



BIENVENIDOS A BORDO

SpicyAirlines

¡Explora los cielos de China con nosotros!

中华人民共和国

世界人民大团结万岁



DAVID MARTÍN PULGAR





**TÉCNICO SUPERIOR EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA
(IFCS02)**

MÓDULO: PROYECTO. CURSO: 2024-2025

CONVOCATORIA: ORDINARIA/ EXTRAORDINARIA

Spicy Airlines

PROFESOR TUTOR: Mercedes Limón Echevarría

Alumnos:

APELLIDOS Y NOMBRE: David Martín Pulgar	DNI: 47474626G
---	--------------------------

MODALIDAD DE PROYECTO:

Modelo a): Proyecto de investigación experimental

Modelo b): Proyectos de gestión

Modelo c): Proyecto emprendimiento

Modelo d): Proyecto bibliográfico



ÍNDICE

<u>1. JUSTIFICACIÓN</u>	3
<u>2. INTRODUCCIÓN</u>	4
<u>3. TOMA DE REQUISITOS</u>	5
<u>Requisitos Funcionales</u>	5
<u>Requisitos No Funcionales</u>	6
<u>4. OBJETIVOS</u>	9
<u>5. DESARROLLO</u>	11
<u>5.1. METODOLOGÍA</u>	11
<u>5.2. TECNOLOGÍAS</u>	12
<u>Tecnologías utilizadas</u>	12
<u>Justificación</u>	14
<u>5.3. PROTOTIPADO Y USABILIDAD</u>	14
<u>5.4. MODELADO DE DATOS</u>	20
<u>Estructura General</u>	20
<u>Diagrama E/R (Entidad-Relación)</u>	21
<u>5.5. SEGURIDAD Y AUTENTICACIÓN</u>	23
<u>Base de datos: permisos y roles</u>	23
<u>Justificación de tecnologías</u>	24
<u>5.6. DISEÑO E IMPLEMENTACIÓN</u>	24
<u>1. Modelos de Datos (Clases Principales)</u>	28
<u>2. ViewModels (Gestión de Estado y Lógica de Negocio)</u>	29
<u>3. Clases Utilitarias (Lógica Adicional)</u>	33
<u>5.7. PRUEBAS Y CALIDAD DEL SOFTWARE</u>	34
<u>Calidad del código</u>	41
<u>6. DESPLIEGUE DEL PRODUCTO</u>	41
<u>Generación del APK</u>	41
<u>7. PRESUPUESTO, RIESGOS, VIABILIDAD</u>	42
<u>Presupuesto estimado</u>	43
<u>Gestión de riesgos</u>	43
<u>Riesgos estratégicos, económicos y sociales</u>	44
<u>8. DIFICULTADES ENCONTRADAS Y RESOLUCIÓN</u>	45
<u>9. CONCLUSIONES</u>	48
<u>10. REFERENCIAS BIBLIOGRÁFICAS</u>	50
<u>11. ANEXOS</u>	52



1. JUSTIFICACIÓN

En los últimos años, el interés por Asia, especialmente por China, ha crecido notablemente entre viajeros de todo el mundo. La fascinación por su cultura milenaria, su gastronomía única y sus paisajes impresionantes ha convertido a ciudades como Chengdu y Chongqing en destinos turísticos emergentes. Chengdu, conocida como la cuna de los pandas gigantes y la gastronomía picante, y Chongqing, con su paisaje urbano futurista recién sacado de una película de ciencia ficción, rodeado de montañas y ríos, ofrecen una mezcla perfecta entre tradición y modernidad que atrae cada vez más a viajeros jóvenes y aventureros.

China ha sido el cuarto país más visitado del mundo en 2024, recibiendo alrededor de **5,7 millones de viajeros anuales**. Este auge turístico se ve reflejado en datos recientes: durante las vacaciones del Primero de Mayo de 2025, el número de visitantes extranjeros en China aumentó un **43,1%** en comparación con el año anterior, y ciudades como Chengdu y Chongqing experimentaron un notable incremento en la llegada de turistas.

Debemos considerar que, desde el 30 de noviembre de 2024 hasta el 31 de diciembre de 2025, los ciudadanos españoles pueden ingresar a China sin necesidad de visado para estancias de hasta 30 días, siempre que el viaje sea por motivos de turismo, negocios, visitas familiares o tránsito.

En respuesta a esta creciente demanda y a las nuevas facilidades de entrada al país, surge "**Spicy Airlines**", una aplicación móvil diseñada para conectar Madrid con las ciudades chinas de Chengdu y Chongqing. Esta herramienta permite consultar, comparar y reservar vuelos según fechas, número de pasajeros, tipo de clase y temporada. Además, integra un sistema de autenticación, gestión personalizada de reservas y una experiencia visual moderna adaptada al estilo de vida móvil actual.

Por tanto, este proyecto no solo sirve como una práctica que fusiona todos los conocimientos adquiridos durante el ciclo formativo, sino que además plantea una solución tecnológica viable y atractiva para un nicho real en crecimiento.



2. INTRODUCCIÓN

Este proyecto tiene como objetivo el desarrollo de una aplicación móvil denominada **Spicy Airlines**, orientada a facilitar la **reserva de vuelos entre Madrid y dos importantes ciudades del suroeste de China: Chengdu y Chongqing**. La idea nace como respuesta al creciente interés turístico por China y la necesidad de ofrecer soluciones digitales específicas y centradas en destinos menos tradicionales pero altamente atractivos.

La aplicación ha sido diseñada para dispositivos Android y desarrollada utilizando **Kotlin** y **Jetpack Compose**, lo que garantiza una interfaz moderna, fluida y adaptable. Se han integrado servicios de backend a través de **Firebase**, tanto para la autenticación de usuarios como para la gestión de datos de vuelos, reservas y pasajeros.

Entre las principales funcionalidades de la aplicación destacan:

- **Selección de vuelos** según destino, fechas, número de pasajeros y clase.
- **Comparación de precios** al estilo de herramientas como Skyscanner.
- **Gestión de usuarios** con registro e inicio de sesión mediante Firebase Authentication.
- **Visualización de reservas** desde un perfil personalizado.
- **Gestión de pasajeros** con datos individuales por cada vuelo.
- **Condiciones dinámicas de precio** según la clase (Turista, Premium, Business).

Además, se han tenido en cuenta aspectos como la seguridad, la validación de datos del usuario, el control de disponibilidad de asientos en tiempo real y la usabilidad de la interfaz, priorizando la **experiencia de usuario (UX)** en cada paso del proceso de reserva.

Este documento recoge todas las fases del ciclo de desarrollo del software, desde la toma de requisitos hasta el despliegue y pruebas finales, siguiendo una metodología estructurada que permite demostrar tanto la viabilidad del proyecto como su correcta implementación técnica.



3. TOMA DE REQUISITOS

Requisitos Funcionales

1. Registro y autenticación

El usuario debe poder registrarse y autenticarse mediante Firebase Authentication.

2. Búsqueda de vuelos

El usuario podrá buscar vuelos entre Madrid, Chongqing y Chengdu, seleccionando fechas específicas. Se podrá buscar: Sólo ida o ida y vuelta.

3. Visualización de vuelos

El usuario podrá ver los resultados de vuelos o combinaciones disponibles, con información de:

- Precios por clase
- Horarios de salida y llegada
- Duración del vuelo
- Temporada del vuelo (baja, media o alta)

4. Selección de pasajeros

El usuario podrá indicar el número de adultos y menores (bebés menores de 3 años viajan gratis).

5. Selección de clase

El usuario podrá elegir entre clase Turista, Premium o Business, teniendo en cuenta la disponibilidad de asientos.

6. Disponibilidad de asientos

La aplicación verificará en tiempo real la disponibilidad de asientos desde Firebase para



cada clase.

7. Formulario de pasajeros

El usuario podrá introducir los datos de cada pasajero: nombre, apellidos, pasaporte, fecha de nacimiento y teléfono.

8. Resumen de reserva

Antes de confirmar, la app mostrará un resumen completo con los datos del vuelo, pasajeros y precio total.

9. Confirmación y almacenamiento de la reserva

El usuario podrá confirmar la reserva, que quedará registrada en Firebase asociada al usuario autenticado.

10. Gestión del perfil y reservas

Desde su perfil, el usuario podrá:

- Consultar sus reservas anteriores (ya realizadas) y futuras (pendientes).
- Editar los datos de los pasajeros de una reserva futura.
- Modificar su información personal (como ciudad o número de teléfono).
- Cerrar sesión.

Requisitos No Funcionales

● Rendimiento

- Los resultados de búsqueda deben cargarse en menos de 15 segundos.
- Las operaciones con Firebase deben ser ágiles y sin bloqueos visuales.



● Seguridad

- Autenticación con Firebase Authentication.
- Las reservas están siempre asociadas al usuario autenticado.
- Cada usuario solo podrá ver y modificar su propia información y reservas.

● Usabilidad

- Interfaz clara e intuitiva con Jetpack Compose.
- Diseño adaptado a distintas resoluciones móviles.
- Uso de iconos visuales y navegación fluida.

● Escalabilidad

- Firebase Firestore permite ampliar el sistema fácilmente (más destinos, más clases...).
- Arquitectura desacoplada con ViewModel por pantalla y navegación modular.

● Compatibilidad

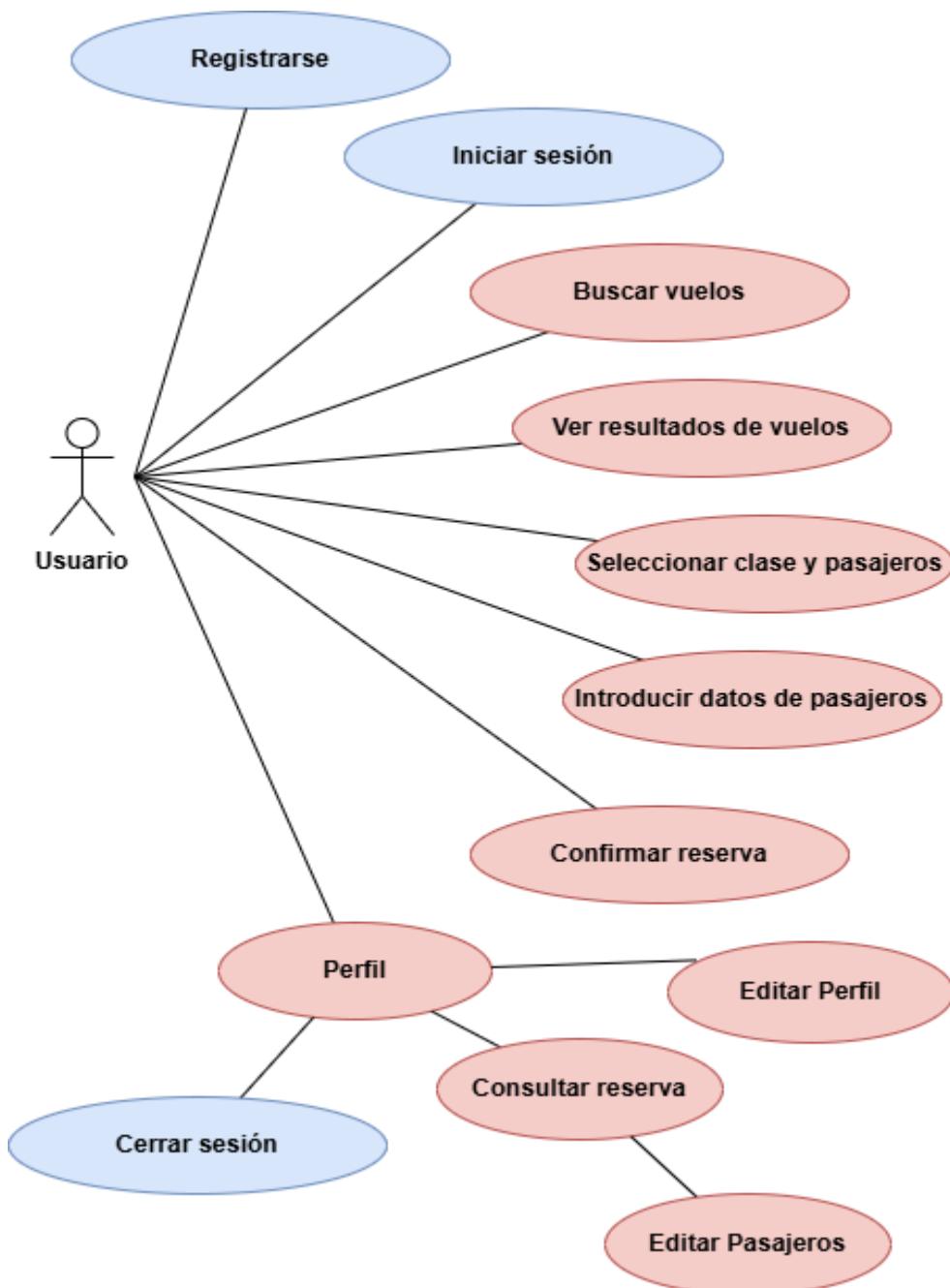
- Aplicación desarrollada exclusivamente para dispositivos Android.

● Mantenibilidad

- Estructura organizada en carpetas (screens, ViewModel, components, utils, etc.).
- Código siguiendo buenas prácticas de clean code.

Casos de Uso.

El diagrama de casos de uso de **Spicy Airlines** muestra las **funcionalidades** que el usuario puede realizar: registrarse, iniciar y cerrar sesión, buscar vuelos, ver resultados, seleccionar clase y pasajeros, introducir datos de pasajeros y confirmar reservas. Además, el usuario puede gestionar su perfil (editar perfil) y consultar o modificar reservas (editar pasajeros). Esto cubre el flujo completo de interacción del usuario con la aplicación.





4. OBJETIVOS

Los objetivos del proyecto SpicyAirlines se han definido para guiar el desarrollo de la aplicación de forma clara, medible y realista. Estos objetivos aseguran la entrega de una app funcional, conectada con Firebase y centrada en una buena experiencia de usuario, con una arquitectura modular y mantenible.

Objetivo general

Desarrollar una aplicación móvil Android, llamada **Spicy Airlines**, que permita a los usuarios registrarse, buscar vuelos entre Madrid y ciudades seleccionadas de China (Chongqing y Chengdu), seleccionar clase y número de pasajeros, introducir datos de los pasajeros, confirmar reservas, y consultar o modificar sus reservas, utilizando Firebase como gestor de base de datos.

Objetivos específicos y plazos estimados:

Objetivo	Plazo
Implementar un sistema de registro e inicio de sesión con Firebase Authentication.	1 semana
Configurar Firestore con las colecciones necesarias: usuarios, vuelos, reservas y pasajeros.	1 semana
Diseñar una interfaz de usuario moderna, clara y adaptable con Jetpack Compose y Material 3.	1 semana



Crear pantalla de búsqueda de vuelos con filtros de fechas, clase y número de pasajeros.	1 semana
Programar la lógica para mostrar resultados de vuelos según origen, destino y disponibilidad real de asientos.	1 semana
Permitir la selección de vuelos (ida o ida y vuelta) y mostrar información detallada (hora, precio, duración).	1 semana
Implementar la pantalla para introducir datos de los pasajeros con validaciones (pasaporte, fecha, etc.)	1 semana
Mostrar un resumen de reserva con todos los datos antes de confirmar (vuelos, pasajeros, precio total, clase).	1 semana
Confirmar y guardar la reserva en Firebase, actualizando también la disponibilidad de asientos.	1 semana
Crear un perfil de usuario que permita consultar reservas pasadas y futuras, editar datos y cerrar sesión.	1 semana
Mejorar la interfaz con iconos personalizados SVG, filtros de ordenación y mensajes visuales como "sin resultados".	1 semana

Tiempo total estimado: 11 semanas



5. DESARROLLO

5.1. METODOLOGÍA

Metodología aplicada: Scrum

Se ha utilizado una metodología ágil basada en **Scrum** debido a su capacidad para adaptarse a proyectos dinámicos como **Spicy Airlines**, donde los requisitos han ido evolucionando durante el desarrollo.

- **Flexibilidad ante cambios:** Durante la creación de la aplicación se han añadido mejoras según se detectaron necesidades, como el control de plazas disponibles por clase, filtros visuales, edición de datos o la separación de vuelos de ida y vuelta.
- **Desarrollo por fases:** Las funcionalidades principales, desde el inicio de sesión hasta la confirmación de reservas o la edición de pasajeros, se han desarrollado en fases cortas e independientes, lo que ha permitido centrarse en cada módulo sin perder de vista el producto completo.
- **Validación constante:** Tras implementar cada funcionalidad, se ha verificado su correcto funcionamiento directamente en Android Studio y Firebase, permitiendo detectar errores rápidamente y hacer ajustes inmediatos.
- **Enfoque educativo práctico:** Aunque el desarrollo se ha llevado a cabo individualmente, se ha simulado una dinámica ágil, con avances semanales, validaciones progresivas y documentación continua como si se estuviera trabajando con un cliente final o equipo.

Organización en Sprints

El desarrollo se ha organizado en **sprints semanales**, cada uno con una duración aproximada de una semana y enfocado en una funcionalidad principal de la aplicación:

- **Sprint 1:** Login y registro con Firebase Authentication
- **Sprint 2:** Pantalla de búsqueda de vuelos



- **Sprint 3:** Carga y visualización de resultados desde Firebase
- **Sprint 4:** Selección de clase y verificación de asientos disponibles
- **Sprint 5:** Formulario y validación de datos de los pasajeros
- **Sprint 6:** Confirmación de reserva y almacenamiento en Firestore
- **Sprint 7:** Pantalla de perfil y consulta de reservas
- **Sprint 8:** Edición de datos de pasajeros y del perfil de usuario
- **Sprint 9:** Mejoras visuales (iconos SVG, filtros, diseño responsive)
- **Sprint 10:** Integración final, pruebas globales y redacción de documentación

5.2. TECNOLOGÍAS

Para el desarrollo de la aplicación *Spicy Airlines*, se han seleccionado tecnologías modernas, eficientes y bien integradas dentro del ecosistema Android. La elección se ha basado en la compatibilidad, la facilidad de desarrollo y despliegue, y el soporte oficial de Google.

Tecnologías utilizadas

- **Lenguaje de programación: Kotlin**

Utilizado por ser el lenguaje oficial recomendado por Google para el desarrollo de aplicaciones Android. Ofrece una sintaxis moderna, mayor seguridad frente a errores y mejor compatibilidad con las bibliotecas actuales.

- **Framework de interfaz: Jetpack Compose**

Jetpack Compose ha sido la opción elegida para el diseño de interfaces. Su enfoque declarativo permite una construcción de UI más intuitiva y flexible, facilitando la creación de pantallas adaptativas y dinámicas.



- **Diseño visual: Material 3 (Google Material Design)**

Se ha seguido la guía de diseño Material Design 3 para garantizar una experiencia de usuario coherente, moderna y visualmente atractiva, utilizando colores temáticos, iconografía SVG y componentes estilizados.

- **Arquitectura: MVVM (Model-View-ViewModel)**

Se ha estructurado el proyecto utilizando el patrón MVVM, separando claramente la lógica de negocio, la interfaz y los datos, lo que facilita la mantenibilidad, pruebas y escalabilidad del sistema.

- **Plataforma backend: Firebase**

- **Firebase Authentication:** Para la autenticación de usuarios mediante correo electrónico y contraseña.
- **Cloud Firestore:** Utilizado como base de datos NoSQL en tiempo real para gestionar información de usuarios, vuelos, reservas y pasajeros.

- **IDE: Android Studio**

Entorno de desarrollo oficial para Android, con emuladores, herramientas de depuración, asistentes de diseño y compatibilidad nativa con Kotlin y Compose.

- **Control de versiones: Git + GitHub**

El código fuente del proyecto se ha versionado utilizando Git y alojado en un repositorio privado de GitHub, permitiendo un control riguroso de los cambios realizados durante el desarrollo.

- **Iconografía personalizada (SVG)**

Se han utilizado iconos en formato SVG personalizados (como iconos de vuelo, calendario, usuario, reloj, etc.) para mejorar la interfaz y reforzar la usabilidad visual.

Justificación

Las tecnologías seleccionadas garantizan un desarrollo eficiente, moderno y centrado en la experiencia de usuario. **Firebase** permite una integración sencilla del backend sin necesidad de servidores externos, **Jetpack Compose** facilita el diseño adaptativo de las pantallas, y el uso de **GitHub** asegura el control del proyecto y su trazabilidad.

5.3. PROTOTIPADO Y USABILIDAD

El prototipado y diseño de interfaz de usuario (**UI**) y experiencia de usuario (**UX**) son fundamentales en el desarrollo de aplicaciones móviles. En el proyecto **Spicy Airlines**, se han aplicado principios clave de UX/UI para garantizar una **interfaz intuitiva, accesible y visualmente atractiva**.

Principios de UX/UI Aplicados

A continuación, se presentan las principales leyes de UX/UI aplicadas en las pantallas de **SpicyAirlines**, junto con ejemplos que demuestran cómo se han implementado.

- **Ley de Hick-Hyman (Simplicidad y Decisión Rápida)**
 - Las pantallas de **Login** y **Registro** presentan opciones mínimas, evitando sobrecargar al usuario.
 - En la **Pantalla de Inicio**, solo se muestran opciones esenciales (origen, destino, fechas, pasajeros, clase), lo que simplifica la búsqueda.
- **Ley de Fitts (Accesibilidad de Elementos Interactivos)**
 - Los botones de acción (Buscar, Continuar, Confirmar) son grandes y accesibles.
 - Los vuelos en **ResultadosScreen** son clicables como tarjetas, facilitando la selección.



- **Ley de Proximidad (Organización Visual)**

- Los campos de texto están agrupados por categorías, como en **DatosPasajerosScreen** (nombre, apellidos, pasaporte).
- Las secciones de vuelos están claramente separadas en **ResultadosScreen**.

- **Ley de Miller (Memoria a Corto Plazo - 7 ± 2)**

- Las pantallas solo presentan información clave, como los datos de vuelo en **VueloSeleccionadoScreen**.
- Los resultados de vuelos están limitados a **información básica** (origen, destino, hora, precio).

- **Ley de Jakob (Familiaridad y Coherencia)**

- **Consistencia visual** con Material Design 3 en todas las pantallas.
- Iconos reutilizados (vuelo, calendario, reloj) para reforzar la familiaridad.

- **Ley de Tesler (Simplificación del Flujo)**

- La opción de "Solo ida" o "Ida y vuelta" en **InicioScreen** simplifica la búsqueda.
- En **ConfirmacionReservaScreen**, el resumen de reserva es claro y directo.

- **Ley de Pareto (80/20 - Priorizar Funcionalidades Clave)**

- En **PerfilScreen**, las acciones clave están claramente disponibles: ver reservas, editar perfil y cerrar sesión.

- **Ley de Control Percibido (Influir en los resultados de la interfaz)**

- Los botones muestran estados visuales claros (activo, inactivo, cargando).
- Mensajes de error específicos se presentan debajo de los campos afectados.

- **Ley de Visibilidad del Estado del Sistema (Informar al usuario del estado)**

- **Indicadores de carga** visibles en las pantallas.
- Los **mensajes de error** son claros y específicos, ayudando al usuario a entender qué ocurre.



Prototipo de SpicyAirlines

1. Pantalla de Inicio

- Botón de "Iniciar sesión"
- Botón de "Registrarse"

2. Pantalla de Inicio de Sesión

- Campo de correo electrónico y contraseña.
- Botón "Entrar".
- Mensajes de error claros si hay problemas (correo o contraseña incorrecta).

3. Pantalla de Registro

- Campos para ingresar correo electrónico, contraseña, nombre, apellidos, ciudad, provincia, código postal y teléfono.
- Validación de contraseña (nivel de seguridad visual).
- Mensajes de error específicos para cada campo.
- Botón "Registrarse".

4. Pantalla Principal

- Selección de origen y destino (Madrid, Chengdu, Chongqing).
- Selección de fechas (sólo ida o ida y vuelta).
- Selección de clase (Turista, Premium, Business).
- Selección de número de pasajeros (adultos y niños).
- Botón "Buscar vuelos".

5. Pantalla de Resultados

- Listado de vuelos disponibles (origen, destino, precio, duración).
- Filtro para ordenar por precio.



6. Pantalla de Selección de Vuelo

- Resumen del vuelo seleccionado (origen, destino, horario, precio).
- Selección de clase.
- Botón "Continuar".

7. Pantalla de Datos de Pasajeros

- Formulario para ingresar nombre, apellidos, pasaporte y teléfono de cada pasajero.
- Validación de campos.
- Botón "Confirmar y continuar".

8. Pantalla de Confirmación de Reserva

- Resumen de la reserva (vuelos seleccionados, clase, pasajeros).
- Precio total.
- Botón "Pagar".

9. Pantalla de Pago Completado

- Mensaje de éxito ("Tu reserva se ha completado").
- Botón "Volver al inicio".

10. Pantalla de Perfil

- Visualización de reservas pasadas y futuras.
- Opción de editar perfil y cerrar sesión.

11. Pantalla de Edición de Perfil

- Formulario para actualizar el perfil (nombre, ciudad, teléfono, etc.)
- Botón "Guardar cambios".

12. Pantalla Editar Pasajeros.

- Formulario editable para nombre, apellidos, pasaporte y teléfono de cada pasajero.
- Validación de campos obligatorios.
- Botón "Guardar cambios".
- Mensaje de confirmación tras guardar.



1 - Pantallainicio

2 - Login

3 - Register

4 - Principal

5 - Resultados

6 - VueloSeleccionado



7 - Pasajeros

SpicyAirlines

Nombre
Apellidos
Teléfono
Pasaporte
Fecha Nacimiento

Pasajero 1

Continuar

8 - ConfirmarReserva

SpicyAirlines

Vuelo
Info...

Pasajeros
David Martín
Pasaporte:

Pagar

9 - PagoCompletado

SpicyAirlines

Reserva Completada
¡Gracias!

Continuar

10 - Perfil

SpicyAirlines

Cerrar sesión
Editar Perfil
Reservas

Reserva 03/05/2024
Reserva 09/11/2025

11 - EditarPerfil

SpicyAirlines

Editar Perfil

Email
Nombre
Apellidos
Ciudad
Provincia
Código Postal
Teléfono
Nueva contraseña

Registrar

12 - EditarPasajeros

SpicyAirlines

Editar Pasajeros

Vuelo
Info...

Pasajero 1

Nombre
Apellidos
Teléfono
Pasaporte
Fecha Nacimiento

Guardar



5.4. MODELADO DE DATOS

Especificación de Requisitos de la Base de Datos

La base de datos de **Spicy Airlines** debe cumplir con los siguientes requisitos:

- Permitir almacenar información de usuarios registrados.
- Gestionar información de vuelos disponibles, incluyendo destinos, horarios y precios.
- Almacenar reservas de vuelos, asociadas a usuarios y vinculadas a uno o varios vuelos.
- Almacenar información de los pasajeros en cada reserva.

Modelo de Datos (Firestore - NoSQL)

Estructura General:

- **Colección: Usuarios**
 - **Documento (ID):** UID del usuario (Firebase Auth)
 - **Campos:**
 - Nombre
 - Apellidos
 - Ciudad
 - Provincia
 - Código Postal
 - Teléfono
 - Correo Electrónico
- **Colección: Vuelos**
 - **Documento (ID):** Identificador único del vuelo
 - **Campos:**
 - Origen
 - Destino



- Fecha de salida (Timestamp)
- Fecha de llegada (Timestamp)
- Clase (Turista, Premium, Business)
- Precio
- Asientos disponibles por clase (Turista, Premium, Business)

- Colección: Reservas

- Documento (ID): Identificador único de la reserva

- Campos:

- ID del Usuario (UID)
 - Fecha de reserva (Timestamp)
 - Total Pasajeros (Adultos + niños)
 - Precio Total
 - Clase seleccionada
 - Vuelos (Array de IDs de vuelos)
 - Subcolección: Pasajeros

- Campos:

- Nombre
 - Apellidos
 - Número de Pasaporte
 - Fecha de Nacimiento
 - Teléfono

Diagrama E/R (Entidad-Relación)

Dado que **Firestore** es una **base de datos NoSQL**, su estructura es jerárquica y orientada a documentos. Sin embargo, podemos representarla visualmente con un diagrama E/R adaptado:

- **Usuarios** (Colección)

↳ Relación uno a muchos con **Reservas**



- **Reservas** (Colección)

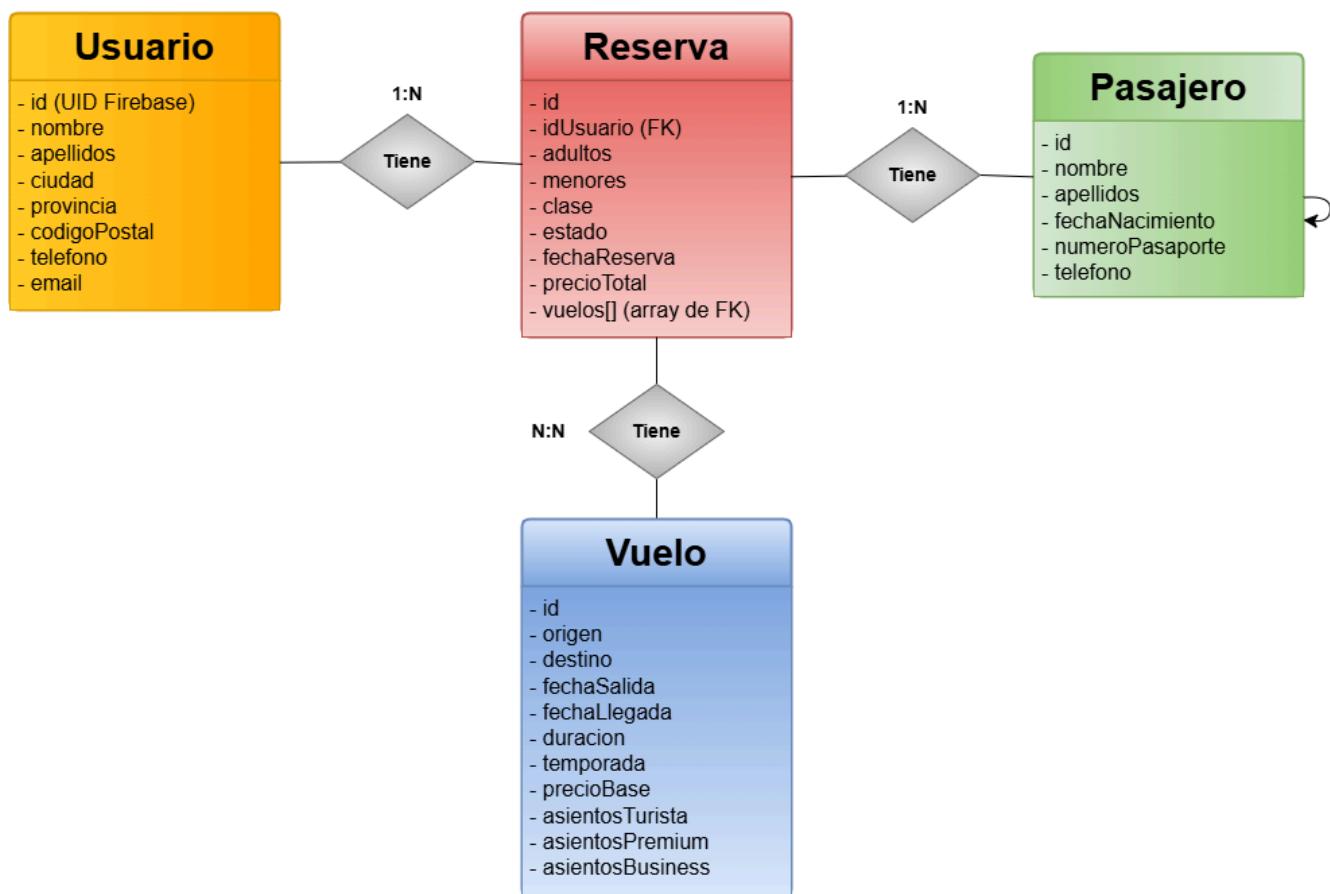
- ↳ Contiene subcolección **Pasajeros**
- ↳ Relación muchos a muchos con **Vuelos**

- **Vuelos** (Colección)

- ↳ Relación muchos a muchos con **Reservas**

- **Pasajero** (Subcolección)

- ↳ Relación uno a muchos con **Reservas**
- ↳ No está relacionado con ninguna otra entidad más allá de su relación con Reserva y no forma parte de otra relación externa



Para subir datos masivos a las colecciones de Firebase Firestore en **Spicy Airlines**, he utilizado archivos **.JSON** que contienen los datos de vuelos y un script en JavaScript (**Node.js**) para cargarlos. Este proceso permite gestionar grandes volúmenes de datos de forma eficiente, ya que la colección "Vuelos" necesita bastantes datos.



5.5. SEGURIDAD Y AUTENTICACIÓN

La aplicación *Spicy Airlines* utiliza **Firebase Authentication** como sistema de autenticación de usuarios. Esta tecnología permite que los usuarios se registren e inicien sesión de manera segura utilizando **correo electrónico y contraseña**, sin necesidad de implementar un sistema propio de control de accesos.

Una vez que el usuario inicia sesión correctamente, **Firebase proporciona un propio identificador único (UID)**, que se usa internamente en la aplicación para relacionar al usuario con sus datos, como sus reservas o perfil personal. Esta identificación permite mantener una **asociación consistente y segura** entre el usuario que ha sido autenticado y la información almacenada en la base de datos.

El acceso a funcionalidades protegidas (como ver o editar reservas) dentro de la app se controla **a nivel de interfaz**, lo que permite que el usuario sólo vea o modifique sus propios datos según el UID.

Base de datos: permisos y roles

Actualmente, durante la fase de desarrollo del proyecto, se han mantenido activas las reglas de seguridad por defecto de Firebase Firestore, que permiten lectura y escritura sin restricciones:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if true;
    }
  }
}
```

Esta configuración abierta ha facilitado las pruebas y el desarrollo continuo de la aplicación sin bloqueos inesperados por parte del sistema de seguridad. Sin embargo, cuando la aplicación vaya a pasar a un entorno de producción, estas reglas deben ser reemplazadas por un sistema más restrictivo.



Aunque las reglas están abiertas, en el lado del cliente (app Android) se aplican controles de autenticación y lógica condicional basada en el usuario autenticado (**auth.currentUser?.uid**) para evitar que se muestren datos incorrectos u de otros usuarios.

Justificación de tecnologías

- **Firebase Authentication:** nos va a ofrecer una solución segura y lista para usar, que simplifica el proceso de autenticación en Android. Proporciona almacenamiento cifrado de contraseñas, verificación de usuarios y fácil integración con Firebase Firestore.
- **Firestore:** permite organizar y consultar documentos estructurados, y además, su sistema de seguridad basado en reglas permite una escalabilidad segura.
- **Jetpack Compose y ViewModel:** gestionan el estado del usuario de forma segura y desacoplada, evitando fugas de datos entre pantallas o pérdidas de sesión.

En resumen, la aplicación implementa un sistema de autenticación seguro a través de **Firebase**, garantizando que cada usuario se identifique correctamente y acceda a funciones personalizadas. Aunque en esta fase de desarrollo las reglas de la base de datos están abiertas para agilizar las pruebas, la estructura del proyecto está preparada para aplicar restricciones en el futuro y cumplir con buenas prácticas de seguridad y privacidad.

5.6. DISEÑO E IMPLEMENTACIÓN

Principios de Diseño

En la implementación de **Spicy Airlines**, se han aplicado los principios fundamentales del diseño orientado a objetos:



- **Encapsulación:** Cada clase es capaz de gestionar su propio estado y funcionalidad, esto asegura una mayor protección de los datos, ya que sólo se pueden acceder por métodos concretos.

- Ejemplo: **ValidaciónPasajero.**

Esto logra que para editar un pasajero se deba acceder a esta función, y así validar los campos, devolviendo siempre un error si no cumple con las condiciones que le hayamos impuesto.

```
// Función que valida los campos de un pasajero
fun validarPasajero(pasajero: Pasajero): ValidacionPasajero {
    // Verifica que el nombre no esté en blanco
    if (pasajero.nombre.isBlank()) {
        return ValidacionPasajero( esValido: false, mensajeError: "Nombre inválido")
    }

    // Verifica que los apellidos no estén en blanco
    if (pasajero.apellidos.isBlank()) {
        return ValidacionPasajero( esValido: false, mensajeError: "Apellidos inválidos")
    }

    // Verifica que el número de pasaporte tenga el formato correcto (3 letras + 6 dígitos)
    val pasaporteRegex = Regex(pattern: "[A-Z]{3}\\d{6}")
    if (!pasaporteRegex.matches(pasajero.numeroPasaporte)) {
        return ValidacionPasajero( esValido: false, mensajeError: "Número de pasaporte inválido")
    }
}
```

- Ejemplo: **HoraUTC.**

La clase HoraUTC en Spicy Airlines aplica encapsulación porque toda la lógica de conversión de fechas y horas a hora local está centralizada en el método formatearFechaHoraLocal. La lógica está protegida y solo se puede utilizar a través del método público, lo que asegura que cualquier cambio en la conversión sólo debe hacerse aquí. Además, al estar organizada en un objeto independiente, es fácilmente reutilizable en toda la aplicación.



```
// Objeto utilitario para manejar la conversión y formato de fechas/hora UTC a hora local
object HoraUTC {

    // Función para formatear una fecha/hora (Timestamp) a la hora local de la ciudad especificada
    fun formatearFechaHoraLocal(timestamp: Timestamp, ciudad: String): String {
        // Determina la zona horaria según la ciudad proporcionada
        val zona = when (ciudad) {
            "Madrid" -> "Europe/Madrid"
            "Chongqing", "Chengdu" -> "Asia/Shanghai"
            else -> "UTC"
        }

        // Convierte el Timestamp (Firebase) a Instant
        val instant = timestamp.toDate().toInstant()
        // Formatea la fecha y hora según el formato especificado y la zona horaria
        val formatter = DateTimeFormatter.ofPattern(pattern: "dd/MM/yyyy HH:mm")
            .withZone(ZoneId.of(zona))

        // Devuelve la fecha/hora formateada
        return formatter.format(instant)
    }
}
```

- **Herencia:** Es lo que permite que una clase hija herede atributos y comportamientos de su clase madre.

- Ejemplo: **ResultadosViewModel**.

Todos los ViewModel de la app heredan de la clase ViewModel que proporciona el propio Jetpack Compose. Esto aporta lógica a todos los demás ViewModel para que gestionen el estado y su ciclo de vida en la app, usando StateFlow o MutableStateFlow.

```
// ViewModel para gestionar los resultados de búsqueda de vuelos
class ResultadosViewModel : ViewModel()
```

Esto permite que ResultadosViewModel tenga acceso a:

- Gestión de ciclo de vida.
- Acceso al viewModelScope para ejecutar corutinas de manera segura.



- Persistencia de los datos mientras la pantalla está activa.
 - Gracias a esta herencia, **ResultadosViewModel** puede añadir métodos específicos como **cargarVuelos()** para manejar datos concretos de la aplicación sin perder las funcionalidades básicas que provee la clase base.
 - Nos ayuda a tener una app con modularidad y que pueda ser mejor mantenida y se pueda extender fácilmente.
-
- **Polimorfismo:** Permite que un mismo método o función pueda comportarse de diferentes maneras dependiendo del objeto o clase que lo esté utilizando.
 - Ejemplo: **ResultadosScreen**.
 - Polimorfismo en las funciones VueloCard y VueloCombinadoCard, VueloCard muestra información de un vuelo único (solo ida). Mientras que VueloCombinadoCard muestra información de una combinación de vuelos (ida y vuelta).
 - Aunque ambas funciones tienen nombres diferentes, el polimorfismo se aplica porque:
 - Comparten un propósito común: mostrar información de vuelos.
 - Su comportamiento varía dependiendo del tipo de vuelo (individual o combinado).
 - Se llaman de manera similar, pero muestran contenido distinto.



```
// Función para mostrar vuelos de ida
@Composable
fun VueloCard(vuelo: Vuelo, sharedViewModel: SharedViewModel, onClick: () -> Unit) {
    val clase by sharedViewModel.claseSeleccionada.collectAsState()
    val multiplicador = when (clase) {
        "Premium" -> 1.5
        "Business" -> 2.0
        else -> 1.0
    }
    val precioPorPasajero = vuelo.precioBase * multiplicador

    var isPressed by remember { mutableStateOf( value: false) }
    val elevation by animateDpAsState(targetValue = if (isPressed) 10.dp else 6.dp)
    val backgroundColor by animateColorAsState(
        targetValue = if (isPressed) Color( color: 0xFFFFC107) else colorPorTemporada(vuelo.temporada)
    )
}
```

```
// Función para mostrar combinaciones de vuelos ida y vuelta
@Composable
fun VueloCombinadoCard(ida: Vuelo, vuelta: Vuelo, sharedViewModel: SharedViewModel, onClick: () -> Unit) {
    val clase by sharedViewModel.claseSeleccionada.collectAsState()
    val multiplicador = when (clase) {
        "Premium" -> 1.5
        "Business" -> 2.0
        else -> 1.0
    }
    val precioPorPasajero = (ida.precioBase + vuelta.precioBase) * multiplicador

    var isPressed by remember { mutableStateOf( value: false) }
    val elevation by animateDpAsState(targetValue = if (isPressed) 10.dp else 6.dp)
    val backgroundColor by animateColorAsState(
        targetValue = if (isPressed) Color( color: 0xFFFFC107) else colorPorTemporada(ida.temporada)
    )
}
```

Diagrama de Clases

1. Modelos de Datos (Clases Principales)

- **Usuario:** Representa la información del usuario autenticado.
- **Vuelo:** Almacena los detalles de cada vuelo disponible (ida o vuelta).
- **Reserva:** Almacena la información de cada reserva realizada por los usuarios.
- **Pasajero:** Contiene los datos de cada pasajero que forma parte de una reserva.



2. ViewModels (Gestión de Estado y Lógica de Negocio)

- **RegisterViewModel.**

- Métodos: **onEmailChange()**, **onPasswordChange()**, **register()**

register(): Verifica que todos los campos del formulario sean válidos y luego crea un nuevo usuario en Firebase Authentication. Si el registro es exitoso, guarda los datos del usuario en Firestore.

```
fun register(onSuccess: () -> Unit) {
    _error.value = null

    // Revisión rápida de errores individuales antes de registrar
    if (listOf(
        _errorNombre.value,
        _errorApellidos.value,
        _errorCiudad.value,
        _errorProvincia.value,
        _errorCodigoPostal.value,
        _errorTelefono.value,
        _errorPassword.value
    ).any { it != null })
    {
        _error.value = "Corrige los errores antes de continuar"
        return
    }
}
```



```
// Intento de registro en Firebase Authentication
auth.createUserWithEmailAndPassword(_email.value, _password.value)
    .addOnSuccessListener { authResult ->
        val uid = authResult.user?.uid
        if (uid != null) {
            val usuario = Usuario(
                email = _email.value,
                nombre = _nombre.value,
                apellidos = _apellidos.value,
                ciudad = _ciudad.value,
                provincia = _provincia.value,
                codigoPostal = _codigoPostal.value,
                telefono = _telefono.value
            )

            // Guardar usuario en Firestore
            firestore.collection(collectionPath: "usuarios")
                .document(uid)
                .set(usuario)
                .addOnSuccessListener { onSuccess() }
                .addOnFailureListener {
                    _error.value = "Error al guardar tus datos: ${it.message}"
                }
        } else {
            _error.value = "No se pudo obtener el ID del usuario"
        }
    }
    .addOnFailureListener {
        val mensaje = when {
            it.message?.contains( other: "email address is already in use") == true -> "Este correo ya está registrado."
            it.message?.contains( other: "badly formatted") == true -> "Formato de correo inválido."
            it.message?.contains( other: "Password should be at least") == true -> "La contraseña debe tener al menos 6 caracteres"
            else -> "Error al registrarse: ${it.message}"
        }
        _error.value = mensaje
    }
}
```

- **ResultadosViewModel.**

- Métodos: **cargarVuelos()**, **ordenarVuelosIda()**, **vueloTieneAsientos()**.

vueloTieneAsientos(): Aquí verificamos que haya asientos disponibles, compara si los asientos son mayor o igual a la cantidad de pasajeros seleccionados

```
// Verifica si un vuelo tiene suficientes asientos disponibles
private fun vueloTieneAsientos(vuelo: Vuelo, clase: String, pasajeros: Int): Boolean {
    return when (clase) {
        "Turista" -> vuelo.asientosTurista >= pasajeros
        "Premium" -> vuelo.asientosPremium >= pasajeros
        "Business" -> vuelo.asientosBusiness >= pasajeros
        else -> false
    }
}
```



- **SharedViewModel.**

- Métodos: **seleccionarVuelo()**, **seleccionarClase()**, **calcularPrecioBillete()**

Este último nos va a crear una condición por la cuál dependiendo de la clase seleccionada, el vuelo aumentará de precio.

```
// Calcular el precio por billete según la clase seleccionada
fun calcularPrecioBillete(vueloIda: Vuelo, vueloVuelta: Vuelo? = null) {
    val clase = _claseSeleccionada.value
    val multiplicador = when (clase) {
        "Premium" -> 1.5
        "Business" -> 2.0
        else -> 1.0
    }

    val precioBase = vueloIda.precioBase + (vueloVuelta?.precioBase ?: 0)
    val precioUnitario = precioBase * multiplicador

    _precioTotal.value = precioUnitario
}
```

- **DatosPasajerosViewModel.**

- Métodos: **inicializarFormularios()**, **validarTodosLosPasajeros()**, **calcularEdad()**.

Gracias a **calcularEdad()**, somos capaces de validar que un pasajero sea menor o mayor de 3 años.

```
// Calcula la edad de un pasajero a partir de su fecha de nacimiento
private fun calcularEdad(pasajero: Pasajero): Int {
    val hoy = Calendar.getInstance()
    val nacimiento = Calendar.getInstance().apply {
        time = pasajero.fechaNacimiento.toDate()
    }
    var edad = hoy.get(Calendar.YEAR) - nacimiento.get(Calendar.YEAR)
    if (hoy.get(Calendar.DAY_OF_YEAR) < nacimiento.get(Calendar.DAY_OF_YEAR)) edad--
    return edad
}
```



Esto lo usamos en **validarTodosLosPasajeros()**, para comprobar que cumplen con la edad seleccionada.

```
// Valida todos los pasajeros y verifica sus datos
fun validarTodosLosPasajeros(): Boolean {
    val erroresList = _pasajeros.value.mapIndexed { index, pasajero ->
        when {
            pasajero.nombre.isBlank() -> "Error: Nombre no válido."
            pasajero.apellidos.isBlank() -> "Error: Apellidos no válidos."
            !Regex(pattern: "[A-Z]{3}\\d{6}").matches(pasajero.numeroPasaporte) -> "Error: Pasaporte debe tener 3 letras y 6 dígitos."
            pasajero.telefono.length != 9 || !pasajero.telefono.all { it.isDigit() } -> "Error: Teléfono debe tener 9 dígitos."
            calcularEdad(pasajero) < 3 && index < adultosEsperados -> "Error: Edad inválida para un adulto."
            calcularEdad(pasajero) >= 3 && index >= adultosEsperados -> "Error: Edad inválida para un niño."
            else -> null
        }
    }

    _errores.value = erroresList
    return erroresList.all { it == null }
}
```

- **ConfirmacionReservaViewModel.**

- Métodos: **guardarReservaFirebase()**, **actualizarAsientos()**

Con este último comprobamos la disponibilidad de los asientos a la hora de mostrar un vuelo.

```
// Función para actualizar el número de asientos disponibles en el vuelo
private fun actualizarAsientos(
    vuelo: Vuelo,
    clase: String,
    cantidad: Int,
    onError: (Exception) -> Unit
) {
    val vueloRef = db.collection(collectionPath: "Vuelos").document(vuelo.id)

    db.runTransaction { transaction ->
        val snapshot = transaction.get(vueloRef)
        val campo = when (clase) {
            "Business" -> "asientosBusiness"
            "Premium" -> "asientosPremium"
            else -> "asientosTurista"
        }

        val actuales = snapshot.getLong(campo) ?: 0
        transaction.update(vueloRef, campo, value: actuales - cantidad)
    }.addOnFailureListener(onError)
}
```



3. Clases Utilitarias (Lógica Adicional)

- **ValidacionUsuario y ValidacionPasajero.**

- Métodos: **validarCamposUsuario()**, **validarPasajero()**

En **ValidacionUsuario()** que verifique que se han completado los campos obligatorios.

```
// Verifica que todos los campos obligatorios no estén en blanco
if (nombre.isBlank() || apellidos.isBlank() || ciudad.isBlank() || provincia.isBlank()) {
    return ValidacionUsuario( esValido: false, mensajeError: "Todos los campos son obligatorios")
}
```

- **PasswordSecurity**

- Enum: **NivelSeguridadContrasena** (MUY_DEBIL, DEBIL, NORMAL, BUENA, EXCELENTE)

```
// Enumeración que define los diferentes niveles de seguridad de una contraseña
enum class NivelSeguridadContrasena{
    val descripcion: String, // Descripción del nivel de seguridad
    val color: Color, // Color asociado al nivel para la interfaz de usuario
    val valor: Float // Valor numérico
} {
    MUY_DEBIL( descripcion: "Muy débil", Color( color: 0xFFFF44336), valor: 0.1f),
    DEBIL( descripcion: "Débil", Color( color: 0xFFFF9800), valor: 0.3f),
    NORMAL( descripcion: "Normal", Color( color: 0xFFFFEB3B), valor: 0.5f),
    BUENA( descripcion: "Buena", Color( color: 0xFF7EAF4C), valor: 0.75f),
    EXCELENTE( descripcion: "Excelente", Color( color: 0xFF2E7D32), valor: 1f)
}
```

Diagrama de clases: archivo SpicyAirlinesDiagram.png



5.7. PRUEBAS Y CALIDAD DEL SOFTWARE

Para garantizar la calidad del software desarrollado, se han implementado múltiples test de forma automatizada, incluyendo **pruebas unitarias, de integración, de interfaz de usuario (UI) y de rendimiento**. Además, se ha seguido una filosofía de desarrollo basada en **Clean Code**, aplicando buenas prácticas.

Tecnologías y herramientas implementadas:

- **JUnit 4:** Framework principal para ejecutar pruebas unitarias y de integración.
- **MockK:** Para simular comportamientos de servicios como FirebaseAuth y Firestore.
- **Jetpack Compose UI Test:** Para pruebas de interfaz de usuario en Compose.
- **InstantTaskExecutorRule:** Para testear LiveData y Flow de forma síncrona.
- **measureTimeMillis:** Para comprobar el rendimiento de carga de pantallas y datos.

Pruebas realizadas:

1. Pruebas unitarias del ViewModel:

- Verifican si el LoginViewModel responde correctamente a casos de éxito y error en el login.
- Se simulan usuarios reales mediante mock de Firebase.
- Se validan valores como usuario, error, isLoading.

Ejemplo: **LoginViewModelTest.kt** y **LoginViewModelTest_Error.kt**

Resultado esperado: el login correcto carga el usuario, el incorrecto muestra mensaje de error.



```
// Configurar Firebase Auth para devolver el usuario
val mockAuthTask = mockk<com.google.android.gms.tasks.Task<AuthResult>>(relaxed = true)
every { auth.signInWithEmailAndPassword(any(), any()) } returns mockAuthTask

// Configurar el listener de éxito de autenticación
every { mockAuthTask.addOnSuccessListener(any<OnSuccessListener<AuthResult>>()) } answers {
    firstArg<OnSuccessListener<AuthResult>>().onSuccess(mockAuthResult)
    mockAuthTask
}
```

```
// EjecutarA el login
viewModel.login( email: "hola22@gmail.com", password: "123456" ) {}

// Verificar que se cargó correctamente el usuario
assertNull(viewModel.error.value)
assertFalse(viewModel.isLoading.value)
assertNotNull(viewModel.usuario.value)
assertEquals( expected: "Test User", viewModel.usuario.value?.nombre)

println("Éxito: Test de login correcto.")
```

```
// Test que verifica que el login falla cuando las credenciales son incorrectas.
@Test
fun `login fails with incorrect credentials`() = runTest {
    // Intentamos iniciar sesión con credenciales incorrectas.
    viewModel.login( email: "wrong@example.com", password: "wrongpassword" )

    // Esperamos a que se complete cualquier tarea pendiente.
    advanceUntilIdle()

    // Verificamos que haya un mensaje de error.
    assertNotNull(viewModel.error.value)
    assertEquals( expected: "Correo o contraseña incorrectos.", viewModel.error.value )

    // Mensaje de éxito en consola si el test pasa correctamente.
    println(" ✅ Éxito: Test de login fallido completado correctamente. Las credenciales incorrectas fueron detectadas")
}
```

2. Pruebas de integración:

- Combinan la autenticación con la carga de datos del usuario desde Firestore.
- Se simula el flujo completo de login en Firebase.

Ejemplo: **LoginIntegrationTest.kt**

Resultado esperado: usuario.value.nombre == "Test User" al loguearse correctamente.



- **Simulación de Firebase:** En esta primera parte se simula un inicio de sesión exitoso usando FirebaseAuth. Se mockea la tarea de autenticación (Task<AuthResult>) y se fuerza el callback onSuccess para que devuelva un resultado simulado.

```
// Mock Firebase Auth
val mockAuthTask = mockk<com.google.android.gms.tasks.Task<AuthResult>>(relaxed = true)
val mockAuthResult = mockk<AuthResult>(relaxed = true)
val mockFirebaseUser = mockk<FirebaseUser>(relaxed = true)

// Configurar autenticación exitosa
every { auth.signInWithEmailAndPassword( email: "hola22@gmail.com", password: "123456" ) } returns mockAuthTask
every { mockAuthTask.addOnSuccessListener(any()) } answers {
    val successListener = it.invocation.args[0] as com.google.android.gms.tasks.OnSuccessListener<AuthResult>
    successListener.onSuccess(mockAuthResult)
    mockAuthTask
}
```

- **Simulación de usuario autenticado:** Tras simular el éxito en el login, se devuelve un FirebaseAuthUser simulado con UID "mockUid", que se utilizará para buscar al usuario en la base de datos.

```
every { mockAuthResult.user } returns mockFirebaseUser
every { mockFirebaseUser.uid } returns "mockUid"
```

- **Simulación de Firestore:** En este bloque se mockea la lectura del documento Firestore asociado al UID del usuario. Se simula también que esta lectura es exitosa llamando a onSuccess con un DocumentSnapshot falso.

```
// Mock Firestore (Carga del usuario)
val mockDocumentTask = mockk<com.google.android.gms.tasks.Task<DocumentSnapshot>>(relaxed = true)
val mockDocumentSnapshot = mockk<DocumentSnapshot>(relaxed = true)

every {
    firestore.collection( collectionPath: "usuarios" ).document( documentPath: "mockUid" ).get()
} returns mockDocumentTask

every { mockDocumentTask.addOnSuccessListener(any()) } answers {
    val successListener = it.invocation.args[0] as com.google.android.gms.tasks.OnSuccessListener<DocumentSnapshot>
    successListener.onSuccess(mockDocumentSnapshot)
    mockDocumentTask
}
```



- **Conversión a objeto y verificación:** Finalmente, se convierte el DocumentSnapshot en una instancia de Usuario, y se verifica que el nombre obtenido desde Firestore es correctamente cargado en el ViewModel.

```
every { mockDocumentSnapshot.toObject(Usuario::class.java) } returns mockUser

// Ejecutar el login
viewModel.login(email: "hola22@gmail.com", password: "123456") {}

// Verificar que el usuario se cargó correctamente
assertNull(viewModel.error.value)
assertFalse(viewModel.isLoading.value)
assertNotNull(viewModel.usuario.value)
assertEquals(expected: "Test User", viewModel.usuario.value?.nombre)

println("Éxito: Test de integración de login completado correctamente.")
```

3. Pruebas de UI (Interfaz de Usuario):

- Se comprueba visualmente la existencia de los elementos principales en pantallas como LoginScreen e InicioScreen.
- Se testea la interacción con los campos y botones, y la visibilidad de errores.

Ejemplos:

- **LoginScreenTest.kt:** Verifica los componentes y funcionamiento del botón “Entrar”.
- **PantallainicioScreenTest.kt:** Mide el tiempo de renderizado de la pantalla inicial. Se utiliza measureTimeMillis para asegurar que los elementos clave, como el botón 'Iniciar sesión',



están visibles en menos de 1000 ms.

```
@Test
fun pantallaInicio_cargaRápidamente() {
    // Medir solo el tiempo de renderizado de la pantalla
    composeTestRule.setContent {
        PantallaInicioAuth(onLoginClick = {}, onRegisterClick = {})
    }

    // Esperamos que el botón esté presente (la pantalla está lista)
    composeTestRule.onNodeWithText(text: "Iniciar sesión").assertExists()

    // Medimos el tiempo de renderizado
    val tiempoCarga = measureTimeMillis {
        composeTestRule.onNodeWithText(text: "Iniciar sesión").assertExists()
    }

    println("Tiempo de carga de contenido: \$tiempoCarga ms")

    // Verificamos que el tiempo de carga sea menor a 1000 ms (ajustable)
    assert(value: tiempoCarga < 1000) { "Error: La pantalla tardó demasiado en cargarse (\$tiempoCarga ms)" }

    println("Éxito: Pantalla de inicio cargada en \$tiempoCarga ms")
}
```

4. Pruebas de rendimiento:

- Se evalúa que la carga real de vuelos en ResultadosScreen no supere los 15 segundos.
- Se usa measureTimeMillis y withTimeoutOrNull para validar el rendimiento.

Ejemplo: **ResultadosScreenPerformanceTest.kt**

Resultado: Carga completada correctamente en menos de 15 segundos.

Cargamos un vuelo de prueba y medimos los segundos que tarda, en este caso en milisegundos



```
// Test que mide el tiempo de carga de resultados de vuelos, asegurando que sea menor a 15 segundos.
@Test
fun cargaResultadosReal_enMenosDe15Segundos() = runBlocking {
    try {
        // Creamos una instancia del ViewModel de Resultados.
        val viewModel = ResultadosViewModel()

        var tiempoCarga = 0L // Variable para almacenar el tiempo de carga.

        // Medimos el tiempo que tarda en cargar los vuelos.
        tiempoCarga = measureTimeMillis {
            // Llamamos al metodo de carga de vuelos con parametros de prueba.
            viewModel.cargarVuelos(
                origen = "Madrid",
                destino = "Chengdu",
                fechaIda = Timestamp.now(),
                fechaVuelta = Timestamp.now().toDate().time.plus( other: 1000 * 60 * 60 * 24 * 2
                    .let { Timestamp(seconds: it / 1000, nanoseconds: 0) }, // Fecha de vuelta 2 dias despues.
                claseSeleccionada = "Turista", // Clase de vuelo seleccionada.
                totalPasajeros = 1 // Total de pasajeros (1).
            )

            // Esperamos a que la carga se complete, con un tiempo maximo de 15 segundos.
            withTimeoutOrNull( timeMillis: 15000 ) {
                viewModel.cargaCompletada.first { it } // Esperamos hasta que se complete la carga.
            }
        }
    }
}
```

Ahora ya verificamos que el tiempo haya sido menor o igual a 15 segundos:

```
// Verificamos si el tiempo de carga es menor o igual a 15 segundos.
if (tiempoCarga <= 15000) {
    Log.d(TAG, msg: "Exito: Los resultados se cargaron en ${tiempoCarga} ms")
    System.out.println("Exito: Los resultados se cargaron en ${tiempoCarga} ms")
} else {
    Log.d(TAG, msg: "Error: La carga de resultados tardó demasiado (${tiempoCarga} ms)")
    System.out.println("Error: La carga de resultados tardó demasiado (${tiempoCarga} ms)")
    assert( value: false ) { "Error: La carga de resultados tardó demasiado (${tiempoCarga} ms)" }
}
} catch (e: Exception) {
    // En caso de que ocurra alguna excepcion durante el test.
    Log.e(TAG, msg: "Excepción durante el test: ${e.message}")
    System.out.println("Excepción durante el test: ${e.message}")
    assert( value: false ) { "Excepción durante el test: ${e.message}" }
}
```

5. Pruebas del filtrado y ordenación de vuelos:

- Se ha verificado la correcta ordenación de vuelos por precio y generación de combinaciones ida-vuelta.
- Se implementaron tests en FakeResultadosViewModel.



Ejemplo: ResultadosViewModelTest.kt

Resultado esperado: orden correcto de precios y combinaciones válidas entre vuelos.

Cargamos la lista de vuelos y creamos el test de ordenación por precio en vuelos de ida. Se verifica que los vuelos se ordenan correctamente de menor a mayor.

```
// Test que verifica que los vuelos de ida se cargan y se ordenan correctamente de menor a mayor precio.
@Test
fun `cargar vuelos ida y ordenar de menor a mayor`() = runTest {
    // Lista de vuelos de prueba.
    val vuelos = listOf(
        Vuelo(id = "1", origen = "Madrid", destino = "Chengdu", precioBase = 200),
        Vuelo(id = "2", origen = "Madrid", destino = "Chengdu", precioBase = 100),
        Vuelo(id = "3", origen = "Madrid", destino = "Chengdu", precioBase = 150)
    )

    // Cargamos los vuelos en el ViewModel y los ordenamos.
    viewModel.cargarVuelosFake(vuelos)
    viewModel.ordenarVuelos(orden: "Menor a mayor")

    // Verificamos que están ordenados correctamente.
    val ordenados = viewModel.vuelosIda.value
    assertEquals(expected: 100, ordenados[0].precioBase)
    assertEquals(expected: 150, ordenados[1].precioBase)
    assertEquals(expected: 200, ordenados[2].precioBase)

    // Mensaje de éxito si pasa el test.
    println("Éxito: Test 'cargar vuelos ida y ordenar de menor a mayor' completado correctamente.")
}
```

Con el siguiente test verificamos la generación correcta de una combinación de vuelo ida-vuelta.

```
@Test
fun `cargar vuelos ida y vuelta y generar combinaciones`() = runTest {
    // Listas de vuelos de ida y vuelta.
    val vuelosIda = listOf(Vuelo(id = "1", origen = "Madrid", destino = "Chengdu", precioBase = 200))
    val vuelosVuelta = listOf(Vuelo(id = "2", origen = "Chengdu", destino = "Madrid", precioBase = 150))

    // Cargamos los vuelos y generamos combinaciones.
    viewModel.cargarVuelosFake(vuelosIda, vuelosVuelta)
    val combinaciones = viewModel.combinacionesValidas.value

    // Verificamos que se ha generado correctamente la combinación.
    assertEquals(expected: 1, combinaciones.size)
    assertEquals(expected: "Madrid", combinaciones[0].first.origen)
    assertEquals(expected: "Chengdu", combinaciones[0].first.destino)
    assertEquals(expected: "Chengdu", combinaciones[0].second.origen)
    assertEquals(expected: "Madrid", combinaciones[0].second.destino)

    // Mensaje de éxito si pasa el test.
    println("Éxito: Test 'cargar vuelos ida y vuelta y generar combinaciones' completado correctamente.")
}
```

Por último, hacemos que se aplique el filtro para que se ordenen los vuelos y sus combinaciones, por precio, de mayor a menor.



```
// Cargamos los vuelos y los ordenamos de mayor a menor.  
viewModel.cargarVuelosFake(vuelosIda, vuelosVuelta)  
viewModel.ordenarVuelos(orden: "Mayor a menor")  
  
// Verificamos que las combinaciones están ordenadas correctamente.  
val combinaciones = viewModel.combinacionesValidas.value  
val precioTotalPrimera = combinaciones[0].first.precioBase + combinaciones[0].second.precioBase  
val precioTotalSegunda = combinaciones[1].first.precioBase + combinaciones[1].second.precioBase  
  
assertTrue(precioTotalPrimera >= precioTotalSegunda)  
  
// Mensaje de éxito si pasa el test.  
println("Éxito: Test 'ordenar combinaciones de mayor a menor' completado correctamente.")
```

Calidad del código

El código ha seguido principios de “**Clean Code**”, tales como:

- **Separación clara entre lógica de UI (pantallas) y lógica de negocio (ViewModel).**
- Uso de funciones bien nombradas, comentarios explicativos y código auto-documentado.
- Utilización de StateFlow y LiveData para mantener el estado reactivo de la app.

6. DESPLIEGUE DEL PRODUCTO

Una vez completado el desarrollo de **Spicy Airlines**, se ha preparado su despliegue como aplicación Android mediante la generación del archivo instalador **.apk**, siguiendo el proceso estándar de publicación en Android Studio. Este archivo permite instalar la aplicación en cualquier dispositivo Android compatible.

Generación del APK

Para generar el paquete instalador (**.apk**), se ha utilizado directamente Android Studio, que proporciona herramientas integradas para compilar y empaquetar la aplicación.

Se han seguido los siguientes pasos:

- Se ha generado el archivo APK con el **Build Bundle(s) / APK(s) > Build APK(s)**, obteniendo un paquete listo para instalar en cualquier dispositivo Android.



Este proceso facilita la distribución privada de la app sin necesidad de publicarla en Google Play.

Respecto al tema de entorno de pruebas y demostración; la aplicación se ha instalado y probado en emuladores Android para validar su correcto funcionamiento.

Para asegurar la disponibilidad durante la defensa del proyecto, se ha grabado un vídeo demostrativo en el que se muestra el uso completo de la aplicación: desde el registro de usuario, la búsqueda y reserva de vuelos, hasta la visualización de reservas y edición del perfil.

Posibilidad de publicación futura

Se podría preparar para una futura publicación en Google Play Store. Para ello, tendríamos qué:

1. Registrar una cuenta de desarrollador en Google Play.
2. Aplicar reglas de seguridad más estrictas en Firebase.
3. Firmar la app con una clave de producción .jks (keystore).
4. Subir el archivo .aab (Android App Bundle) a la consola de Google Play.

Con esto se quiere demostrar que se trata de una aplicación bastante **escalable**, con gran **potencial de crecimiento** y que puede convertirse en una aplicación mucho más robusta y adaptable a nuevas funcionalidades o mercados, como la inclusión de más destinos, integración de métodos de pago reales o la expansión a plataformas iOS en el futuro.

7. PRESUPUESTO, RIESGOS, VIABILIDAD

Para evaluar la viabilidad global del proyecto **Spicy Airlines**, se ha realizado una estimación presupuestaria y un análisis detallado de los principales riesgos que podrían afectar al desarrollo y lanzamiento de la aplicación, no solo desde un punto de vista técnico, sino también estratégico, económico y social.



Presupuesto estimado

El siguiente desglose presenta una estimación realista de los costes asociados al desarrollo de la aplicación:

Concepto	Coste estimado (€)
Desarrollo de la app Android (Kotlin + Compose)	8.750 € (250h x 35 €/h)
Backend y configuración de Firestore	1.400 € (40h x 35 €/h)
UI/UX y prototipado visual (Figma + iconos)	600 € (20h x 30 €/h)
Pruebas de calidad, rendimiento y validación	1.200 € (40h x 30 €/h)
Infraestructura Firebase (plan gratuito)	0 €
Gastos técnicos externos (dominio, despliegue)	100 €
Total estimado	12.050 €

Esta previsión refleja un enfoque eficiente de recursos gracias al uso de herramientas modernas como Firebase y Android Studio, que permiten reducir los costes de servidores o licencias.

Gestión de riesgos

Durante el desarrollo del producto, se han gestionado los **riesgos técnicos** mediante un tablero de trabajo en **Trello**, que ha permitido documentar incidencias reales como errores de validación,



reservas incompletas o sincronización con Firebase, así como definir medidas concretas para mitigar cada problema detectado.

Enlace al tablero de riesgos técnicos (Trello): <https://trello.com/b/Ym4ItJKM>

No obstante, para garantizar la sostenibilidad del proyecto en un entorno real, se ha realizado también un análisis de **riesgos empresariales, económicos y sociales** que podrían comprometer el éxito comercial o estratégico de la aplicación. A continuación, se resumen los principales riesgos no técnicos identificados:

Riesgos estratégicos, económicos y sociales

Riesgo	Tipo	Impacto	Medida de mitigación
Baja adopción del producto por parte del público objetivo	Económico	Alto	Realizar campañas de marketing específicas (segmentación).
Alta competencia con plataformas consolidadas	Estratégico	Alto	Enfocar el producto a un nicho concreto (Madrid – China)
Dificultad para monetizar o generar ingresos sostenibles	Económico	Alto	Estudiar modelos de negocio o acuerdos con aerolíneas/turismo.
Cambios legislativos en visados o restricciones.	Político	Medio	Adaptar rutas o mensajes si cambia la normativa.
Barreras culturales o idiomáticas entre usuarios europeos y chinos	Social	Medio	Traducir la app al chino e inglés, y adaptar mensajes culturales en futuras versiones.
Falta de diferenciación clara en el mercado	Estratégico	Medio	Potenciar elementos únicos: app ligera, sin publicidad, centrada en vuelos Madrid-China.



Este análisis permite prever posibles obstáculos y establecer estrategias tempranas para mitigarlos. El enfoque modular y escalable de la aplicación, junto con el control ágil de tareas mediante Trello, refuerzan la viabilidad técnica, económica y de negocio del proyecto **Spicy Airlines**.

8. DIFICULTADES ENCONTRADAS Y RESOLUCIÓN

Durante el desarrollo de la aplicación **Spicy Airlines**, se identificaron varias dificultades tanto técnicas como organizativas. A continuación, se detallan los principales problemas encontrados y las soluciones aplicadas:

1. Gestión de asientos y combinaciones de vuelos ida y vuelta:

Uno de los mayores retos fue implementar correctamente la lógica que permitiera buscar vuelos de ida o combinaciones de ida y vuelta, asegurando que existieran suficientes asientos en la clase seleccionada (Turista, Premium o Business).

Solución: Se aplicaron filtros adicionales en el ResultadosViewModel para verificar que tanto los vuelos individuales como las combinaciones cumplieran con los requisitos de fechas y disponibilidad de plazas. Además, se usó un SharedViewModel para mantener la coherencia de los datos entre pantallas.

2. Selección y validación de fechas con restricciones dinámicas:

El sistema de selección de fechas requería validar que la fecha de vuelta fuera al menos dos días posterior a la de ida, y limitar el rango máximo del viaje.

Solución: Se adaptó el componente DatePickerFirebase para aplicar estas restricciones de forma visual e interactiva, mejorando la usabilidad y evitando errores de búsqueda.

3. Guardado estructurado en Firebase con subcolecciones:

Al almacenar reservas en Firebase, fue complejo mantener la estructura correcta con múltiples vuelos y una subcolección de pasajeros.

Solución: Se definió una estructura de forma clara, donde cada reserva contiene los IDs de los vuelos (vuelos: List<String>), los datos del usuario, el precio total y una subcolección "pasajeros". Se utilizaron callbacks y manejo de errores para asegurar que todo se guardara correctamente.



4. Sincronización de datos entre colecciones de Firebase:

En pantallas como el perfil, era necesario mostrar las reservas junto con los datos completos de los vuelos asociados.

Solución: Se creó una clase ReservaConVuelo que combinaba la información de cada reserva con los datos de sus vuelos, gestionando todo desde el PerfilViewModel.

5. Validaciones de formularios:

Durante el registro, edición de perfil o introducción de pasajeros, surgió la necesidad de validar campos como teléfonos, códigos postales o pasaportes.

Solución: Se centralizaron todas las validaciones en funciones reutilizables dentro de la carpeta utils, asegurando consistencia y claridad en el código.

6. Problemas con el emulador por falta de memoria RAM:

Durante el desarrollo, el emulador de Android Studio presentaba problemas de rendimiento debido a las limitaciones de RAM del equipo.

Solución: Se optó por usar otros equipos ajenos, emulando ahí, hasta que hubiera otro equipo donde emular la app, se iba comprobando mediante debugs y preview todo el código.

7. Inserción masiva de vuelos en Firebase mediante JSON:

Crear manualmente todos los vuelos iba a tomar demasiado tiempo.

Solución: Se desarrolló un pequeño script en JavaScript, junto con un archivo .json que contenía toda la colección de vuelos generados previamente con sus respectivos campos. Para ello, se utilizó la herramienta de línea de comandos firebase-tools, que permitió subir los datos directamente a Firestore desde local ejecutando un comando específico. Este proceso automatizado permitió la inserción masiva de datos de forma rápida, evitando errores manuales y ahorrando tiempo en el desarrollo.

8. Uso de SharedViewModel para mantener el estado entre pantallas:

Una de las dificultades encontradas fue conservar los datos seleccionados por el usuario (como fechas, clase, vuelo o pasajeros) al navegar entre las distintas pantallas de la



aplicación. Si se usaban parámetros en la navegación, estos podían perderse o complicar el flujo.

Solución: Se implementó un SharedViewModel, accesible desde múltiples pantallas, que mantiene en memoria los datos esenciales durante todo el proceso de reserva. Esto facilitó el acceso y actualización de la información sin necesidad de pasarlos como argumentos entre pantallas, mejorando la organización del código y la experiencia de usuario.

9. Implementación del componente reutilizable DatePickerFirebase:

Seleccionar fechas de viaje en función del tipo de trayecto (ida o ida y vuelta) presentaba retos en cuanto a restricciones lógicas y visuales.

Solución: Se desarrolló un componente DatePickerFirebase personalizado que, según el modo seleccionado, permite elegir una o dos fechas y aplica validaciones como rango máximo de días o separación mínima entre ida y vuelta. Además, se reutiliza en otras pantallas como la introducción de fecha de nacimiento de los pasajeros, garantizando coherencia y facilidad de mantenimiento.

10. Uso de múltiples dependencias externas:

La inclusión de varias dependencias (Firebase, Material 3, Glide, etc.) provocó conflictos y errores de sincronización en el proyecto.

Solución: Se revisaron una por una las versiones y compatibilidades en el archivo build.gradle, manteniendo solo las necesarias y asegurando la correcta sincronización del proyecto con Gradle.

11. Exceso de ambición en la planificación inicial:

Inicialmente, se intentó abarcar demasiadas funcionalidades a la vez, lo que generó confusión y pérdida de tiempo al no tener una base sólida por la que empezar.

Solución: Se decidió reestructurar el proyecto y empezar con una planificación más clara y realista, construyendo primero las pantallas principales antes de integrar Firebase y los detalles avanzados.



9. CONCLUSIONES

El desarrollo de la aplicación **Spicy Airlines** ha supuesto un reto técnico, organizativo y creativo que ha permitido aplicar de forma práctica todos los conocimientos adquiridos durante el ciclo de Desarrollo de Aplicaciones Multiplataforma. A lo largo del proyecto **se han abordado tareas reales**, tales como la conexión con bases de datos en la nube mediante Firebase, la autenticación segura de usuarios, la creación de interfaces con Jetpack Compose y la aplicación de principios de usabilidad y diseño responsivo.

Gracias a una planificación clara y flexible, basada en la **metodología Scrum**, se ha logrado construir una aplicación funcional que permite buscar, comparar y reservar vuelos desde Madrid a las ciudades chinas de Chengdu y Chongqing, gestionando reservas y pasajeros de forma personalizada. El uso de **ViewModel** y **StateFlow** ha permitido mantener el estado entre pantallas y construir un sistema coherente y mantenable.

Durante el proyecto se han definido y aplicado **procedimientos de evaluación de actividades e incidencias**. Cada funcionalidad se desarrolló y validó en sprints semanales, estableciendo indicadores de calidad como el rendimiento (<15 segundos en búsquedas), la validación de formularios, la sincronización con Firebase y la correcta gestión del estado de la app. Las incidencias o riesgos detectados fueron registrados en Trello.

Asimismo, se ha contemplado la posibilidad de cambios en los recursos, reorganizando tareas, redefiniendo prioridades y ajustando funcionalidades (como separar vuelos de ida y vuelta o añadir filtros de clase). Estos cambios se han gestionado con un enfoque ágil, manteniendo la trazabilidad a través de GitHub y el tablero de planificación.

Entre los **logros** más destacados se encuentran:

- Implementación de filtros avanzados por clase y disponibilidad.
- Carga masiva de datos en Firebase mediante JSON y firebase-tools.
- Gestión visual clara y adaptable con Jetpack Compose y Material 3.
- Validación dinámica de fechas y datos con componentes personalizados como DatePickerFirebase.
- Mantenimiento del estado global con SharedViewModel.
Pruebas unitarias e integración con MockK, Firebase simulado y Compose UI Tests.



Este proyecto no solo ha servido como consolidación de conocimientos técnicos, sino también como una **experiencia realista de trabajo en un entorno profesional**, con control de versiones, gestión de incidencias, prototipado visual y documentación técnica completa.

PLAN A FUTURO

Con el desarrollo de **Spicy Airlines**, se plantea una evolución futura con nuevas funcionalidades y mejoras que van a convertir la app en un producto más completo, seguro y profesional, como:

- **Mejoras de seguridad:**

- Aplicar reglas estrictas en Firebase Firestore para restringir el acceso a usuarios autenticados.
- Implementar verificación por correo electrónico en el registro.
- Cifrado adicional en campos sensibles.

- **Nuevas funcionalidades:**

- Pantalla de pago real.
- Funcionalidad de “He olvidado mi contraseña” con recuperación por correo electrónico.
- Envío automático de un correo de confirmación con los datos de la reserva tras finalizarla.
- Añadir una pantalla “Contáctanos”, con un formulario para dudas o sugerencias.
- Sistema de notificaciones push para avisos de vuelo, cambios o promociones.
- Controlar el estado de una reserva (estado=false es una reserva cancelada o pasada)

- **Expansión de destinos y configuraciones:**

- Añadir más ciudades de China u otros destinos asiáticos.
- Incluir selección de moneda y conversión automática de precios.
- Soporte multilingüe (inglés, chino, español) para atraer a un público más amplio.



- **Publicación en Google Play:**

- Firmar la app con clave de producción.
- Preparar material gráfico promocional.
- Realizar pruebas de compatibilidad y rendimiento adicionales.
- Lanzamiento como MVP (Producto Mínimo Viable) y recogida de feedback.

Estas mejoras permitirían escalar el proyecto a un producto más robusto, preparado para su uso real y competitivo dentro del sector del turismo digital, especialmente en mercados nicho como los vuelos España–China.

10. REFERENCIAS BIBLIOGRÁFICAS

- Android Developers. (2024). Pruebas con ViewModel y LiveData.
<https://developer.android.com/topic/libraries/architecture/viewmodel#testing>
- Android Developers. (2024). Pruebas en Android Studio.
<https://developer.android.com/studio/test/test-in-android-studio?hl=es-419>
- Android Developers. (2024). Publicar apps en Google Play (checklist).
<https://developer.android.com/distribute/best-practices/launch/launch-checklist>
- Adobe XD Ideas. (2023). Principios de diseño UX/UI.
<https://xd.adobe.com/ideas/process/ui-design/ui-design-principles/>
- Firebase. (2024). Añadir datos a Firestore.
<https://firebase.google.com/docs/firestore/manage-data/add-data?hl=es-419>
- Firebase. (2024). Configuración inicial de Firebase para Web.
<https://firebase.google.com/docs/web/setup?hl=es>
- Firebase. (2024). Firebase Authentication para Android.
<https://firebase.google.com/docs/auth/android/start?hl=es-419>
- Firebase. (2024). Firebase CLI: Subir datos con firebase-tools.
<https://firebase.google.com/docs/cli?hl=es-419>
- Firebase. (2024). Firebase Firestore para Android.
<https://firebase.google.com/docs/firestore/quickstart?hl=es-419>



- Firebase. (2024). Política de privacidad.
<https://firebase.google.com/support/privacy?hl=es-419>
- Figma. (2024). Centro de ayuda oficial - Figma.
<https://help.figma.com/hc/en-us/sections/30880632542743>
- Git. (2024). Pro Git book. <https://git-scm.com/book/es/v2>
- GitHub Docs. (2024). Cómo usar GitHub. <https://docs.github.com/es/get-started>
- Google Developers. (2024). Cómo firmar APKs y AABs para producción.
<https://developer.android.com/studio/publish/app-signing>
- Google Material Design. (2024). Guía oficial de Material Design 3. <https://m3.material.io/>
- Jetpack Compose. (2024). Guía oficial de Jetpack Compose.
<https://developer.android.com/jetpack/compose/documentation>
- Jetpack Compose. (2024). Tutorial interactivo de Jetpack Compose.
<https://developer.android.com/jetpack/compose/tutorial>
- KeepCoding. (2023). Ventajas y desventajas de Kotlin.
<https://keepcoding.io/blog/ventajas-y-desventajas-de-kotlin/>
- MockK. (2024). Documentación oficial de MockK. <https://mockk.io/>
- Nielsen Norman Group. (2023). 10 heurísticas de usabilidad.
<https://www.nngroup.com/articles/ten-usability-heuristics/>
- Stack Overflow. (2013). How can I create tests in Android Studio.
<https://stackoverflow.com/questions/16586409/how-can-i-create-tests-in-android-studio>
- Stack Overflow. (2021). JUnit version dependency in androidTest can't use 4.13.
<https://stackoverflow.com/questions/70097484/junit-version-dependency-in-androidtest-cant-use-4-13>
- Stack Overflow. (2022). Unresolved reference: MockK.
<https://stackoverflow.com/questions/73799505/unresolved-reference-mockk>
- Trello. (2024). Guía oficial de Trello. <https://trello.com/es/guide/trello-101>
- UX en Español. (2023). 10 leyes principales de UX/UI.
<https://uxenespanol.com/articulo/10-leyes-principales-de-uxui>



- Wikipedia. (2024). Scrum (desarrollo de software).
[https://es.wikipedia.org/wiki/Scrum_\(desarrollo_de_software\)](https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software))
- YouTube. (2023). Tutorial Firebase Firestore desde 0.
<https://www.youtube.com/watch?v=BWNiMQkW3u4>
- Yago González. (2022). Tutorial básico de Trello. <https://yagogonzalez.com/tutorial-trello/>

11. ANEXOS

Se entregarán los archivos necesarios para la comprensión y complementación del proyecto, una APK, un vídeo demostrativo, el diagrama DER y el diagrama de clases.