



UAX

UNIVERSIDAD ALFONSO X EL SABIO

Aplicación móvil

“FlyApp”

TRABAJO FINAL DE GRADO

Autor/a: Álvaro Lozoya Llorente

Tutor/a: Germán Fernández Martínez Fecha: 2023

Índice:

1. ABSTRACT	1
2. RESUMEN	1
3. OBJETIVOS Y MOTIVACIÓN	3
4. ANTECEDENTES	3
5. ANÁLISIS Y ESPECIFICACIÓN DE REQUISITOS	7
5.1.1 TECNOLOGÍAS	7
5.1.2 HERRAMIENTAS	8
5.1.3 REQUISITOS	9
5.2 FUNCIONALIDADES CLAVE	9
5.3 CASOS DE USO	11
5.4 DIAGRAMA DE FLUJO	12
6. PROPUESTA DE SOLUCIÓN Y DISEÑO	13
6.1 PROPUESTA DE SOLUCIÓN	13
6.2 DISEÑO DE SOLUCIÓN DE PROPUESTA	23
7. Plan de trabajo	38
8. DESARROLLO DE LA SOLUCIÓN	40
8.1 APIs UTILIZADAS	40
9. DESPLIEGUE E INSTALACIÓN	41
9.1 CREACIÓN DEL PROYECTO	41
10. Problemas encontrados	46
11. EVOLUCIÓN Y TRABAJO FUTURO	48
12. BIBLIOGRAFÍA	50

1. ABSTRACT

FlyApp is a nationwide flight management application, where both workers and users can use it in a very comfortable and intuitive way.

Workers will be able to use **FlyApp** to manage flight tickets without any kind of inconvenience or cumbersome equipment, being able to perform all the functions of their work from the application itself.

Users will be able to enjoy a seamless application where they will be able to purchase flight tickets in different ways such as "Paypal, VISA, MasterCard..." or the **FlyApp** wallet itself which can be recharged with the same payment gateways to ensure the user's convenience when purchasing a flight.

FlyApp will have support channels (for both users and employees) to try to solve any problems that may occur.

2. RESUMEN

FlyApp es una aplicación de gestión de vuelos a nivel nacional, donde tanto trabajadores como usuarios podrán usarla de una forma muy cómoda e intuitiva.

Los trabajadores podrán usar **FlyApp** para gestionar la entrada a los vuelos sin ningún tipo de inconveniente o equipo aparatoso, pudiendo realizar todas las funciones de su trabajo desde la propia aplicación.

Los usuarios podrán disfrutar de una fluida aplicación donde se les permitirá adquirir billetes de vuelos de diferentes formas ya sea “Paypal, VISA, MasterCard...” o la propia billetera de **FlyApp** la cual se puede recargar con los mismos portales de pago para asegurar la comodidad del usuario a la hora de adquirir un vuelo.

FlyApp contará con canales de soporte (tanto para usuarios como trabajadores) para intentar solucionar cualquier problema que pueda ocurrir.

Una vez hecho el resumen de la aplicación paso a explicar brevemente todas las funcionalidades que presenta la aplicación y como ha sido su evolución a lo largo del desarrollo.

Las principales funcionalidades de la aplicación son: Iniciar sesión con multiusuario, es decir, iniciar sesión con diferentes cuentas (solo una a la vez), cerrar sesión, mostrar vuelos reales, calcular un precio con la distancia entre aeropuertos, comprar un billete para un vuelo, generar un código **QR** y validarla, guardar en la base de datos todos los vuelos, usuarios y billetes.

Según fué avanzando el desarrollo de esta aplicación muchas de las funcionalidades evolucionaron para obtener un mejor funcionamiento/rendimiento o simplemente porque necesitaban evolucionar por un erróneo planteamiento inicial, por ejemplo la funcionalidad de comprar un billete, evolucionó a poder comprar varios billetes para un mismo vuelo desde la misma cuenta, ya que si hay varios pasajeros estos tendrían que comprar el billete desde una cuenta propia, y en caso de una familia por ejemplo no sería coherente.

O la funcionalidad de mostrar los vuelos evolucionó finalmente a un buscador entre un origen y un destino que muestra los vuelos entre esos dos aeropuertos, ya que es innecesario estar viendo todos los vuelos que hay disponibles si únicamente quieres viajar de Madrid a Barcelona.

Otra funcionalidad que evolucionó fue el registro de usuarios y sus datos, juntando todo en el registro para no tener que pedir los datos de la persona cada vez que quiera comprar un billete para el/ella mismo/a.

De todas formas más adelante entraré en más detalle sobre todos los cambios y problemas que he tenido a lo largo del desarrollo.

3. Objetivos y motivación

El objetivo principal de **FlyApp** es hacer la compra y comparación de vuelos más sencilla y accesible de lo que ya es, este mismo objetivo también es la motivación para realizar esta aplicación.

Los demás objetivos de **FlyApp** son los más comunes entre cualquier aplicación/página de compra de vuelos/billetes:

- Ser la aplicación nº1 en la compra de billetes
- Cuidar al usuario de una forma muy especial con una aplicación muy intuitiva y fácil de usar
- Tener unos precios asequibles para todo el mundo, rebajando en la medida de lo posible el valor de dichos billetes

En definitiva, como se ha comentado anteriormente la motivación de **FlyApp** es esforzarse en cumplir con todos los objetivos de la mejor forma posible.

4. Antecedentes

En el caso de los antecedentes podemos encontrar miles de propuestas que realizan la misma función o funciones similares que **FlyApp** ya que hay miles de aplicaciones y páginas que ofrecen compra de billetes, ya sea para vuelos, trenes o autobuses. En esencia todo es lo mismo. Aunque cada una tiene sus diferencias y cualidades.

Algunos ejemplos de estas aplicaciones son las siguientes:

- Renfe: Una aplicación de una empresa muy conocida en España donde puedes comprar billetes de tren para viajar por el país.



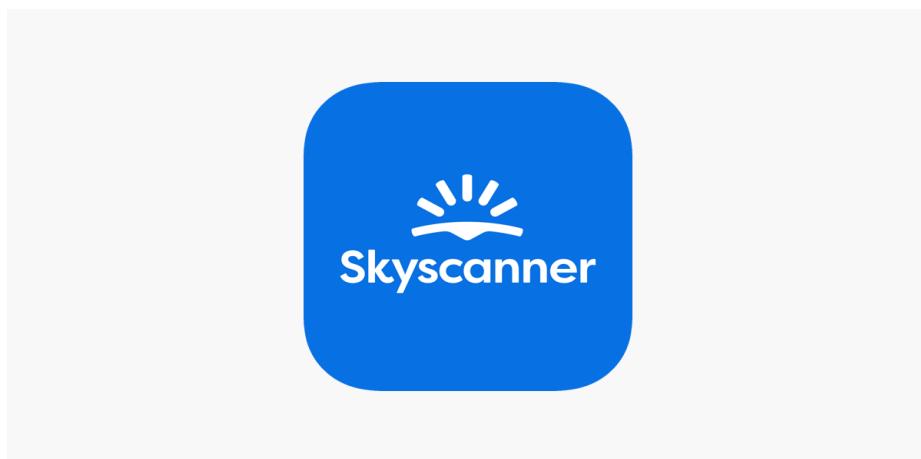
- Avanza: Una aplicación de una empresa de transporte con autobuses que permite la compra de billetes de sus autobuses.



- Iberia: Una aplicación de una empresa muy conocida de compra de billetes de avión para viajar por todo el mundo.



- Skyscanner: Una aplicación que ofrece la posibilidad de comparar y comprar billetes de vuelos principalmente, aunque ofrecen más servicios como hacer reservas en hoteles, realizar alquiler de coches, etc.



A continuación vamos a ver una tabla comparativa entre estas aplicaciones y FlyApp:

Funciones	Renfe	Avanza	Iberia	SkyScanner	FlyApp
Ver y comparar billetes	✓	✓	✓	✓	✓
Comprar billetes	✓	✓	✓	✗	✓
Cartera virtual	✗	✗	✗	✗	✓
Mapa a tiempo real	✗	✓	✗	✗	✓
Selección de asiento	✓	✗	✓	✗	✓

fuente(creación propia)

Como se puede observar en esta tabla se han comparado las funciones de:

- Ver y comparar billetes por su precio, horario etc.
- Comprar billetes, teniendo una cartera virtual para hacerlo, a parte de ofrecer las opciones tradicionales de pago.
- Visualización de un mapa a tiempo real para ver dónde está cada vehículo
- La opción de elegir asiento.

Los resultados de estas funciones son bastante variados entre sí, siendo la más destacada de todas la función de comprar los billetes con la opción de la cartera virtual, ya que todas ofrecen las vías tradicionales de pago, pero no una propia.

En el caso de SkyScanner cabe destacar que no ofrece la compra directa de los billetes, si no que ofrece un atajo / redirección a la página de cada aerolínea o servicio donde quieras comprar.

5. Análisis y especificación de requisitos

A continuación indico los requisitos y tecnologías necesarias para que el proyecto funcione sin problemas:

5.1.1 Tecnologías

Las tecnologías utilizadas son principalmente:

- **Node.js**: La tecnología más importante utilizada ha sido **Node.js**, esta es fundamental para que pueda funcionar react native a la hora de desarrollar. Esta tecnología la usamos con el comando **npm**, fundamental para instalar todo tipo de dependencias en el proyecto.
- **React Native**: Como tecnología principal tenemos **React Native**, la cual nos permite desarrollar una aplicación que puede usar todas las funciones del dispositivo móvil de forma nativa, mejorando el rendimiento frente a cualquier otra aplicación.
- **Expo**: Como tecnología secundaria tenemos **Expo**, un conjunto de herramientas que nos ayudan a configurar el proyecto de React Native de forma automática, permitiendo ahorrar tiempo a la hora de crear una aplicación.
- **NoSQL**: Para la base de datos vamos a usar una base de datos **NoSQL**, es decir, no relacional. Esto permite un funcionamiento más fluido y mucha más capacidad de almacenamiento, la herramienta que vamos a usar para esta tecnología la indico a continuación en el apartado **5.1.2 Herramientas**

5.1.2 Herramientas

Las herramientas utilizadas durante el desarrollo son las siguientes:

- **Android Studio:** Otra tecnología utilizada en este proyecto ha sido **Android Studio**, para utilizar un dispositivo virtual y poder desarrollar la aplicación de forma visual y cómoda sin necesidad de tener que usar tu propio dispositivo.
- **Expo Go:** Es una herramienta desarrollada por **Expo** para poder visualizar la aplicación en tiempo real en un dispositivo físico sin hacer uso de cables, en mi caso la he usado de forma intercalada, al igual que los sistemas operativos (Windows y Ubuntu). Un uso muy bueno que se le puede dar es por ejemplo ver cómo queda la app visualmente en distintos dispositivos a la vez, sin tener que crear una apk.
- **Visual Studio Code:** VSStudio es la herramienta que he utilizado como **IDE** “(Integrated Development Environment)” o “(Entorno de desarrollo integrado)” para implementar todo el código de la aplicación.
- **Git:** Para el control de versiones he utilizado la herramienta **Git**, en concreto **GitHub**, el cual permite llevar un control sobre las versiones y el desarrollo del proyecto, permitiendo cometer errores sin miedo a que deje de funcionar.
- **Firebase:** En este proyecto vamos a usar Firebase como “herramienta”, es decir, vamos a usar la base de datos de Firebase para gestionar tanto los usuarios, como los datos. Firebase es una base de datos **NoSQL**, es decir, una base de datos no relacional creada por Google.
- Diferentes dependencias para llevar a cabo el desarrollo, por ejemplo un efecto gradiente que hay en la aplicación, la gestión de las pantallas en toda la aplicación o la generación y lectura de códigos **QR** (más adelante explicaré con más detalle todas las dependencias y cómo usarlas).

5.1.3 Requisitos

Los requisitos para poder utilizar la aplicación son los siguientes:

- Conexión a internet: La conexión a internet es necesaria para poder realizar las llamadas necesarias a la *API* y a la base de datos.
- Versión de Android mínima (6.0): Es necesario tener como mínimo la versión de Android (6.0) ya React Native a partir de la versión (0.64) no admite versiones de Android inferiores.

5.2 FUNCIONALIDADES CLAVE

Entre las funcionalidades clave de la aplicación podemos encontrar las siguientes explicadas a continuación:

- 1-** Funcionalidad de creación y autenticación de usuarios personalizados utilizando la tecnología que nos provee Firebase Authentication.
- 2-** Funcionalidad de ver todos los vuelos y filtrarlos a través de una llamada inicial a la API que guardará la respuesta de los vuelos y permitirá a través de llamadas a la base de datos dicho filtro para encontrar el vuelo que el usuario esté buscando.
- 3-** Funcionalidad de comprar un billete permitiendo al usuario entrar dentro de un vuelo para ver sus detalles y si le interesa comprar ese vuelo, poder hacerlo introduciendo los datos de uno o varios pasajeros.
- 4-** Funcionalidad de elegir asiento en un vuelo, complementando la funcionalidad anterior pudiendo elegir dónde se sentarán los pasajeros en el avión correspondiente.

5- Funcionalidad de ver los vuelos comprados, es decir, un usuario que haya comprado uno o varios vuelos podrá verlos y consultarlos siempre que quiera. Cabe destacar que quedan separados entre “próximos vuelos” y “vuelos anteriores”, siendo visibles todos los pasajeros que haya introducido el usuario en dichos vuelos.

6- Funcionalidad de cargar la cartera con dinero virtual, a través de las plataformas de pago más famosas (Paypal, Visa, MasterCard...). Se podrá recargar la cartera para adquirir los vuelos, en caso de no tener saldo suficiente, saltará una alerta avisando de ello. (Lógicamente estas plataformas de pago serán una simulación no siendo necesario ingresar los datos que serían obligatorios en las plataformas de pago reales).

7- Funcionalidad de cambiar los datos del usuario, permitiendo así modificar cualquier dato erróneo o que haya cambiado en la vida real.

8- Funcionalidad de acceder a la aplicación con multiusuario, es decir, entrar con distintas cuentas a la app e incluso entrar a la vista de empleados si introduces una cuenta que tenga este rol.

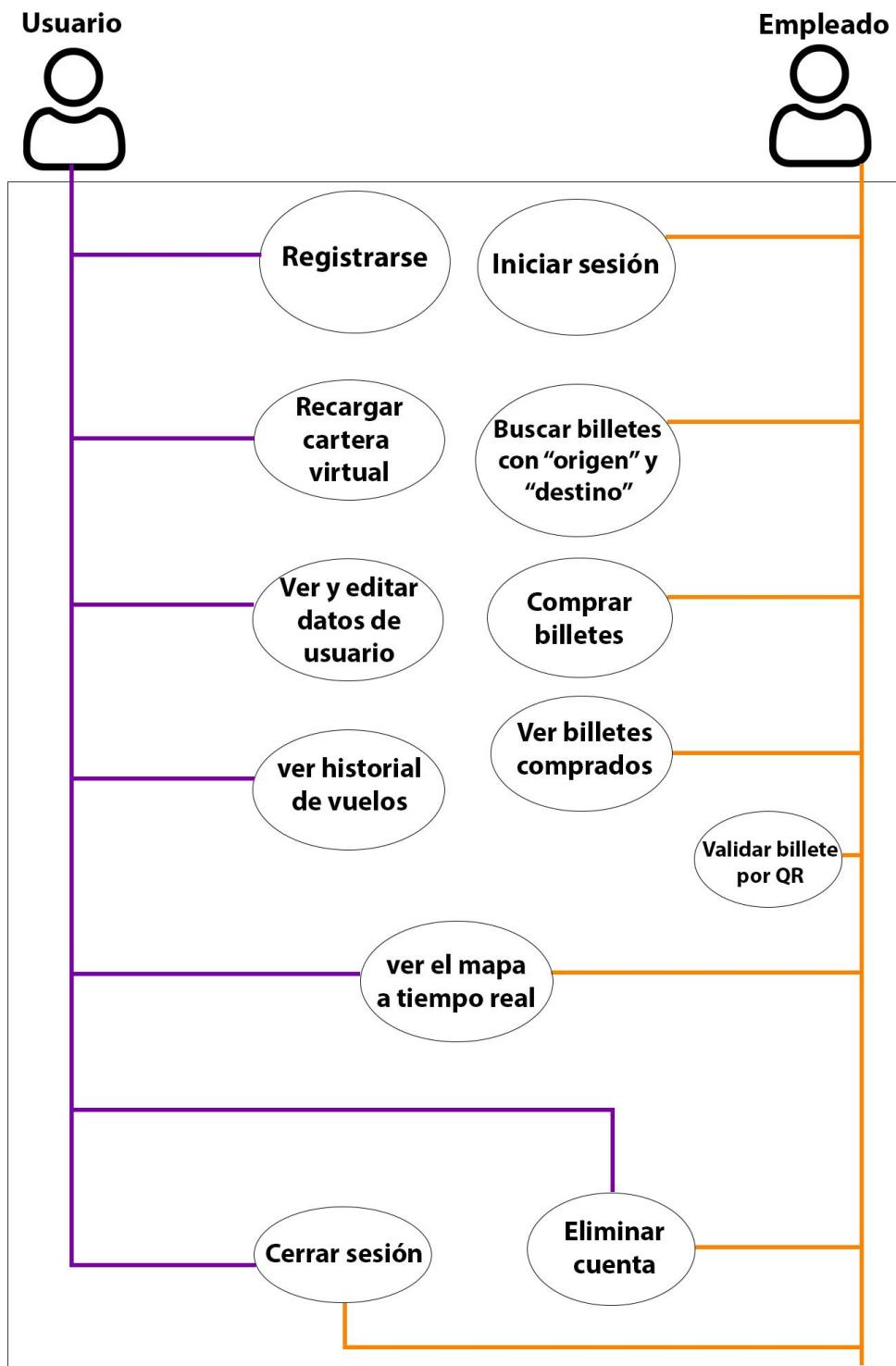
9- Funcionalidad de cerrar sesión para poder iniciar sesión con otra cuenta.

10- Funcionalidad de crear un código *QR* único para cada billete, el cual un empleado validará a la hora de hacer el *check-in*.

11- Funcionalidad de guardar en una base de datos todos los datos obtenidos de las funcionalidades anteriores y hacer uso de ellos para que toda la aplicación pueda funcionar de forma correcta.

5.3 CASOS DE USO

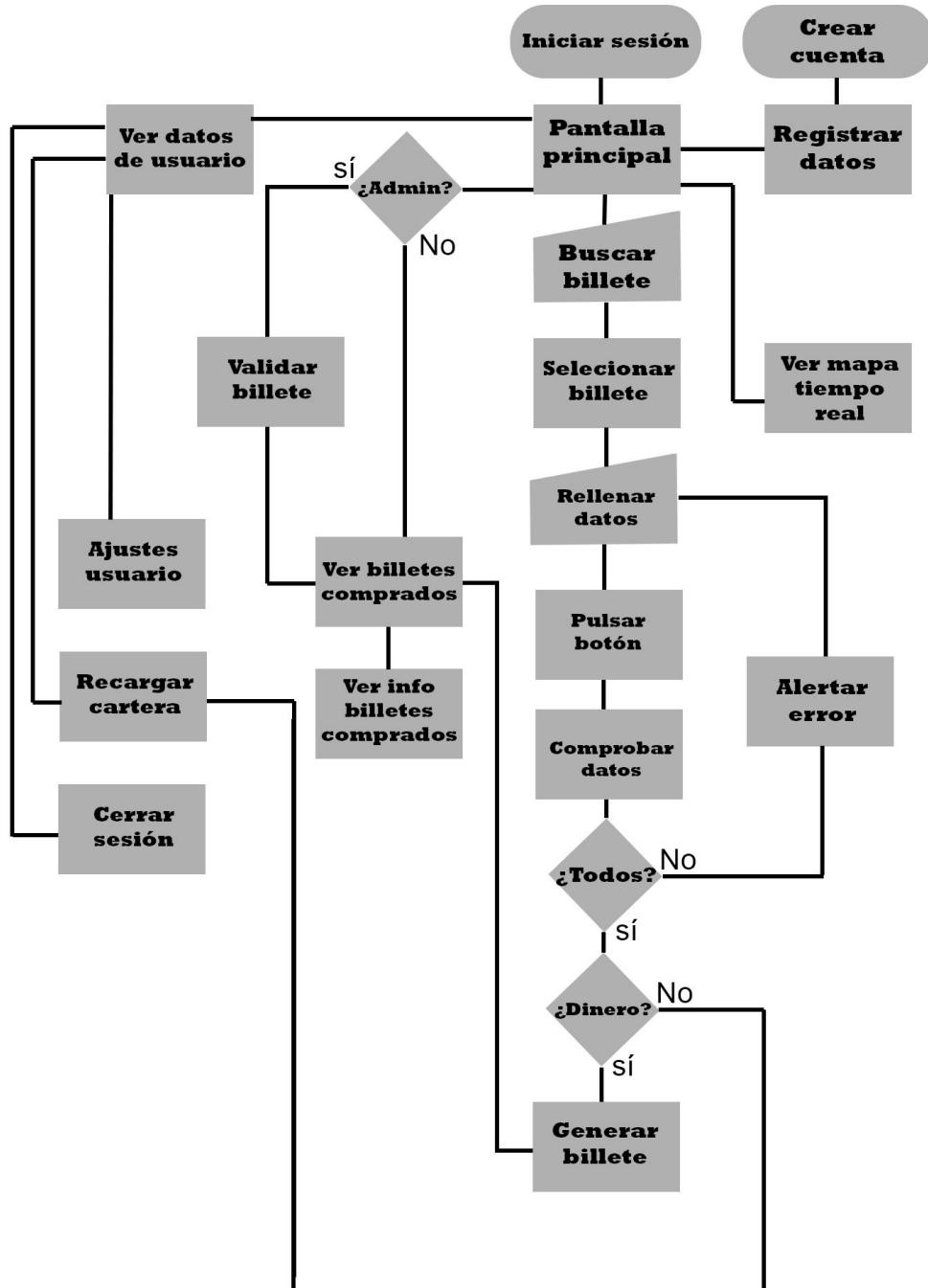
A continuación vamos a ver un diagrama de casos de uso sobre FlyApp:



fuentecreación propia)

5.4 DIAGRAMA DE FLUJO

Ahora vamos a ver un diagrama de flujo que nos explica de forma visual qué acciones se realizan internamente en cada pantalla cuando se realiza una acción, de todas formas entrará en más detalles sobre el flujo de funcionamiento de la aplicación y cómo se ha llevado a cabo más adelante en el punto **6.1 Propuesta de solución**:



fuente(creación propia)

6. Propuesta de solución y diseño

6.1 Propuesta de solución

En este apartado voy a proceder a explicar mi propuesta de la solución, para ello voy a exponer detalladamente cómo está desarrollada la aplicación en su totalidad de forma ordenada, es decir, como se vería al utilizarla:

Antes de comenzar con las pantallas y su funcionamiento interno, cabe destacar que se está utilizando la base de datos Firebase, como he mencionado anteriormente.

Para que funcione correctamente se debe tener creado un proyecto en Firebase, con todo lo que vamos a utilizar activado.

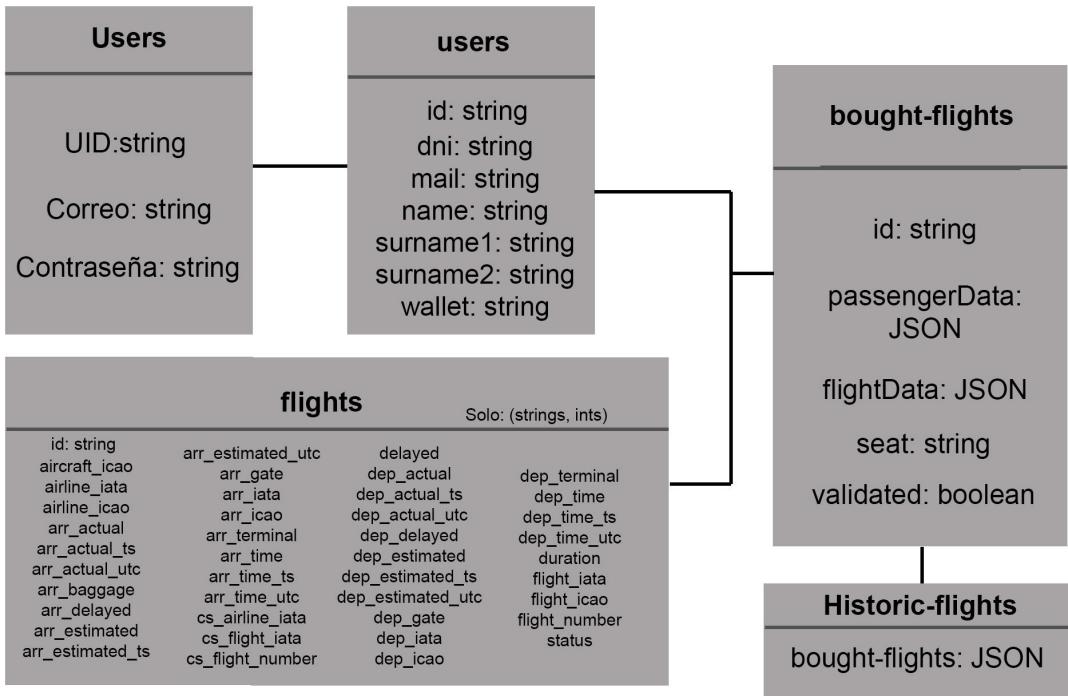
Firebase ofrece una base de datos aislada llamada **Authentication** la cual permite gestionar las cuentas de los usuarios de forma muy rápida y sencilla, englobando las funciones de registro y autenticación de usuarios. Además también se está utilizando **Firestore** para almacenar datos, la base de datos hay que tenerla inicializada y configurada en la aplicación, siguiendo la documentación que nos proporciona Firebase. Copiamos la configuración de nuestro proyecto y la colocamos en un archivo de la aplicación, el cual se especifica en dicha documentación, después debemos inicializar la base de datos y la conexión a Firebase de la siguiente forma:

```
const app = initializeApp(firebaseConfig);
const db = getFirestore(app);
```

fuente(creación propia)

Más adelante en el punto 9.1 Creación del proyecto se darán más detalles.

Es importante ver con atención el diagrama conceptual de datos para entender con más claridad cómo se “relacionan” las distintas colecciones de la base de datos. Quiero recordar que Firebase es una base de datos no relacional, por lo que la relación entre las colecciones se realiza de forma artificial a través del código:



fuente(creación propia)

También cabe destacar que todas las pantallas de la aplicación están siendo manejadas y administradas con la librería “react-navigation” la cual permite de forma muy sencilla cambiar entre pantallas, permitiendo guardar los datos de una pantalla anterior si fuese necesario o enviando esos datos a la siguiente pantalla o la pantalla donde se necesiten utilizar. Un sencillo ejemplo para ver cómo se implementaría en el código es lo siguiente:

```

<NavigationContainer>
  <Stack.Navigator
    initialRouteName='Login'
    screenOptions={{
      headerShown: false
    }}>
    <Stack.Screen name="Login" component={Login} />
  
```

fuente(creación propia)

Para todas las pantallas de la aplicación, y:

```

<Tab.Navigator
  initialRouteName='Vuelos'
  screenOptions={{
    headerShown: false,
  }}
>
  <Tab.Screen name="Vuelos" component={Flights} options={{
    tabBarIcon: ({ color, size }) => (
      <Icon name="search" color={color} size={size} />
    ),
    tabBarStyle: {
      backgroundColor: 'black',
      borderTopWidth: 0,
      elevation: 0,
    },
    tabBarActiveTintColor: 'orange',
    tabBarInactiveTintColor: 'grey',
  }}
/>

```

fuente(creación propia)

para las pantallas que van a aparecer en la barra de navegación inferior (como se puede apreciar en los diseños expuestos en el punto **6.2 Diseño solución propuesta**)

Comenzamos con la pantalla de “Register”, ésta es la pantalla en la que podremos registrar una cuenta (correo y contraseña).

Una vez explicadas estas herramientas que se han usado para gestionar los datos y las pantallas, vamos a observar como se ha implementado esta pantalla de registro:

Comenzamos introduciendo unos ***TextInputs*** para poder introducir los datos a registrar, y les introducimos las opciones necesarias para que puedan funcionar de forma correcta, veremos con detalle concretamente el ***TextInput*** de contraseña:

```

<TextInput
  style={styleLogin.input}
  value={password}
  onChangeText={text => setPassword(text)}
  ref={emailInput}
  placeholder="password"
  autoCapitalize="none"
  autoCorrect={false}
  secureTextEntry={!showPassword}
  returnKeyType="send"
  blurOnSubmit={true}
/>
<TouchableOpacity
  style={styleLogin.showPasswordButton}
  onPress={() => setShowPassword(!showPassword)}
>
  <Icon
    name={showPassword ? 'eye' : 'eye-slash'}
    size={25}
    color="gray"
  />
</TouchableOpacity>

```

fuente(creación propia)

Como se puede observar este **TextInput** contiene un **TouchableOpacity**, el cual permite hacer click sobre un ícono extraído de la librería “react-native-vector-icons”, que nos permite utilizar los iconos vectorizados de una lista muy larga de opciones, en este caso al ser el TextInput de la contraseña se ha añadido un ícono de ojo para mostrar u ocultar los caracteres de la misma.

Para guardar y manejar los datos introducidos se han utilizado **useState**, unos **hooks** que permiten almacenar y actualizar unos valores en el estado de un componente, permitiendo mantener y manipular los datos de forma dinámica, sin necesidad de crear clases extra. Los **useState** utilizados son los siguientes:

```
const [email, setEmail] = useState<string>();
const [password, setPassword] = useState<string>();
const [showPassword, setShowPassword] = useState<boolean>(false);
```

fuente(creación propia)

Estos **useState** han almacenado de forma dinámica los valores introducidos en los **inputs** para que después de pulsar el botón de registro se envíen a la base de datos de autenticación de Firebase, este envío de datos se realiza de la siguiente forma:

```
createUserWithEmailAndPassword(auth, email, password)
  .then((userCredential) => {
    const user = userCredential.user;
    navigation.navigate('Login');
  })
  .catch((error) => {
    const errorMessage = error.message;
    Alert.alert(errorMessage);
  });
});
```

fuente(creación propia)

Como se puede observar en la imagen, una vez creado el usuario la aplicación te llevaría de forma automática a la pantalla de “Log-In”, y en caso de que surja cualquier error este se mostrará en pantalla, indicando al usuario a que se debe el mismo. Cabe destacar que para mostrar exactamente el error de forma clara habría que ver el código de error para mostrar un mensaje personalizado para el usuario (no incluido en la imagen para no abusar de capturas de código).

En el caso de la pantalla de “Log-In” sería exactamente igual que el registro pero sustituyendo la función **createUserWithEmailAndPassword** por la función

signInWithEmailAndPassword, con exactamente la misma estructura pero comprobando que las credenciales sean correctas, comprobando su existencia en la base de datos. Una función extra añadida a la pantalla de “Log-In” es la opción “He olvidado mi contraseña” que nos redirige a la página de recuperación de contraseña, que funciona enviando un correo electrónico al usuario, a través de la función correspondiente que nos proporciona Firebase.

Una vez nos hemos logueado, la aplicación comprueba si es la primera vez que inicias sesión, comprobando si existe un documento en la base de datos con el uid del usuario (generado de forma automática por Firebase), en caso de ser la primera vez, este documento no se encontrará y el usuario será redireccionado a la pantalla de registro de datos, la cual con unos **TextInputs** recoge los datos, guardándolos en **useStates** al igual que en las pantallas anteriores y utilizando la siguiente función:

```
const uid = user.uid;
await setDoc(doc(db, "users", uid), {
  mail: mail,
  name: name,
  surname1: surname1,
  surname2: surname2,
  phone: phone,
  dni: dni,
  wallet: "0.00",
});
navigation.navigate('Main');
```

fuente(creación propia)

Se guardan los datos del usuario, relacionándolos a través del **id** del documento y el **uid** del usuario que son exactamente el mismo.

Cabe destacar que a partir de aquí estamos haciendo uso del servicio de Firebase llamado **Firestore**, el cual funciona con la misma configuración que hemos realizado siguiendo la documentación de Firebase, las funciones que vamos a utilizar son **setDoc** (mostrada en la foto anterior), **addDoc**, **getDocs** y **deleteDoc**.

Una vez se tienen todos los datos necesarios del usuario, la aplicación se dirige automáticamente a la pantalla principal, la cual se compone de dos **inputs** de texto más, que sirven para introducir un origen y un destino para realizar la búsqueda, al igual que en las pantallas anteriores, los datos se guardan en **useStates**, los cuales una vez pulsado

el botón de búsqueda se envían a la función correspondiente para poder realizar la llamada a la *API* y posteriormente guardar la respuesta en la base de datos:

```
const handleFindFlights = async () => {
  // Realiza la solicitud a la API
  fetch(`https://airlabs.co/api/v9/schedules?arr_iata=${arr_iata}&dep_iata=${dep_iata}&api_key=${API_KEY}`)
    .then(response => response.json()) // Parsea la respuesta JSON
    .then(data => {
      // Guarda cada objeto del array "response" por separado en la base de datos de Firebase
      const response = data.response;
      response.forEach(obj => {
        addDoc(collectionRef, obj)
          .then(docRef => console.log('Documento guardado:', docRef.id))
          .catch(error => console.error('Error al guardar el documento:', error));
      });
    })
    .catch(error => console.error('Error al obtener los datos:', error));
  await fetchFlights();
}

const fetchFlights = async () => {
  const q = query(
    collection(db, 'flights'),
    where('arr_iata', '==', arr_iata),
    where('dep_iata', '==', dep_iata)
  );

  try {
    const querySnapshot = await getDocs(q);
    const flightsData = querySnapshot.docs.map(doc => doc.data());
    setFlights(flightsData);
    setSchedule(true);
    setShowDropdown(false);
  } catch (error) {
    Alert.alert("Ha ocurrido un error, intentelo de nuevo más tarde")
  }
};
```

fuente(creación propia)

Como se puede observar en la imagen, la primera acción que se realiza al pulsar el botón de búsqueda es hacer una llamada a la *API* de **AirLabs**, la cual, filtrando por aeropuerto de origen y destino, nos devuelve los horarios, sacando los vuelos disponibles en un rango de 10 horas, una vez recibida la respuesta, esta se guarda en la base de datos, guardando cada vuelo como un documento diferente, es importante mencionar que si se repite algún vuelo, este no se guardaría en la base de datos, puesto que solo existe un vuelo con los mismos datos al mismo tiempo, esto lo gestiona Firebase de forma automática si se ha realizado una correcta configuración de la colección.

Después se hace una llamada a la base de datos, la cual muestra los resultados obtenidos de la búsqueda del usuario en un *ScrollView* con un componente que muestra los datos necesarios del vuelo (hora, origen, destino y precio), al hacer click sobre uno de ellos, nos dirigiremos a la pantalla de comprar billete. En caso de ocurrir algún error, saltará una alerta indicando que ha ocurrido un error y que se intente de nuevo más tarde.

Cabe destacar que se está comprobando la hora del dispositivo:

```
const currentTime = new Date().toLocaleTimeString();
```

fuente(creación propia)

y comparando contra la base de datos si hay algún vuelo que haya despegado ya, en caso de que haya despegado este se borra.

El precio de los vuelos se basa únicamente en la distancia en kilómetros que separa los aeropuertos, con un precio de 0.15 Euros el kilómetro. Para hallar la distancia entre los aeropuertos en la tierra, he utilizado una fórmula que permite calcularlo a través de coordenadas sexagesimales (latitud y longitud). Realizando una llamada a la colección *iataports* se obtienen las coordenadas y el nombre de cada aeropuerto, para mostrar su nombre en lugar de su código *iata*. La fórmula adaptada al código es la siguiente:

```
const calcPrice = (lat1, lon1, lat2, lon2) => {
  const radioTierra = 6371; // Radio de la Tierra en kilómetros

  const toRad = (valor) => (valor * Math.PI) / 180; // Conversión de grados a radianes

  const deltaLat = toRad(lat2 - lat1);
  const deltaLon = toRad(lon2 - lon1);

  const a =
    Math.sin(deltaLat / 2) ** 2 +
    Math.cos(toRad(lat1)) * Math.cos(toRad(lat2)) * Math.sin(deltaLon / 2) ** 2;

  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

  const distancia = radioTierra * c;
  const price = distancia * 0.15;
  return price.toFixed(2);
}
```

fuente(creación propia)

Como se puede apreciar en la imagen, se hace un cálculo con las coordenadas y la curvatura de la tierra, lo que nos proporciona una distancia aproximada en kilómetros debido a que la fórmula no contempla la altitud de los aeropuertos. Después, a esa distancia se le multiplica 0.15 (que es el precio del kilómetro) y se devuelve el precio que corresponda (siempre con dos decimales).

Con otra llamada a la base de datos, al hacer click sobre un componente de vuelo, se comprobará que el usuario tiene saldo suficiente en la cartera virtual de la aplicación, en caso afirmativo, se le llevará directamente a la pantalla de compra de billete, donde podrá llenar los datos necesarios para realizar la compra del mismo. En caso de que no tenga suficiente saldo, se mostrará una alerta avisando al usuario de que su saldo es inferior a la cantidad requerida para comprar el billete y se le dará la opción de recargar la cartera o ver los detalles del billete de todas formas.

Una vez nos encontramos en la pantalla de compra de billete, podemos observar una serie de preguntas y valores que debemos llenar y contestar para poder realizar la compra, tal y como se puede ver en los bocetos de las pantallas del apartado **6.2 Diseño de solución de propuesta**. También tenemos el botón de validar los datos del/los pasajero/s, el cual crea los billetes con los datos de cada pasajero y un campo llamado *creating* que indica que se está creando. Más adelante, a la hora de pulsar el botón de comprar, se buscan en la base de datos los billetes que están en creación del usuario, filtrando por su id, una vez obtenidos, se actualizan todos los campos con los datos del billete y, si se realiza la compra, se deshabilita el campo *creating* para dar por finalizado este paso. Si se cancela la compra, se buscan en la base de datos los billetes del usuario en creación y se procede a su borrado.

Para terminar, podemos ver el precio y un nuevo botón de compra, el cual comprueba que el saldo en la cartera sea suficiente (en caso de que se quiera comprar más de un billete a la vez), si es insuficiente, se le alertará y redireccionará a la pantalla de recarga de la cartera, si el saldo es suficiente, se le pedirá confirmación, y, si se acepta, los datos se guardarán en la base de datos, creando un documento en la colección **boughtflights** donde se almacenará el billete de cada pasajero de forma individual (utilizando el uid del usuario para que aparezcan en su historial si ha comprado un billete para un segundo o un tercero).

Una vez realizada la compra y las operaciones internas pertinentes, se podrán ver los componentes que simbolizan los billetes comprados en la pantalla de “Mis vuelos”. Éstos se muestran realizando una llamada a la base de datos de todos los vuelos que contienen el *uid* del usuario, para mostrar sólo los que ese usuario ha comprado. En estos componentes podremos ver: la hora del vuelo, el origen, el destino y el número de pasajeros del billete que ha comprado el usuario, al hacer click sobre uno de estos

componentes, se abrirá una pantalla con todos los datos del billete de cada usuario de forma separada, mostrando en la parte superior un código ***QR*** el cual se genera de forma automática con el ***id*** del documento del billete que al crearse de forma automática por Firebase siempre es único. Permitiendo así distinguir cada billete de forma individual para que un empleado pueda validarlos al escanearlos:

```
<QRCode
  size={350}
  value={BoughtId}
  logo={{ uri: LogoImage }}
  logoSize={60}
  logoBackgroundColor='transparent'
/>
```

fuente(creación propia)

El código ***QR*** se genera gracias a la librería ***react-native-qrcode-svg***, la cual al recibir un valor en forma de texto genera un código con ese texto, en este caso es el ***id*** del documento que contiene los datos del billete en cuestión, como ya he mencionado anteriormente.

Cuando un empleado lo escanea, las operaciones que se realizan internamente son una lectura al código ***QR*** y una búsqueda en la base de datos por el ***id*** del billete, una vez localizado el billete en la base de datos, se modifica el campo llamado ***validated*** de false a true para que se muestre como validado al momento del escaneo:

```
<View style={styleFlight.container}>
  <BarcodeScanner
    onBarcodeScanned={scanned ? undefined : handleBarcodeScanned}
    style={styleFlight.qrScanner}
  />
  {scanned && <Button title={'Escanear'} onPress={() => setScanned(false)} />}
</View>
```

fuente(creación propia)

Cómo se puede observar en la imagen, el escáner de código ***QR*** escanea de forma automática y en caso de que ocurra un error, se avisará al empleado, el cual deberá escanear de nuevo el código. También tiene disponible un botón para que se reinicie el escáner de forma manual, en caso de éste que no se reinicie automáticamente al final de la función llamada ***handleBarcodeScanned***:

```

const handleBarcodeScanned = ({ data }) => {
  setScanned(true);
  try{
    updateDoc(doc(db, 'flights', data),{
      | validated: true
    })
  }catch (error){
    Alert.alert("Ha ocurrido un error al validar el billete, intentelo de nuevo");
  }
  setScanned(false);
};

```

fuente(creación propia)

Como se puede ver, la función recibe el *id* del billete y lo filtra en la base de datos para así poder modificar el valor de *validated* con la función *updateDoc*. Lo que indicaría que el billete se ha validado con éxito. (Este proceso se repite con cada uno de los billetes de forma individual).

De forma visual en la pantalla del usuario saldría un tick de color verde indicando que el billete está validado y por tanto se ha completado el *check-in*.

Haciendo uso de la barra de navegación inferior podemos observar la pantalla que contiene el mapa a tiempo real de todos los aviones el cual se obtiene directamente desde la ***API* de Airlabs**, cabe destacar que, debido a que la versión gratuita de **Airlabs** solo dispone de mil llamadas mensuales. Cuando estamos realizando la llamada para el mapa a tiempo real estamos consumiendo una llamada cada segundo lo que nos dejaría con un total de 16 minutos de funcionalidad aproximadamente y teniendo en cuenta que cada vez que se realiza una búsqueda de billetes se consume una llamada he decidido que solo se realice una vez la llamada al mapa, consumiendo una sola llamada y por ende siendo necesario recargar la página si se quiere ver la posición de los aviones en ese momento. De todas formas en los apartados **10. Problemas encontrados** y **11. Evolución y trabajo futuro**

entraré en más detalles sobre cómo se podría solucionar este problema que ocasiona la limitada cifra de llamadas disponibles de la ***API***.

De nuevo haciendo uso de la barra de navegación inferior podemos observar la última vista, la cual contiene toda la información del usuario, la cantidad de dinero que contiene su cartera, sus datos y tres opciones que podemos clicar para navegar a otras

pantallas que, al igual que todas las pantallas de la aplicación se hace con `navigation.navigate()`. Las funciones que podemos encontrar en esta pantalla son las siguientes:

- Recargar la cartera: Al hacer click sobre ella nos lleva a la pantalla donde podremos recargar la cartera, esta es exactamente la misma pantalla a la que nos lleva la aplicación si a la hora de comprar un billete si no tenemos saldo suficiente.
- Ajustes de la cuenta: Donde podremos modificar cualquier dato que hayamos introducido en la fase de registrar los datos (Explicado anteriormente) por si hubiésemos cometido cualquier error o hubiésemos algún cambio en alguno de los datos
- Cerrar sesión: al pulsar sobre esta opción cerraremos la sesión de la cuenta actual para poder acceder con otra cuenta diferente si se desea.

Después en la misma pantalla podemos observar un historial de los vuelos realizados.

Cabe destacar que la base de datos de **Firebase** nos brinda la opción `onSnapshot` la cual nos permite actualizar la base de datos a tiempo real, así que en el momento en el que un dato cambia o se añade a la base de datos, se actualiza en la aplicación de forma automática. De esta forma el usuario no tiene que estar recargando la aplicación para ver si algo ha cambiado.

```
const documentRef = doc(db, 'tickets');
const unsubscribe = onSnapshot(documentRef, (doc) => {
  if (doc.exists()) {
    setReload(!reload)
  }
});
```

fuente(creación propia)

Una vez explicado con detalle el funcionamiento de la aplicación a nivel interno, podemos dar este apartado por finalizado.

6.2 Diseño de solución de propuesta

A continuación, podremos ver los bocetos de todas las pantallas de la aplicación para poder tener una idea visual de lo explicado en el punto anterior **6.1 Propuesta de**

solución. Lógicamente, al no ser las pantallas definitivas en la versión final de la aplicación, muchas de ellas pueden ser diferentes en la versión final del proyecto:



Aquí podemos observar la pantalla de carga de la aplicación, que se muestra al arrancar la aplicación, mientras está cargando y, gracias a **Expo** esto se controla de forma automática.

Una vez la aplicación ha cargado completamente se da paso a la pantalla de “login”, que es la primera en mostrarse al abrir la aplicación por primera vez.



La pantalla que vemos aquí es dicha pantalla de “Log-In” o “Inicio de sesión”, en la cual vemos unos campos donde podremos introducir el correo electrónico y la contraseña de la cuenta, siempre que la hayamos registrado previamente para acceder a la aplicación y poder utilizarla con total libertad. También contiene un botón que nos sirve para validar el correo y la contraseña y, si son correctos y existen en la base de datos, podremos acceder con normalidad a la app.

También podemos observar dos textos, uno que dice “Olvidé mi contraseña”, que sirve, en caso de haber olvidado la contraseña, para poder restablecerla a través de un correo electrónico. Este servicio está moderado y controlado por **Firebase**. Y otro texto que dice “¿No tienes cuenta?” -> Registrarme” la cual nos lleva a la siguiente pantalla:



Que se trata ni más ni menos que de la pantalla de “Registro”, la cual nos permite registrar un correo electrónico y una contraseña para crear una cuenta en la aplicación. Al igual que la pantalla de “Inicio de sesión”, se compone de dos campos para introducir los datos, un botón para validar y crear la cuenta y un texto que al hacer click sobre él, nos llevaría de vuelta a la pantalla de “Inicio de sesión”.

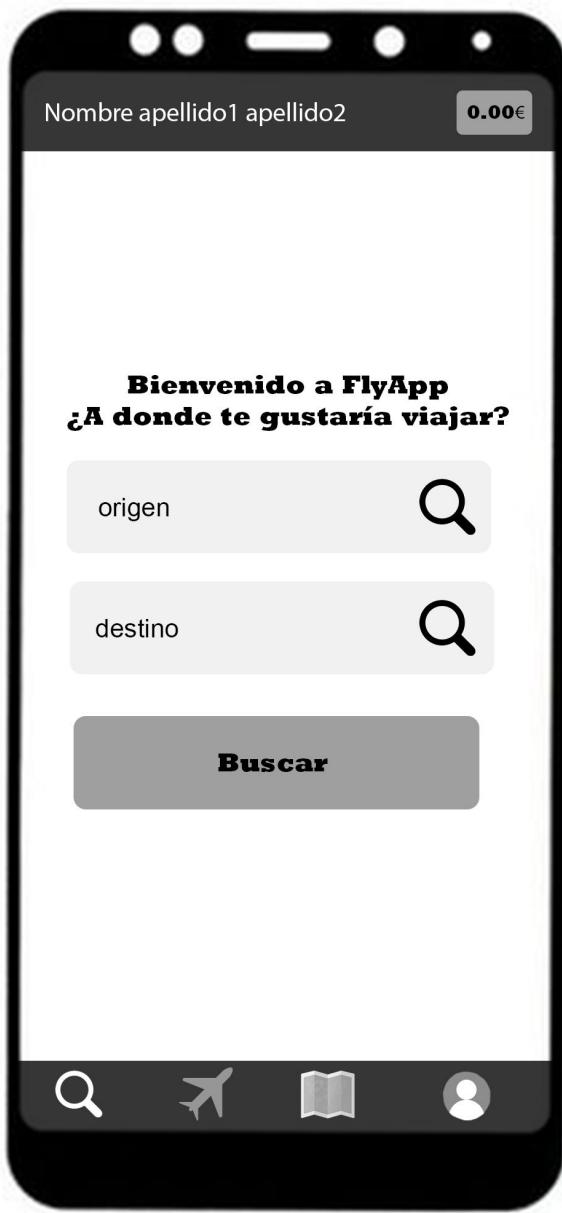
Una vez finalizado el registro del correo y la contraseña del nuevo usuario que será

introducido a la base de datos, aparecerá lo siguiente:



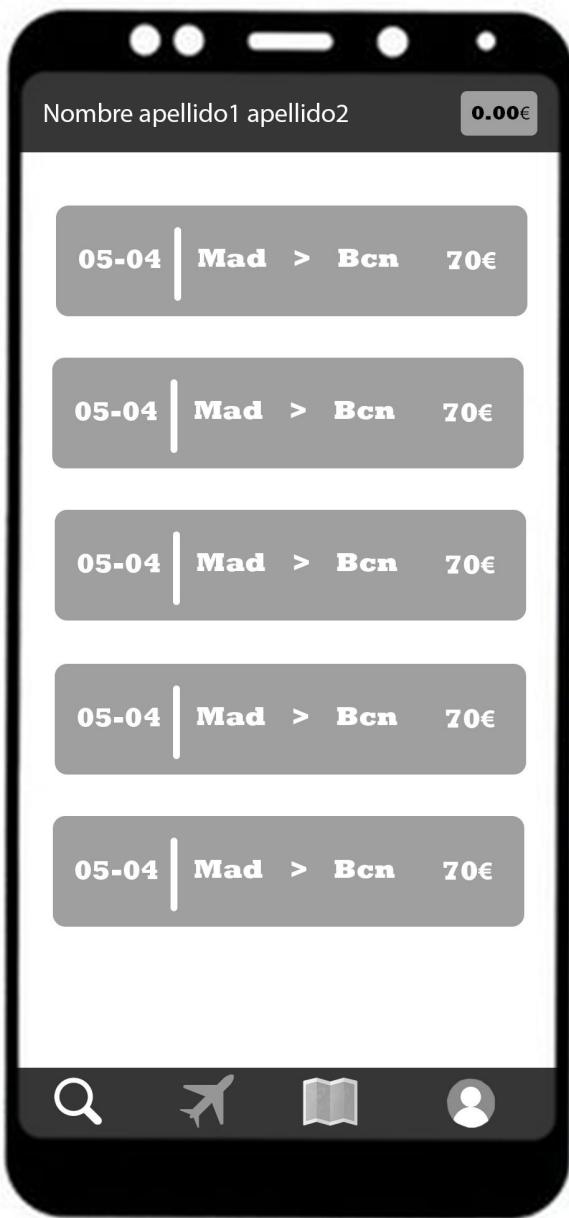
La pantalla que podemos ver aquí es la pantalla de registro de los datos del usuario, como he explicado anteriormente en el apartado **6.1 Propuesta de solución** esta pantalla sólo aparece en el momento en el que registramos una cuenta nueva, guardando los datos de la persona para que en el futuro, a la hora de comprar un “billete” para un vuelo que sea para él o ella, no tenga que introducir sus datos. Estos se introducirán de forma automática, permitiendo su edición manual.

Al igual que las dos pantallas anteriores podemos observar unos campos para introducir los datos y un botón para validar y guardar esos datos.



Aquí tenemos la pantalla principal de la aplicación. Podemos observar que se compone de una barra que contiene el nombre y apellidos del usuario y el valor que contiene en la “Cartera virtual” con la que funciona la aplicación, si hacemos click sobre el monto de la “Cartera virtual” accederemos a la pantalla donde la podremos recargar con “dinero ficticio” y poder así comprar “billetes”. También tenemos dos campos para introducir un origen y un destino. Y un botón para buscar vuelos entre esos dos aeropuertos (a nivel nacional).

Después podemos ver en la parte inferior una “Barra de navegación” con unos iconos que nos permiten navegar entre distintos apartados dentro de la aplicación.

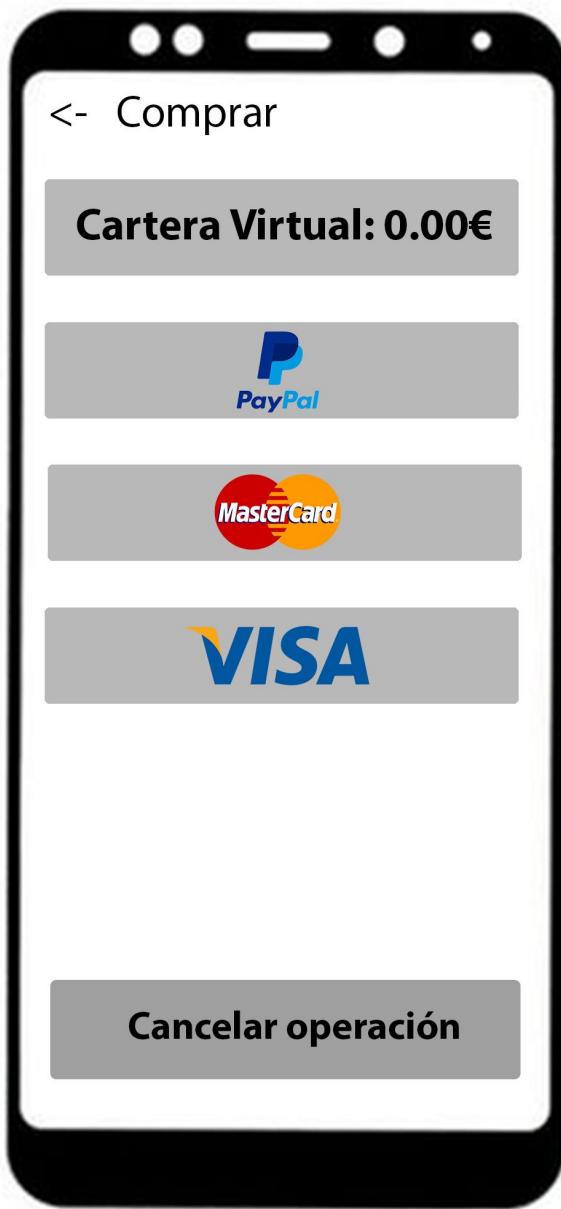


Aquí podemos observar el boceto de la pantalla principal de la aplicación una vez introducidos un origen y un destino (en los campos de la pantalla anterior). En este caso se simula una búsqueda entre los aeropuertos de “Madrid” y “Barcelona”. Como se puede ver, en este caso tenemos varios “vuelos”, si ahora se hace click sobre uno de ellos, se abrirá la vista de “Comprar billetes”:



En esta pantalla podemos observar las preguntas y datos que debemos contestar y llenar para poder adquirir un billete de forma exitosa, cabe destacar que la selección de asiento es un desplegable, puesto que es un campo opcional. Los asientos ocupados se mostrarán en color rojo. También cabe destacar que en caso de no seleccionar ninguno se asignará uno que esté libre de forma aleatoria.

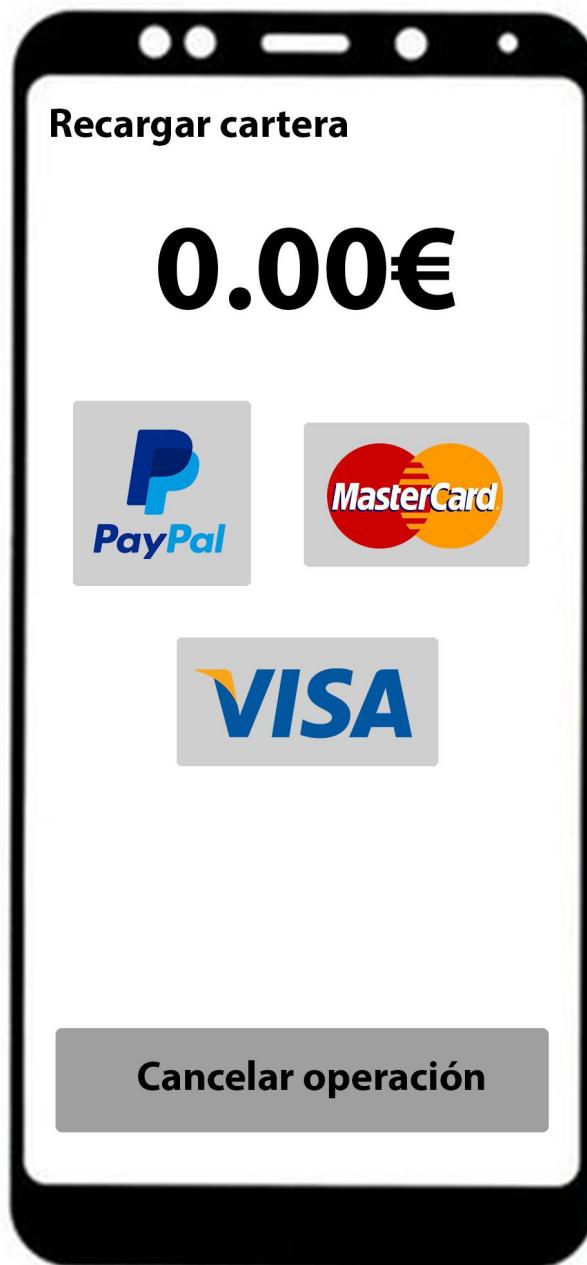
Como he mencionado en el punto anterior **6.1 Propuesta de solución**, en caso de no tener saldo suficiente para completar la compra se redireccionará al usuario con una alerta a la pantalla de recarga de cartera, que veremos a continuación.



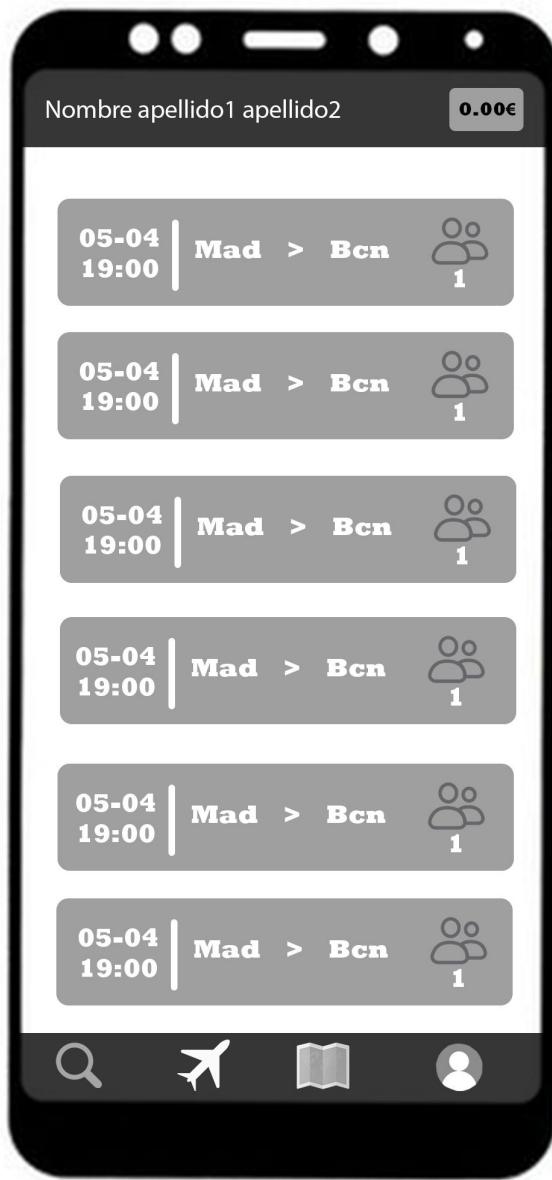
En esta pantalla tenemos las opciones de pago, nada más cargar se comprueba internamente el saldo disponible de la cartera y en caso de no tener suficiente se avisa al usuario.

Si se pulsa la opción de pagar con la cartera lo que se hace es restar el precio del billete a la cartera y se crea el billete, guardándolo en la base de datos.

Si se pulsa cualquiera de las otras opciones lo que se hace es crear el billete directamente.



En esta vista se observa la opción de recargar la cartera interna de Flyapp, en ella, se debe seleccionar desde qué plataforma de pago se realizará dicha recarga, siendo éstas opciones, como se puede ver en la figura, VISA, MasterCard y Paypal, para darle la opción al usuario de utilizar la que se le antoje.



Aquí podemos ver la pantalla de “Mis vuelos”, donde se pueden observar los vuelos que el usuario ha comprado, de tal forma que quedan accesibles para poder consultar cualquier tipo de información o poder acceder al código *QR* a la hora de hacer el “Check-In” y que un empleado pueda validar ese billete.

Al hacer click sobre alguno de ellos nos lleva a la siguiente vista con toda la información del billete:



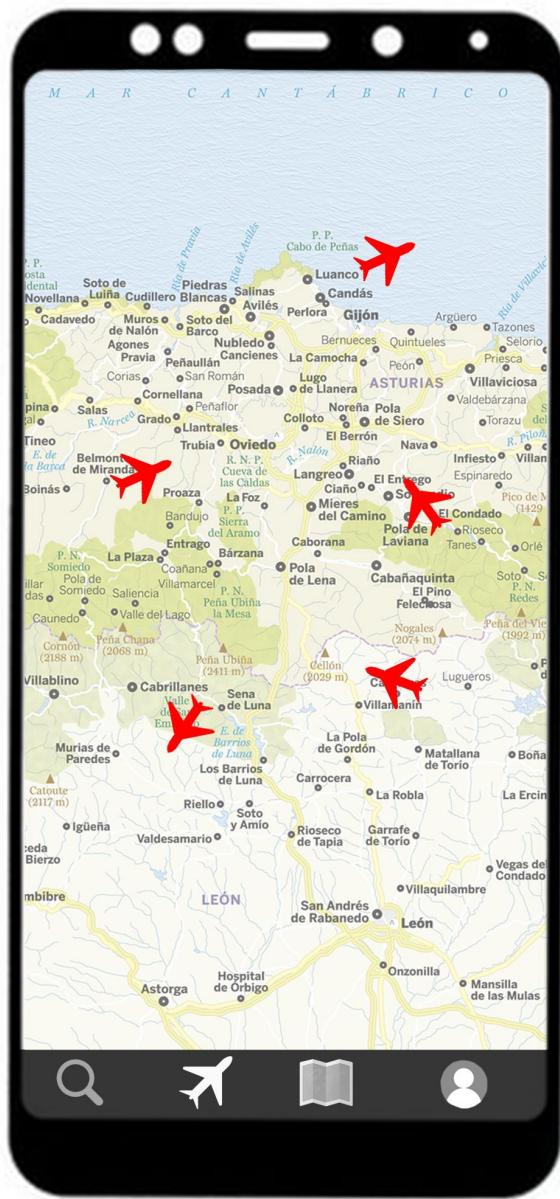
Como podemos observar esta es la pantalla donde podemos ver los billetes. El código **QR** se usa para validar los billetes, justo debajo podemos observar los datos del pasajero al que corresponde dicho billete.

Después podemos observar toda la información del vuelo, la hora, la aerolínea, etc.

Una vez el billete ha sido validado se mostrará encima del código **QR** un tick de color verde que indica que ha sido validado con éxito.



Aquí podemos observar un boceto de lo que vería un empleado a la hora de validar un billete de un pasajero, vemos que tenemos un botón con el que podemos realizar nuevas lecturas, aunque no es necesaria su pulsación ya que se debería de reiniciar automáticamente al escanear un *QR*.



En esta vista podemos ver el mapa, en el cual salen los aviones que hay en marcha a tiempo real, se pueden seleccionar y ver sus datos, ruta, etc. Permitiendo, por ejemplo, si un familiar va a llegar al aeropuerto ver cuánto queda para el aterrizaje (Debido a que este servicio consume muchas llamadas a la API no es funcional a tiempo real, simplemente hace una llamada y guarda esos datos en la base datos, mostrando únicamente la posición de los aviones en el momento de hacer la llamada. En el apartado **11. Evolución y trabajo futuro** entraré en más detalle sobre esta vista.).



Esto es un boceto inicial de la pantalla del área de usuario, donde se puede ver la cantidad de “dinero virtual” que se tiene actualmente en la cartera, seguido del nombre completo de usuario y una serie de opciones que nos permiten hacer distintas funciones:

- La primera función es la de “recargar cartera”, al hacer click sobre ella nos lleva a la vista donde podemos “ingresar dinero virtual” de forma ficticia a esta cartera, pudiendo así realizar compras en la aplicación.
- La segunda función nos lleva a la vista que contiene los ajustes de la cuenta, la cual nos permite cambiar datos de la cuenta, tales como el nombre, número de teléfono, dni, contraseña, etc. Y visualmente es igual a la pantalla de registro de

datos del usuario.

- La tercera opción nos permite cerrar sesión para poder iniciar la aplicación con otra cuenta distinta o simplemente cerrar la sesión cuando el usuario lo vea necesario.

Por último podemos ver una lista de “vuelos pasados”, donde sale un historial de los vuelos que se han realizado en esa cuenta a lo largo del tiempo.

Antes de terminar cabe destacar que la opción de eliminar la cuenta se encuentra en esta pantalla, para acceder a ella tendremos que hacer scroll hacia abajo para que aparezca el botón ya que este se encuentra fuera de la vista, oculto bajo la barra de navegación inferior.

7. Plan de trabajo

El plan de trabajo que voy a seguir a lo largo de todo el desarrollo del proyecto va a ser mediante el modelo de “Sprints” divididos en semanas. Cada uno de estos “Sprints” semanales está compuesto por pequeños objetivos individuales, como las pantallas, las funcionalidades, las bases de datos, etc.

De esta forma puedo mantener un control estable y ordenado sobre el desarrollo, sabiendo en todo momento en que punto del desarrollo voy a estar y viendo con antelación si algo necesita más tiempo de lo esperado o en su defecto dedicarle más tiempo al día para sacarlo adelante.

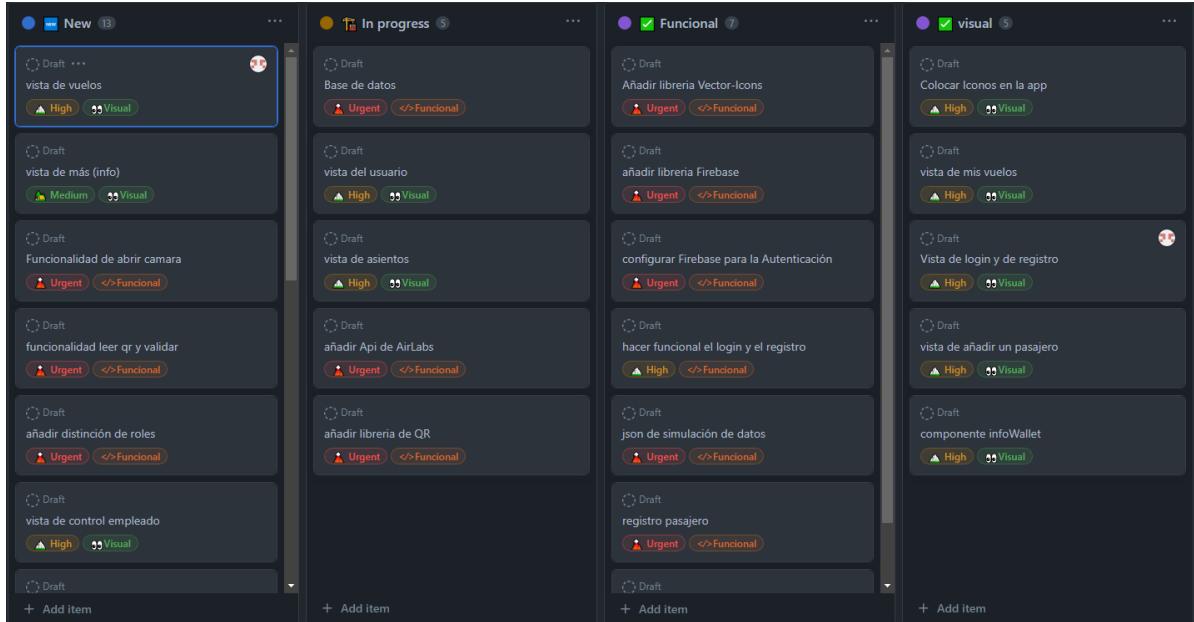
Los primeros días voy a estar desarrollando y organizando de forma escrita la idea de la aplicación, pensando que es lo que voy a necesitar a lo largo del desarrollo, cómo me voy a organizar exactamente, qué tecnologías voy a usar y cómo lo voy a estructurar.

Una vez tenga organizado todo correctamente comenzaré creando un proyecto en **Github** para ir marcando los objetivos y tener de forma visual todo lo que tengo o vaya

organizando para desarrollar la aplicación.

Una vez puestos todos los objetivos en el proyecto de **Github** lo único que tengo que ir haciendo es mover los objetivos a las columnas necesarias para recordarme a mí mismo en qué punto del desarrollo me encuentro.

Lo explicado en el párrafo anterior se muestra en la siguiente imagen:



fuente(creación propia)

A lo largo del proceso de organización y desarrollo han ido surgiendo algunos errores en planteamiento o errores de programación, entrará en más detalles sobre esto en el punto **10. Problemas encontrados**.

También para tener de forma más sencilla y visual los sprints estoy haciendo un diagrama de gantt, el cual dejo a continuación:

FlyApp



fuente(creación propia)

8. Desarrollo de la solución

8.1 APIs utilizadas



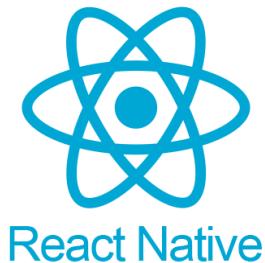
La **API** que se ha utilizado (desde la cual se obtiene toda la información de los vuelos) ha sido **AirLabs**, esta **API** provee en tiempo real los horarios y la información de todos los vuelos existentes en el mundo a tiempo real y con una anticipación de 10 horas en el plan gratuito, aunque los planes de pago ofrecen muchas más ventajas y, lógicamente mucho más tiempo de antelación en todos los datos.

AirLabs ofrece un total de 1000 llamadas al mes, sin límite diario y con la posibilidad de crear varias cuentas en caso de ser necesario ya que no te pide los datos de ninguna tarjeta ni método de pago, por este motivo y porque es la **API** que más información de vuelos brinda ha sido la elegida para usarse en esta aplicación

9. Despliegue e instalación

9.1 Creación del Proyecto

Para desarrollar la aplicación utilizaremos la tecnología llamada React Native:



con el Framework/Cliente llamado **Expo**:



El cual permite desplegar la aplicación de una manera muy rápida y sencilla, realizando las configuraciones para IOS y ANDROID de forma casi automática. El desarrollo de la aplicación se puede realizar en cualquier sistema operativo, tanto en Windows como en Ubuntu (Linux) o MACos, ya que React native con **Expo** nos permite trabajar en todos estos sistemas operativos sin ningún problema.

Gracias al control de versiones con **Git** se puede tener un control total sobre el proyecto, permitiendo que si ocurre algún error en el desarrollo éste se pueda solucionar volviendo a un punto anterior. En este caso vamos a utilizar **Github**.



Para configurar **Github** necesitaremos tener una cuenta creada y descargar **Github Desktop** si nos encontramos en un sistema Windows o bien utilizar comandos de terminal si nos encontramos en un sistema Linux o MACos. Una vez tengamos todo listo crearemos un repositorio en **GitHub**, puede ser público o privado ya que no influye en absolutamente nada de nuestro trabajo.

Cuando tengamos el repositorio creado lo que haremos será clonarlo en una carpeta en nuestro equipo y posteriormente crear el proyecto de React dentro de esa carpeta, después subiremos los cambios realizados al repositorio para que cada cambio futuro que se realice a lo largo del desarrollo se pueda subir y tener activo de esta forma el control de versiones.

Es importante realizar una subida de los cambios cada día que avancemos en el proyecto o cada vez que desarrollemos algo importante, para evitar la posibilidad de que el proyecto falle y perdamos el trabajo realizado no guardado.

Para instalar **React Native** y **Expo** nos hace falta instalar **node.js**



En mi caso, debido a problemas de compatibilidad en las últimas versiones, **Expo** me pidió que instalara la versión de node.js 16.20.0 puesto que es la más estable con **Expo** al momento del desarrollo de este proyecto.

Como ya he comentado se va a usar **Expo**. Y para ello he tenido que instalarlo a través de comandos en la Powershell de windows, aunque el proceso sería exactamente igual en cualquier sistema operativo, es importante resaltar que este proceso debe realizarse dentro de la carpeta del repositorio clonado en nuestro dispositivo, una vez en su interior, con la terminal procederemos a ejecutar el siguiente comando para instalar **Expo**:

- ***npm install -g expo-cli***

Una vez que tenemos **Expo CLI** instalado en el equipo debemos ejecutar el siguiente

comando:

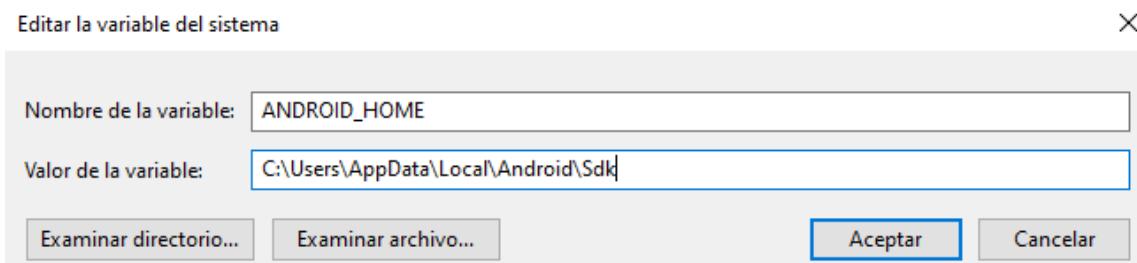
- `npx create-expo-app NombreApp`

Este creará la aplicación haciendo las configuraciones básicas y descargando una plantilla sobre la que podremos comenzar a trabajar.

Una vez completado este proceso ejecutamos el siguiente comando para iniciar el servidor y poder trabajar sobre la aplicación de forma visual y a tiempo real:

- `npx expo start --clear`

Para ver la aplicación que estamos desarrollando se puede utilizar un dispositivo virtual, el cual se debe configurar en el equipo, para ello descargamos **Android Studio** para dispositivos Android o **XCode** para dispositivos IOS, una vez descargado, nos iremos al apartado llamado **Virtual Device Manager** y descargamos un dispositivo virtual con la versión de Android que queramos (cuanto más reciente mejor). Después debemos crear una variable de entorno, para ello introducimos en el buscador de Windows variables de entorno, entramos y damos a la opción llamada **Variables de entorno**, después, en el apartado variables del sistema, tenemos que crear una nueva variable llamada **ANDROID_HOME** con el valor que tenga la ruta del **sdk** de Android, respetando siempre los nombres de los directorios o carpetas que contenga:



fuente(*creación propia*)

También está la opción de utilizar la aplicación móvil de **Expo** llamada **Expo Go**, la cual nos permite ejecutar y ver la aplicación en nuestro propio dispositivo, viendo todos los cambios que vamos realizando de forma instantánea, inalámbrica y sin necesidad de configurar un dispositivo virtual. Ya que nos genera un código **QR** que al escanearlo nos

permite ver la aplicación en tiempo real.

Ya tenemos casi todo listo, ahora al iniciar **Expo** podemos escanear el código **QR** que nos aparece en la consola (si estamos usando **Expo Go**) o podemos pulsar la tecla **a** para que se ejecute de forma automática el dispositivo virtual configurado (si no estamos usándolo).

Ahora necesitamos un **IDE**, como he mencionado anteriormente usaremos **VScode** así que iremos a descargarlo a la tienda de windows o a cualquier buscador con conexión a internet, una vez descargado, para utilizarlo debemos ejecutar en la consola este comando en la ruta donde se encuentra nuestro proyecto:

- **code .**

De esta forma se abrirá **VScode** de forma automática con todo el proyecto, permitiéndonos empezar a trabajar.

Ahora, debemos descargar todas las librerías que necesitaremos, los comandos de instalación pueden ser algo diferentes pero todos se pueden utilizar con **npm** o con **yarn** (requiere instalación adicional)

un ejemplo de comando de instalación sería este:

- **npm install @react-navigation/native**

dado este ejemplo, algunas de las librerías utilizadas han sido:

- **react-navigation**
- **react-native-gesture-handler**
- **react-native-vector-icons**
- **expo-linear-gradient**
- **expo-status-bar**
- **firebase**

al estar subido a **Github**, si nos traemos el proyecto debemos ejecutar el comando:

- ***npm install***

- o

- ***yarn install***

para instalar todas las dependencias necesarias.

Para el desarrollo se ha usado TypeScript que es una forma de “tipar” el código para evitar cometer errores y tener un código más limpio:



Para la base de datos he utilizado Firebase:



Para configurar Firebase en nuestro proyecto tenemos que crearnos una cuenta e iniciar un proyecto, seguiremos los pasos que nos salgan hasta llegar al punto en el que nos den la configuración de firebase para nuestra aplicación, la cual colocaremos en un archivo en la raíz del proyecto llamado “**firebaseConfig.js**”:

```
// Configuration for connect Firebase
export const firebaseConfig = {
  apiKey: "",
  authDomain: "",
  projectId: "",
  storageBucket: "",
  messagingSenderId: "",
  appId: "",
  measurementId: ""
};
```

fuente(creación propia)

Cabe destacar que debemos crear un proyecto de base de datos WEB ya que estamos utilizando React Native que viene directamente de React.

Una vez tenemos este archivo lo único que nos queda es importarlo e inicializarlo como he mostrado anteriormente en el punto **6.1 Propuesta de solución**, y ya tendremos disponible y funcional Firebase. Para poder utilizarlo en nuestro proyecto, solo tenemos que importar las funciones necesarias y consultar cómo funcionan en la documentación para desarrolladores que ofrece Firebase.

10. Problemas encontrados

Los problemas encontrados a lo largo del desarrollo han sido muy variados y abundantes, como es normal en un desarrollo, al fin y al cabo en un proyecto de este calibre es completamente normal encontrarse con problemas, por muy bien organizado que esté dicho proyecto, así que a continuación enumeraré los problemas que me he encontrado a lo largo del desarrollo de mi aplicación, **FlyApp**:

Comprar billetes:

A la hora de comprar varios billetes a la vez (para dos o más pasajeros) se sobrescribir los datos del pasajero y se generaban el número de billetes con los mismos datos, para solucionar esto se añadió un botón que valida los datos y crea los billetes con los datos de cada pasajero y un campo llamado *creating* que indica que se está creando. Más adelante a la hora de pulsar el botón de comprar, se busca en la base de datos los billetes que están en creación del usuario, filtrando con su id, una vez obtenidos se actualizan todos los campos con los datos del billete y si se realiza la compra se deshabilita el campo *creating* para dar por finalizada la compra. Si se cancela la compra se busca en la base de datos los billetes del usuario en creación y se procede a su borrado.

Compatibilidad de dependencias:

Uno de los errores más comunes que surgen son los problemas de compatibilidad entre dependencias, ya que con cada actualización algo puede cambiar y por ende dejar de funcionar plenamente. La solución que tenemos ante este problema es dejar versiones de las librerías estáticas, las podemos encontrar en el archivo **package.json**.

Otra posible solución es buscar dependencias que realicen las mismas acciones pero que sean compatibles con nuestra versión de **React Native** o **Expo**.

Llamadas de la *API*:

Las llamadas a la *API* han sido el mayor de los problemas a nivel de organización del proyecto, debido a que tenemos una gran cantidad de *APIs* que nos devuelven datos muy parecidos, pero todas ellas ofrecen en su plan gratuito muy pocas llamadas, por lo que el desarrollo se ha realizado pensando en optimizar cada una de las llamadas realizadas.

Base de datos:

Al estar en un servidor, el tiempo de respuesta de la base de datos puede variar, para solucionar este problema, he creado unos efectos de carga que no desaparecen hasta que se haya cargado todo correctamente.

Una vez que tenemos las respuestas de la base de datos, necesitamos filtrar los datos para que se muestre todo en pantalla de forma ordenada.

También tenemos el problema de que las llamadas se guardan completas en la base de datos y no podemos borrar todos los documentos y guardar de nuevo los de la llamada, ya que si dos o más usuarios buscan más o menos a la vez, los resultados de un usuario que esté buscando un vuelo desaparecen y se muestran los nuevos guardados, es decir, los que otro usuario esté buscando.

Para solucionar esto lo que se ha hecho es ir comprobando con la hora del dispositivo las horas de los vuelos en la colección de *flights*, permitiendo así que los vuelos que se pasen de la hora de despegue sean borrados.

Asientos:

Los asientos también han sido un problema, ya que existe la posibilidad de que dos usuarios compren el mismo a la vez o que seleccionen un asiento ya ocupado, lo que provocaba que la aplicación colapsase y se cerrase.

Para solucionar esto se realiza una comprobación, en caso de que esto ocurra no se guarda en la base de datos, si no que se advierte al usuario para que seleccione otros

asientos diferentes, también se añadió que los asientos aleatorios (en caso de que el usuario no haya seleccionado ninguno de los disponibles) vayan en fila, puesto que si son dos pasajeros es preferible que viajen juntos, aunque no seleccionen asientos específicos.

Calcular distancia y precio:

Calcular las distancias fue un problema que surgió al principio porque se necesitan las coordenadas de los aeropuertos para calcularla, esto se soluciona copiando a mano las coordenadas de cada aeropuerto de España y almacenándolas en la colección donde se guardan los nombres de los mismos aeropuertos, de esta forma solo se necesita hacer una llamada a la base de datos y, haciendo uso de la fórmula anteriormente mencionada, calcular la distancia entre los dos aeropuertos.

11. Evolución y trabajo futuro

Para el trabajo futuro y evolución de la aplicación se deben destacar varios puntos:

Lo primero que sería necesario es pagar la api de **AirLabs**, para poder disponer de un número de llamadas ilimitado, permitiendo hacer totalmente funcional el mapa a tiempo real, siendo indiferente el número de usuarios que lo estén visualizando a la vez.

Como segundo punto de trabajo futuro se deberían ampliar las funcionalidades, optimizando las ya existentes para mejorar el rendimiento de la aplicación y añadiendo funcionalidades nuevas como:

- Guardar rutas en favoritos.
- Poder ver los mejores asientos en la selección de asientos.
- Recopilar todos los aeropuertos existentes para extender el alcance de la aplicación.
- Mostrar un localizador encima del código **QR** y añadir una opción para empleados de validar el código introduciendo a mano el localizador en caso de fallo en la lectura.

Como tercer punto, cabe mencionar el despliegue de todos los servicios a través de un

servidor propio con una base de datos no relacional configurada en él, donde se realizarían todas las llamadas y gestiones necesarias en los vuelos disponibles, así como en los datos de los usuarios, mejorando el rendimiento de la aplicación considerablemente.

Por último, cabe la posibilidad de hablar con las aerolíneas y las principales vías de pago para conseguir de forma oficial los billetes y pagarlos de forma real para que se convierta en una aplicación totalmente funcional, como las que hemos mencionado anteriormente que ofrecen servicios parecidos a **FlyApp**.

12. Bibliografía

- [1] Documentación Expo: *Expo Documentation*. (s. f.). Expo Documentation. Se puede encontrar en: <https://docs.expo.dev/>
- [2] Documentación Firebase: Documentation. (s. f.). *Firebase*. Se puede encontrar en: <https://firebase.google.com/docs>
- [3] Documentación API AirLabs: AirLabs.Co. (s. f.). *AirLabs Data API*. Se puede encontrar en: <https://airlabs.co/>
- [4] Documentación React Navigation: *React Navigation | React Navigation*. (s. f.). Se puede encontrar en :<https://reactnavigation.org/>
- [5] Documentación dependencia generador de QR: Uzoma, C.(s.f.). *Creating QR Codes with React Native*. Se puede encontrar en :
<https://blog.openreplay.com/creating-qr-codes-with-react-native/>
- [6] Base de datos de aeropuertos: BroadcastVision. (s. f.). *Airport-ICAO-IATA-Database/airlines.sql* at *master*. *BroadcastVision/Airlines-ICAO-IATA-Database*. GitHub Se puede encontrar en :
<https://github.com/SqlHareketi/Airport-List-With-IATA-Codes/blob/master/db.sql>
- [7] Base de datos de aerolíneas: BroadcastVision. (s. f.). *Airlines-ICAO-IATA-Database/airlines.sql* at *master*. *BroadcastVision/Airlines-ICAO-IATA-Database*. GitHub. Se puede encontrar en :
<https://github.com/BroadcastVision/Airlines-ICAO-IATA-Database/blob/master/airlines.sql>