitado que manda a la aplicación refrescar su información por lo que los cambios también se ven reflejados al instante.

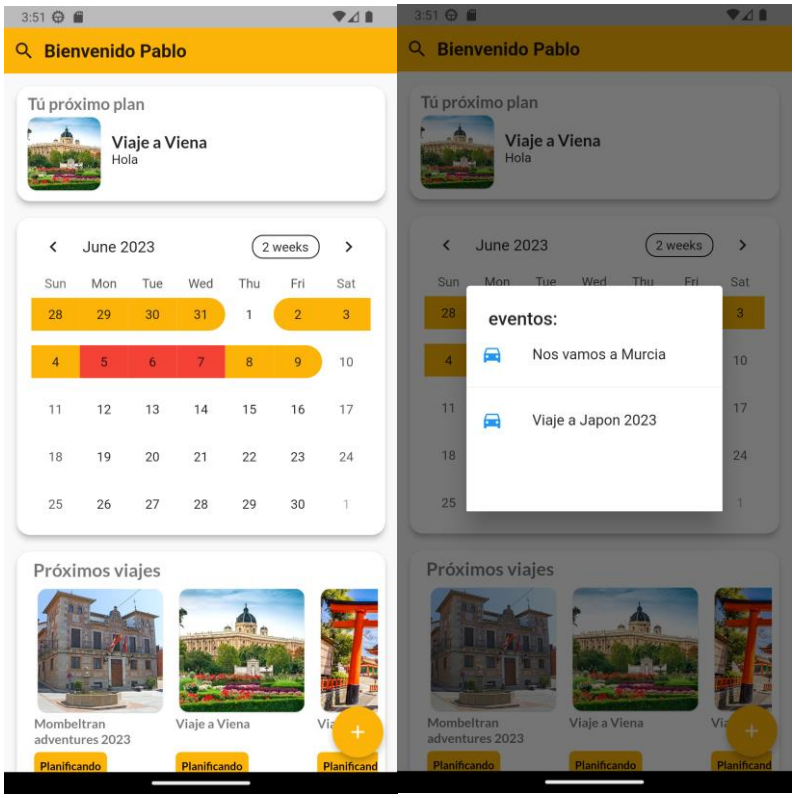
Los nuevos viajes se crean sin ningún componente, con el estado “Planificando” y con el usuario que lo ha creado como único invitado al viaje.



4.4.4.1 Calendario

El calendario de la pantalla de inicio muestra los rangos de fechas de todos los viajes en los que participa el usuario, además si se da el caso de que el usuario participe en varios viajes que se solapen, el calendario lo indicara marcando los días solapados en rojo para que el usuario sepa que algo va mal. Tocar un día marcado en naranja lleva al usuario a la pantalla del viaje asociado a esa fecha, tocar un día marcado en rojo abrirá un pop-up con una lista de tiles, cada una con el nombre de uno de los viajes que ocupan ese día. Tocar cualquiera de esos tiles lleva a usuario al viaje correspondiente para que pueda gestionar el problema.

El calendario se abre empezando por el mes actual, se puede retroceder hasta el mismo mes del año anterior y avanzar hasta el mismo mes de dentro de 3 años.



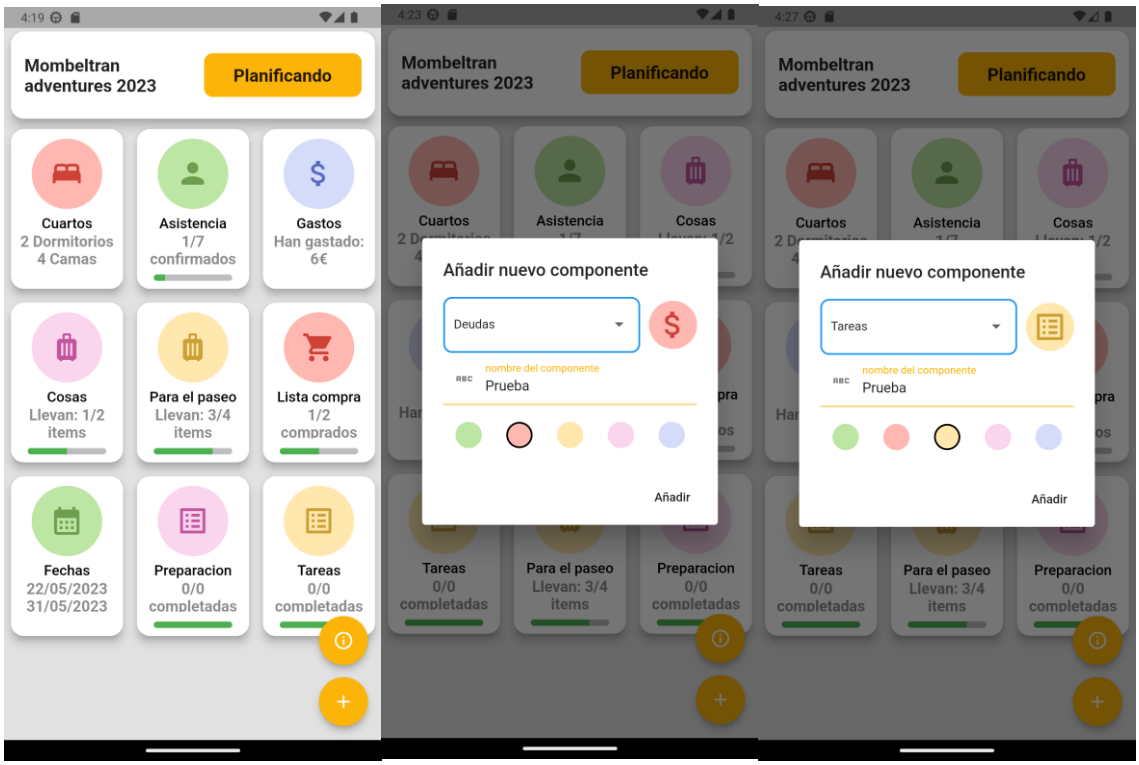
4.4.4.2 Listas de viajes

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

4.4.5 Pantalla de viaje

A un viaje podemos añadir 7 tipos de componente, más adelante explicaremos cada uno de ellos en profundidad. Para crear un nuevo componente el usuario

pulsar el botón con el símbolo “+” de la esquina inferior derecha, lo que abrirá un pop up en el que debe seleccionar de la lista desplegable el tipo de componente que quiere crear, dar un nombre para el componente y elegir de la lista de colores el que mejor le parezca. Seleccionar el tipo de componente y el color cambian en tiempo real la vista previa del icono del componente que se encuentra a la derecha del desplegable de tipos.



4.4.5.1 Pop Up de información del viaje

4.4.5.2 Lista de componentes

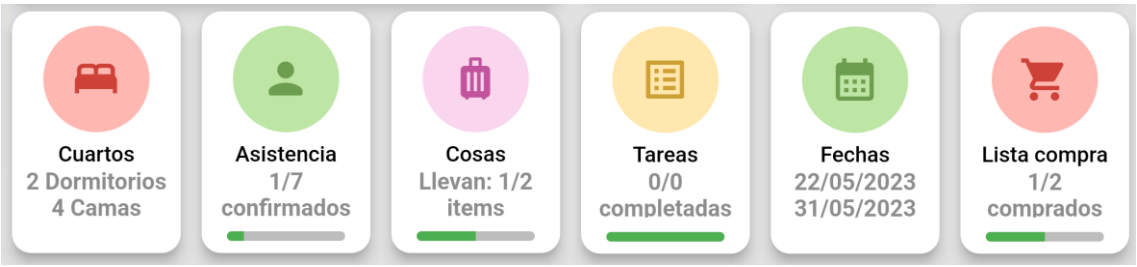
Las tarjetas de componentes no son solo un vinculo entre la pantalla de cada componente y la pantalla de viaje, estan diseñadas para mostrar de forma resumida y clara la información más relevante de su componente, lo que

convierte la lista de tarjetas en un panel de control en el que los usuarios pueden ver facilmente el estado de la planificacion del viaje sin tener que abrir cada uno de los componentes que contiene.

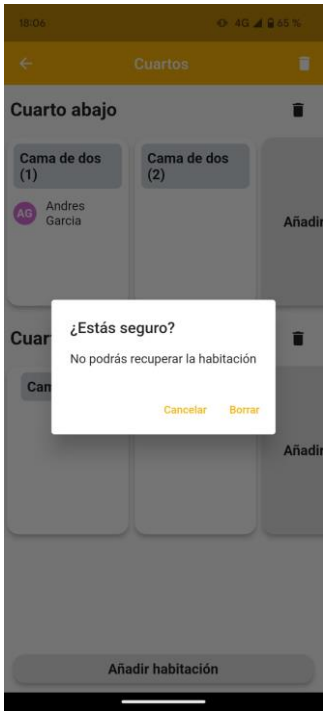
Cada tipo de componente tiene asociado un tipo de tarjeta diseñado para mostrar de la mejor forma posible la informacion más relevante. Por ejemplo el componente de fechas

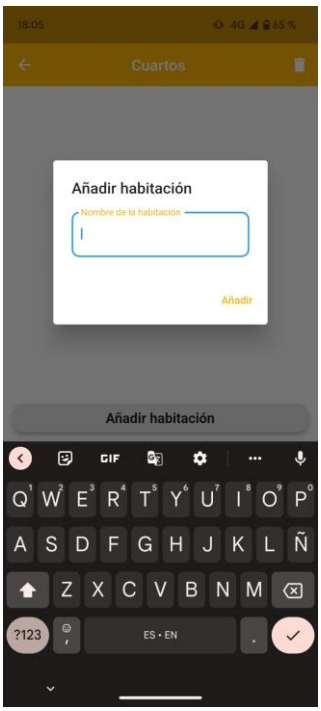
Las tarjetas de componentes no son solo un boton para dirigirse desde la pantalla de viaje hasta la pantalla del componente que representan, cada tarjeta representa una vista previa de la información mas relevante del componente.

El objetivo de las cards es funcionar a modo de panel de control de forma que un usario pueda ver



4.4.6 Componente de distribución de habitaciones

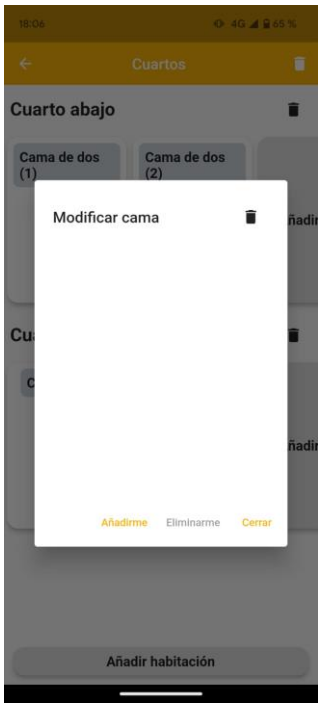




de cuartos y camas del lugar en el que se hospedaran durante el viaje. Al ver de forma sencilla la distribución de camas es más fácil planificar quien ocupara cada habitación, en las páginas de alquiler para viajes como Airbnb es muy

común que solo se anuncie el numero de camas de la casa y los usuarios tienen que orientarse con las fotos para saber como están repartidas por la casa.

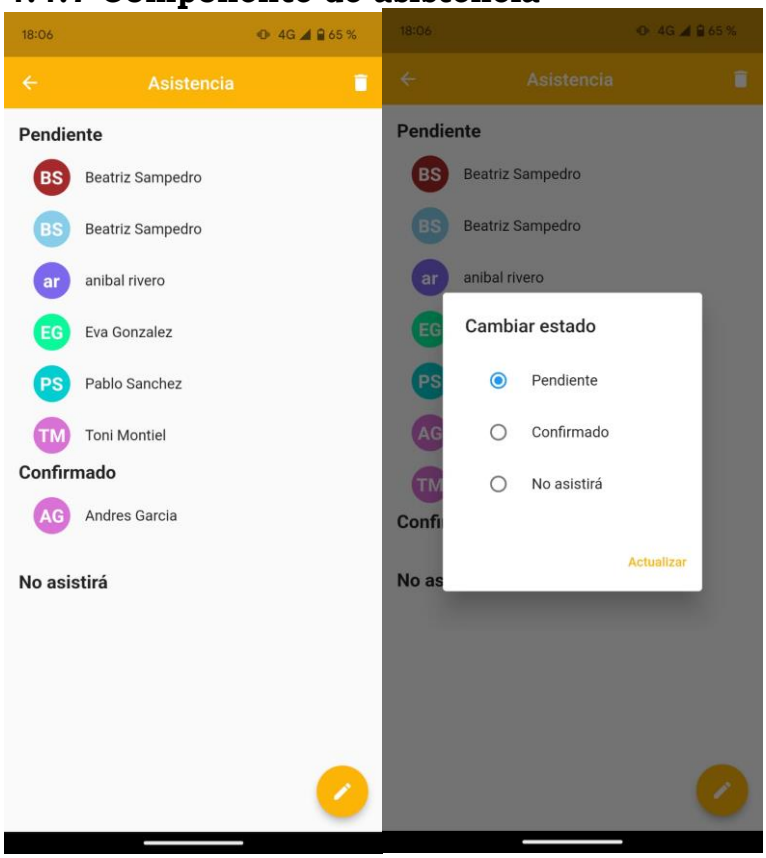
En el momento en el que



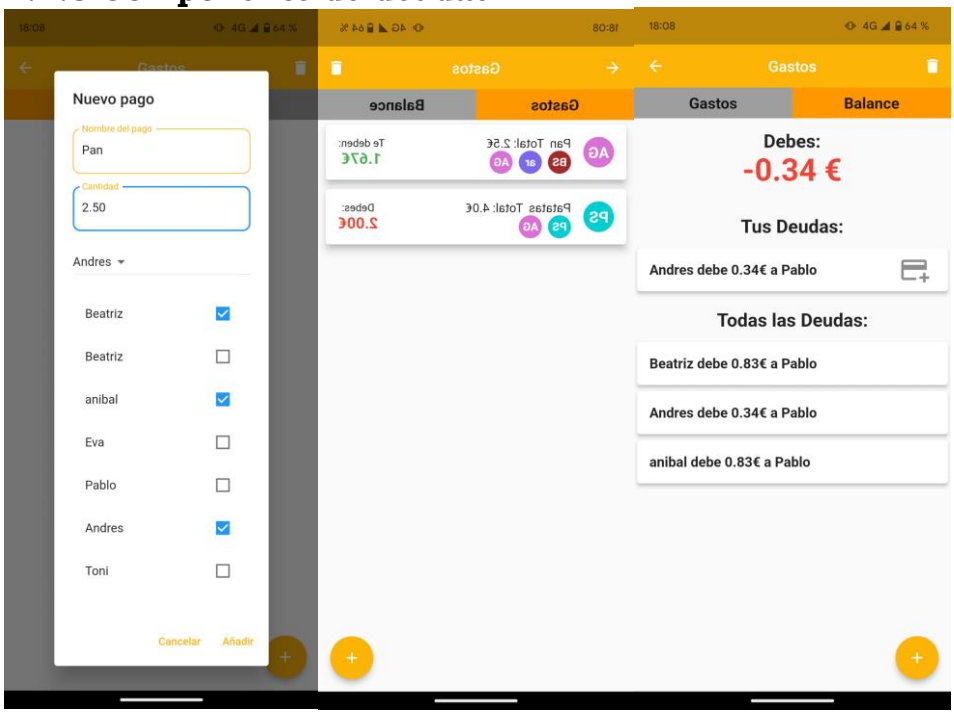
.

.

4.4.7 Componente de asistencia



4.4.8 Componente de deudas

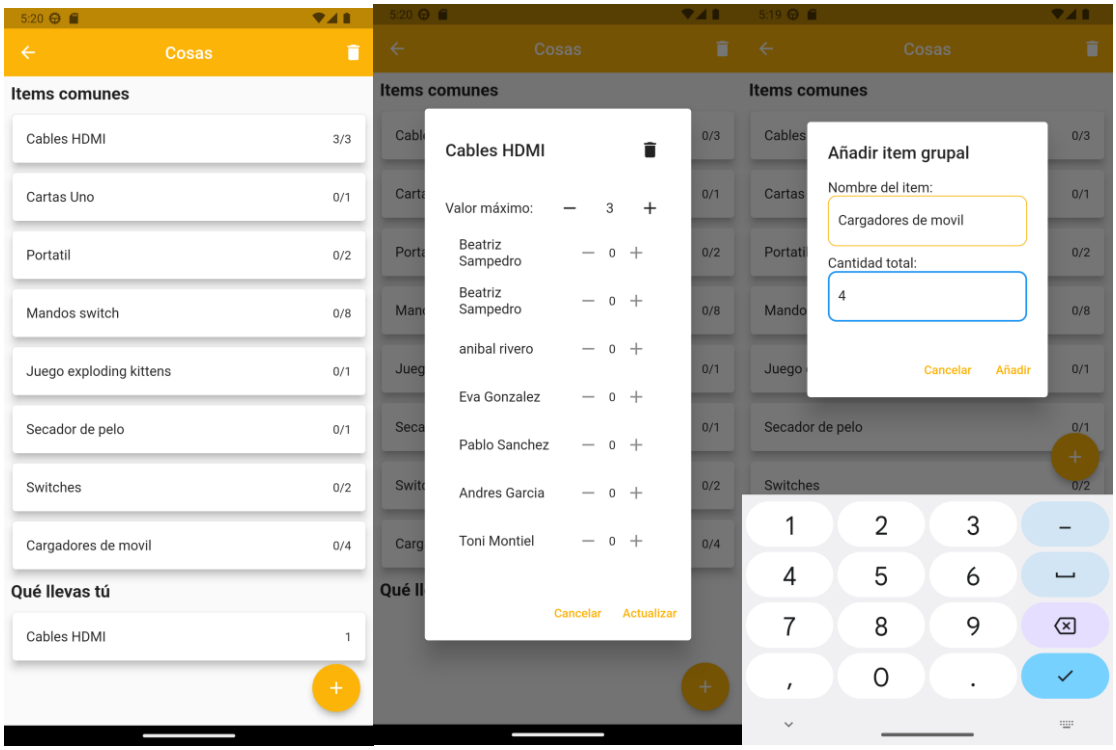


4.4.9 Componente de equipaje grupal

En el componente de equipaje grupal cualquier usuario invitado al viaje puede añadir con el botón de la esquina inferior derecha un nuevo item grupal. Pulsar este botón abrirá un pop up que pedirá al usuario un nombre y una cantidad para el nuevo item. Al abrir el pop up se despliega automáticamente el teclado con la primera tecla en mayúscula y apuntando al campo nombre, al tocar en el campo cantidad el teclado pasa a ser numérico.

Cuando el usuario pulsa en añadir el item se añade al instante a la pantalla en la lista de ítems comunes. Los usuarios pueden tocar cualquiera de estos ítems para desplegar un pop up en el que podrán modificar la cantidad requerida de dicho item y la cantidad del mismo encargado a cada usuario invitado al viaje pulsando los botones de + y – situados junto al nombre de dicho usuario.

Debajo de la lista de ítems comunes tenemos una lista llamada “Que llevas tú” que muestra los ítems en los que la asignación del usuario es mayor a 0 ademas de la cantidad que tiene que llevar el usuario que esta usando la aplicación. Esto permite al usuario ver fácilmente lo que debe llevar al viaje sin tener que revisar cada item de forma individual.



4.4.10 Componente de tareas

4.4.11 Componente de compra

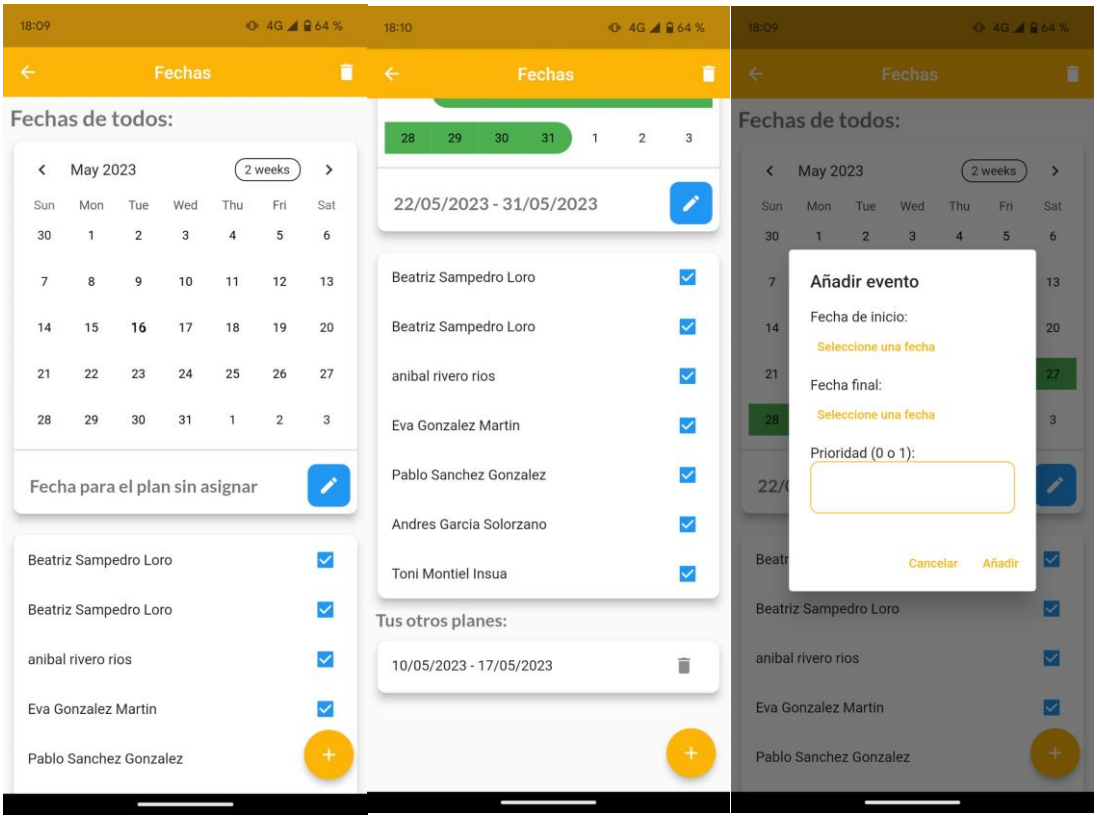
4.4.12 Componente de fechas

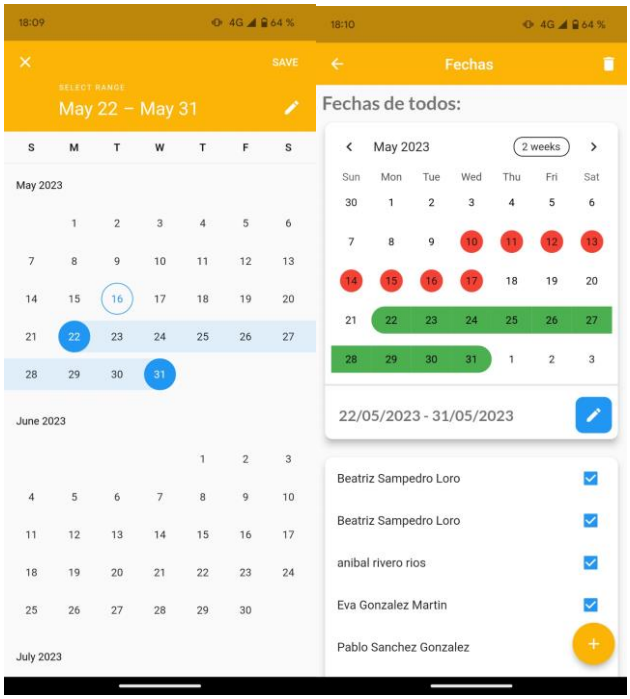
Este componente es uno de los más complejos y al mismo tiempo de los más útiles. Cuando el usuario abre por primera vez un componente de tipo calendario se encontrara con 3 cosas.

Calendario y lista de usuarios:

En el calendario del componente fechas se verán reflejadas dos cosas, veremos en verde el rango de fechas seleccionado para el viaje, que puede ser creado o actualizado pulsando el botón azul en la parte inferior del calendario.

Además podremos ver también las fechas en las que cualquiera de los invitados al viaje no podría asistir. Estas fechas pueden ser añadidas por cada usuario pulsando el botón de la parte inferior derecha de la pantalla que abre un pop up preguntando por la fecha inicial y final del rango en que no podrán acudir y un grado de importancia, pueden marcar Posible o Imposible para dar mas flexibilidad a la planificación. Las fechas marcadas como Posible se muestran de color naranja y las Imposible de color Rojo.





(no sé porque me mete espacios, pero no ha sido manualmente)

(los componentes además de ser una tarjeta pequeña dentro de la lista son también una pantalla completa a la que se accede al tocar esas tarjetas por lo que están a la misma altura que el resto de las pantallas, estas aclaraciones no estarán escritas en la entrega final pero como aun no puede verse el contenido del índice quería aclararlo)

5 Evaluación y futuros cambios

6 Conclusiones

7 Análisis del impacto

8 Futuro

9 Bibliografía