Universidad Tecnológica ECOTEC

Entrega #1:

Diseño Arquitectónico y Principios de Software

Grupo:

Sistema de Reservas de Espacios Públicos

Nombre de los Integrantes:

Ashlee Coello: Fronted

Diego Rubio: Backend

Danny Freire: Pruebas y CI/CD

David Matute: UX/UI

Índice

Contenido

Índice 1
Introducción 2
Contenido Técnico 2
Historias de usuario 3
Diagrama de casos de uso 3
Arquitectura propuesta 4
Aplicación de principios SOLID (3 ejemplos) 5
Patrones de diseño usados 5
Consideraciones de seguridad y escalabilidad 5
Seguridad: 6
Escalabilidad: 6
Estructura inicial del proyecto en el repositorio 6
Diagrama de Componentes 9
Diagrama de secuencia 9
Diagrama de Clases 10
Diagrama de Actividades 11
Flujo de Trabajo 13
Evidencia de colaboración 15
Commits 15
Pull Request 17
Issues 18
Configurarcion de CI/CD: Github Actions 18
Despliegue continuo a entorno cloud: FireBase 19
Investigación de usuarios: encuestas, entrevistas, personas 23
Mapa de empatía o journey map 24
Prototipos (baja y alta fidelidad) 25
Diseño de interfaces 28
Paleta: 28
..... 28
Tipografía: 28
Navegación: 29
Evaluación de accesibilidad (WCAG): contraste, teclado, lectores de Pantalla 33
- Pruebas de usabilidad (con 2-3 usuarios reales) 35

- Tipos de pruebas aplicadas: unitarias, integración, funcionales 38
- Código de pruebas automatizadas (Jest, Cypress, PyTest, etc.) 40
Código de pruebas automatizadas (Jest, Cypress, PyTest, etc.).....41
- Pruebas de rendimiento (carga con 10 usuarios simulados) 44
- Pruebas de seguridad básicas (OWASP: inyección, autenticación) 46
Anexo 57
Evaluación de accesibilidad (WCAG): contraste, teclado, lectores de Pantalla 58
Tipos de pruebas aplicadas: unitarias, integración, funcionales 58
- Pruebas de seguridad básicas (OWASP: inyección, autenticación) 63
- Código de pruebas automatizadas (Jest, Cypress, PyTest, etc.) 66

Introducción

La primera entrega del proyecto Sistema de Reservas de Espacios Públicos tiene como propósito definir los cimientos del desarrollo, estableciendo la arquitectura, los principios de diseño y las funcionalidades iniciales que guiarán la implementación futura. En esta fase se busca:

- Definir historias de usuario que representen las necesidades principales del sistema.
- Elaborar el diagrama de casos de uso para visualizar interacciones clave.
- Proponer la arquitectura del sistema (capas, MVC, microservicios, etc.).
- Aplicar principios de diseño basados en SOLID.
- Identificar patrones de diseño apropiados.
- Considerar aspectos de seguridad y escalabilidad.
- Establecer la estructura inicial del repositorio y archivos base del proyecto.

Esta entrega proporciona la base conceptual y técnica que permitirá avanzar hacia un desarrollo estructurado, escalable y alineado a las buenas prácticas de ingeniería de software.

Contenido Técnico

Historias de usuario

1. **Como usuario**, quiero registrarme con correo y contraseña en la app para acceder a mi cuenta personal.

2. **Como usuario**, quiero explorar una lista de espacios disponibles (parques, canchas, auditorios) para elegir el que necesito.

3. **Como usuario**, quiero consultar un calendario de disponibilidad de cada espacio para hacer reservas de forma anticipada.

4. **Como usuario**, quiero ver un mapa de Guayaquil con los espacios geolocalizados para ubicar rápidamente dónde están.

5. **Como usuario,** quiero gestionar mi perfil y revisar mi historial de reservas para tener control sobre mis actividades.

6. **Como usuario**, quiero valorar y calificar los espacios públicos que he reservado para que los demás ciudadanos puedan conocer el estado del bien público.

Diagrama de casos de uso

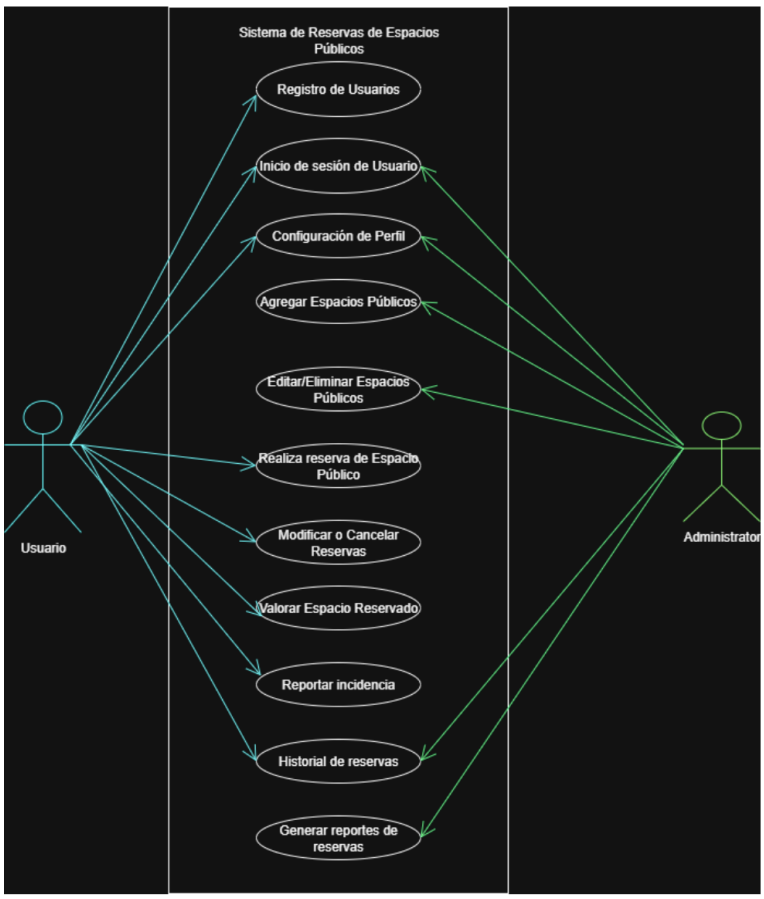


Diagrama Caso de Uso

Arquitectura propuesta

El sistema se desarrollará en Android Studio, utilizando una arquitectura en capas bajo el patrón MVC/MVVM, ya que permite separar la interfaz de usuario, la lógica de negocio y la gestión de datos de manera clara y organizada.

- **Capa de Presentación (View):** corresponde a las pantallas de la aplicación (Activities/Fragments en Android), donde el usuario podrá registrarse, iniciar sesión, reservar espacios y gestionar sus actividades.

- **Capa de Lógica de Negocio (Controller/ViewModel):** contiene la lógica del sistema, validaciones de datos y manejo de interacciones entre la vista y el modelo.
- **Capa de Datos (Model):** incluye las clases que representan las entidades principales (Usuarios, Espacios, Reservas, Valoraciones, etc.), así como la conexión con la base de datos local (SQLite) o servicios externos.

En una primera fase, el sistema funcionará de manera local con almacenamiento en SQLite; sin embargo, la arquitectura está diseñada para conectarse en el futuro a un servidor o API REST, lo que permitirá mayor escalabilidad, integración con notificaciones y generación de reportes avanzados.

Aplicación de principios SOLID (3 ejemplos)

- **Single Responsability Principle (SRP):** Cada clase tiene una única responsabilidad, por ejemplo, ReservationRepository solo se encarga de gestionar reservas.
- **Open/Close Principle (OCP):** El sistema será abierto a la extensión (por ejemplo: agregar login con redes sociales en el futuro), pero cerrado a modificaciones del código existente.
- **Liskov Substitution Principle (LSP):** Las clases hijas deben poder sustituir a sus padres sin problema.
- **Dependency Inversion Principle (DIP):** Los ViewModels dependerán de interfaces (IUserRepository) y no de implementaciones concretas, facilitando pruebas y mantenibilidad.

Patrones de diseño usados

- **Repository Pattern:** Abstracción de acceso a datos (base de datos interna) para usuarios y reservas.
- **Factory Method:** Para crear diferentes usuarios (Administrador, Usuario normal).
- **Singleton:** Para la instancia única de la base de datos o cliente de mapas.
- **Observer:** Implementado con LiveData o StateFlow en Android, para actualizar la UI en tiempo real cuando cambien los datos.

Consideraciones de seguridad y escalabilidad

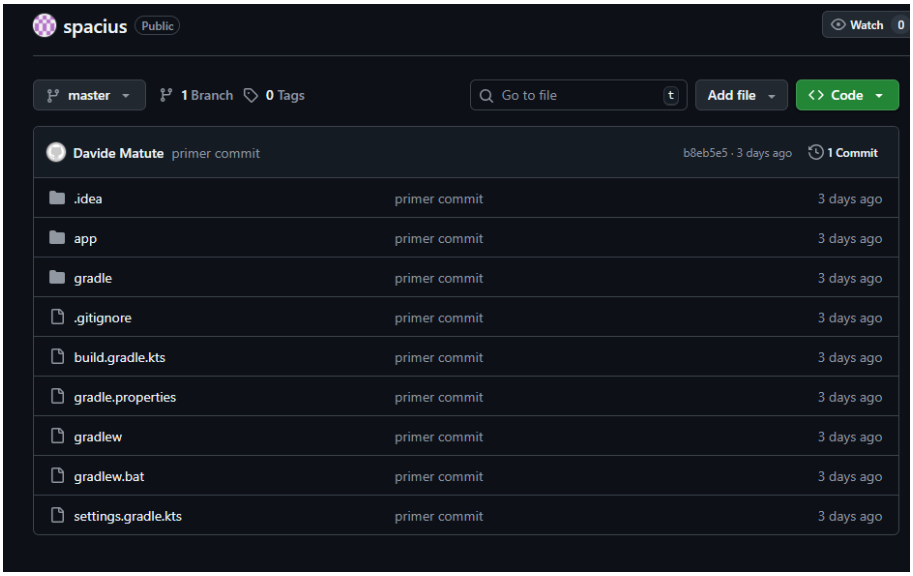
Seguridad:

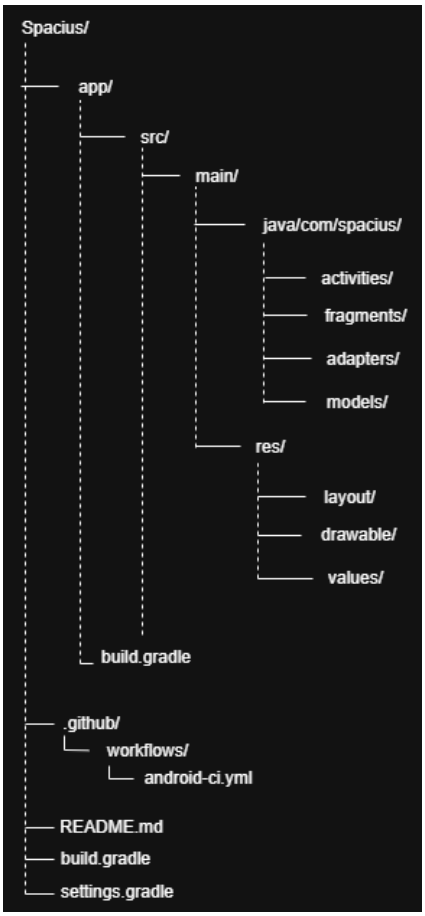
- Autenticación gestionada por la propia aplicación (correo y contraseña).
- Cifrado de contraseñas (hash + salt) antes de guardar en la base de datos.
- Reglas de acceso para diferenciar usuarios y administradores.
- Uso de HTTPS/TLS cuando la app se conecta a un backend o API externa.

Escalabilidad:

- Arquitectura limpia que permite agregar más funcionalidades sin romper la base.
- Repositorios modulares -> posible migración a Firestore o API REST en el futuro.
- Google Cloud Functions o backend para notificaciones masivas si se requiere.

Estructura inicial del proyecto en el repositorio





Link: https://github.com/davmatute-lang/spacius

Diagrama de Componentes

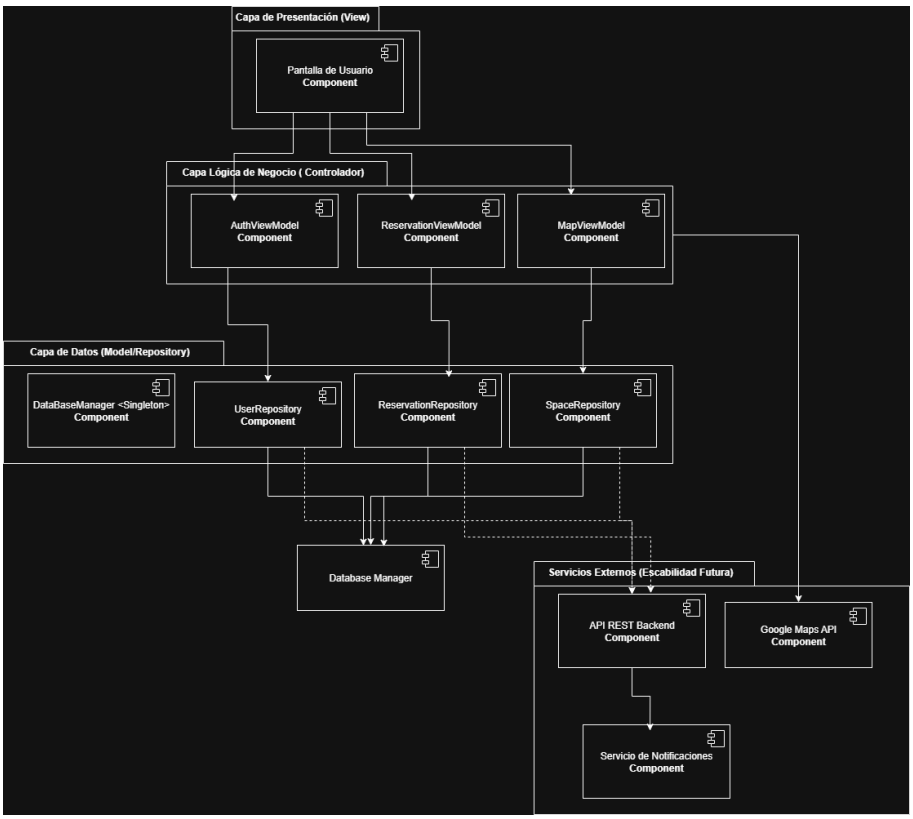
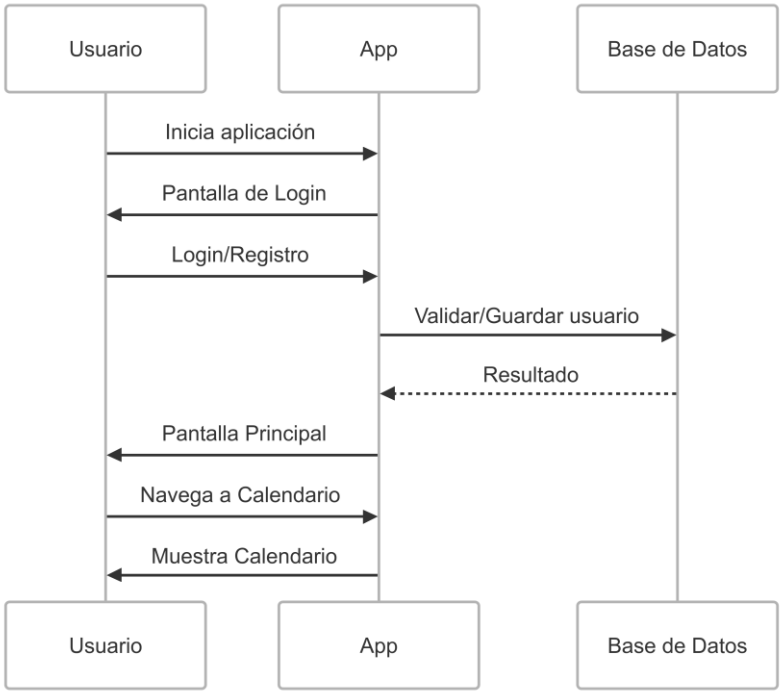


Diagrama de secuencia



Link del diagrama a detalle: https://www.mermaidchart.com/d/dd1b74e8-de79-4830-b1c5-42b70771c9ab

Diagrama de Clases

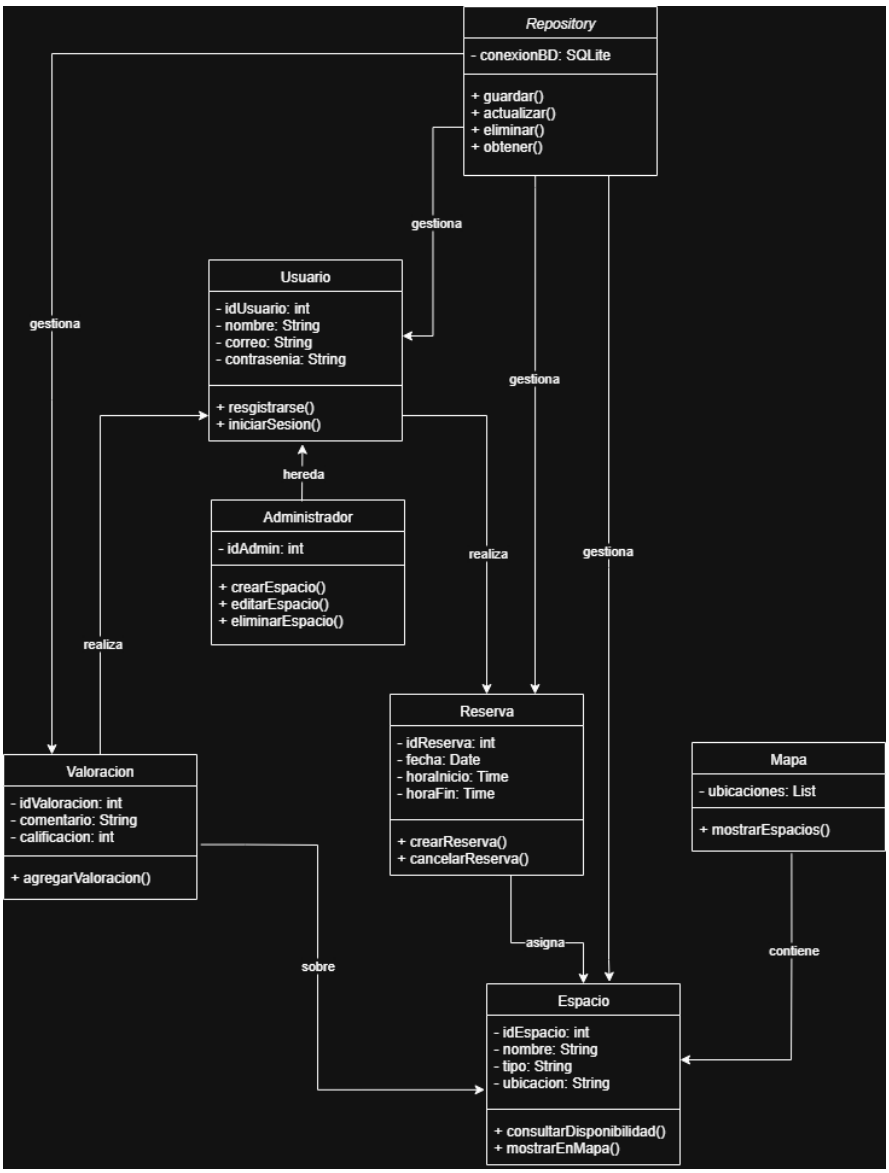
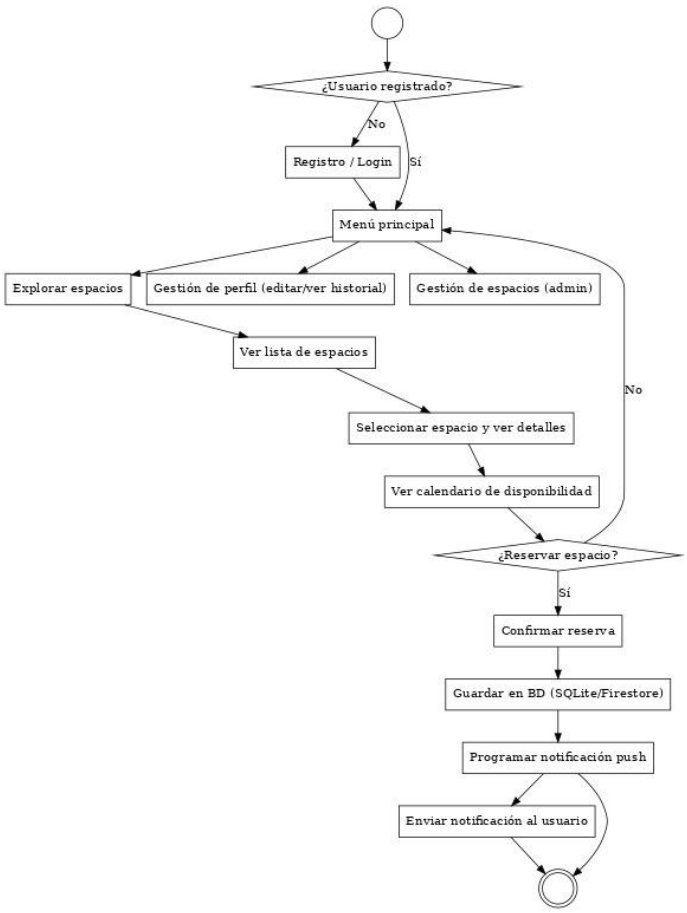


Diagrama de Actividades



Entrega #2:

Control de Versiones y CI/CD

Grupo:

Sistema de Reservas de Espacios Públicos

Nombre de los Integrantes:

Ashlee Coello: Fronted

Diego Rubio: Backend

Danny Freire: Pruebas y CI/CD

David Matute: UX/UI

Flujo de Trabajo

Paso 1 – Davide crea el repositorio y pantalla de inicio

1. Crear repo spacius-app en GitHub
2. Inicializar con README.md, .gitignore (Android), licencia MIT
3. Clonar local y crear rama develop

git clone https://github.com/davide/spacius-app.git cd
spacius-app
git checkout -b develop git push
-u origin develop

4. Proteger main → solo merge vía Pull Request
5. Crear rama feature/home-screen desde develop git
checkout develop
git checkout -b feature/home-screen

6. Implementar **HomeActivity/HomeFragment** y **lista de espacios + base de datos local (SQLite/Firestore)**
7. Commit + push → abrir Pull Request a develop → revisión → merge aprobado

Paso 2 – Danny implementa calendario y reservas

Historia de usuario: Consultar calendario y hacer reservas anticipadas

1. Crear rama feature/calendar-reservations desde develop git
checkout develop

git pull origin develop

git checkout -b feature/calendar-reservations

2. Implementar **CalendarFragment** y **ReservationRepository**, integrando con la base de datos de espacios creada por Davide
3. Commit + push → abrir PR a develop → revisión por Davide → merge aprobado

Paso 3 – Diego implementa mapa

Historia de usuario: Ver mapa con espacios geolocalizados

1. Crear rama feature/map desde develop git

checkout develop

git pull origin develop

git checkout -b feature/map

2. Implementar **MapFragment** e integración con **Google Maps API** usando la lista de espacios de Davide
3. Commit + push → abrir PR a develop → revisión por Davide → merge aprobado

Paso 4 – Ashlee implementa perfil, historial y notificaciones

Historias de usuario: Gestionar perfil, historial de reservas y recibir notificaciones push

1. Crear rama feature/profile-notifications desde develop git

checkout develop

git pull origin develop

git checkout -b feature/profile-notifications

2. Implementar **ProfileFragment**, **ReservationHistory**, integración con **Firebase Cloud Messaging**
3. Commit + push → abrir PR a develop → revisión → merge aprobado

Paso 5 – Davide prepara versión estable

1. Crear rama release release/v1.0 desde develop git

checkout develop

git pull origin develop

git checkout -b release/v1.0

2. Verificar pruebas + build APK
3. Merge a main y crear tag en GitHub git

checkout main

git merge release/v1.0 git

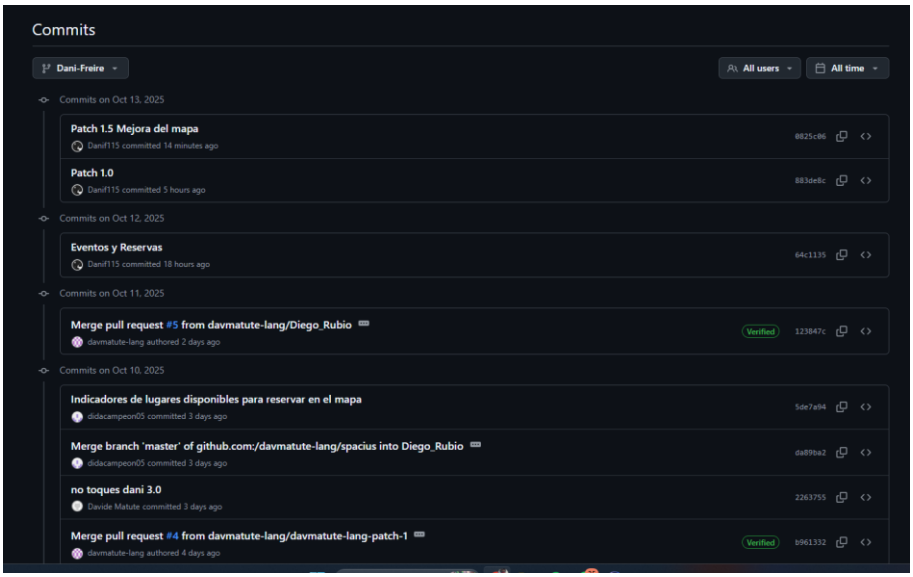
push origin main

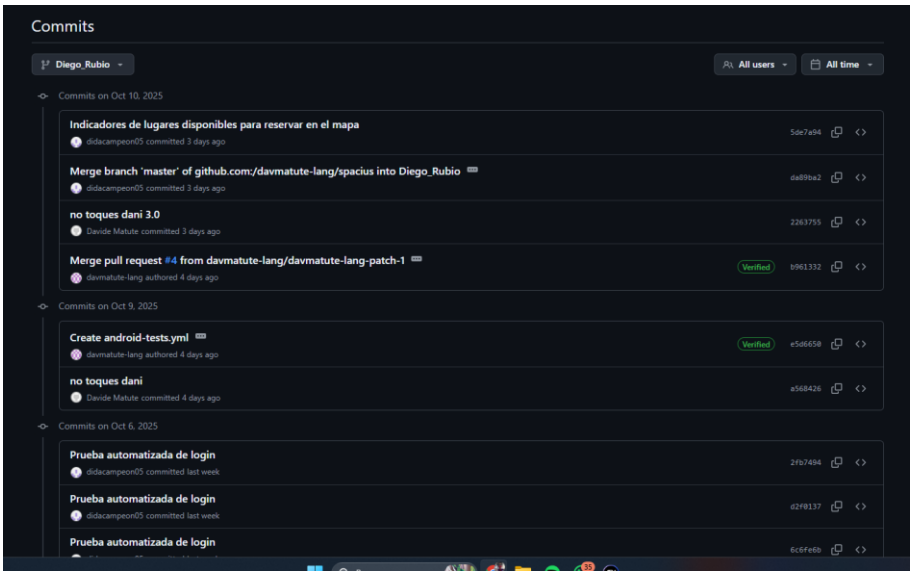
git tag -a v1.0 -m "Primera versión estable de Spacius" git

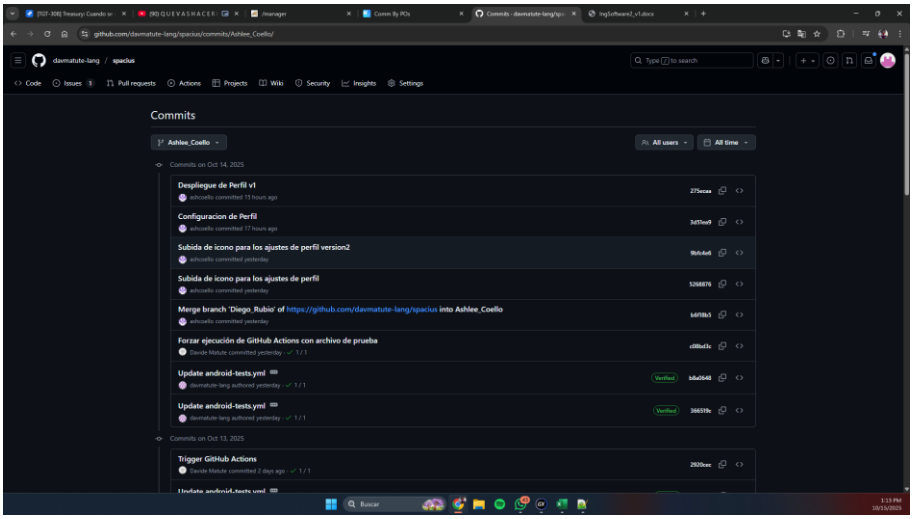
push origin v1.0

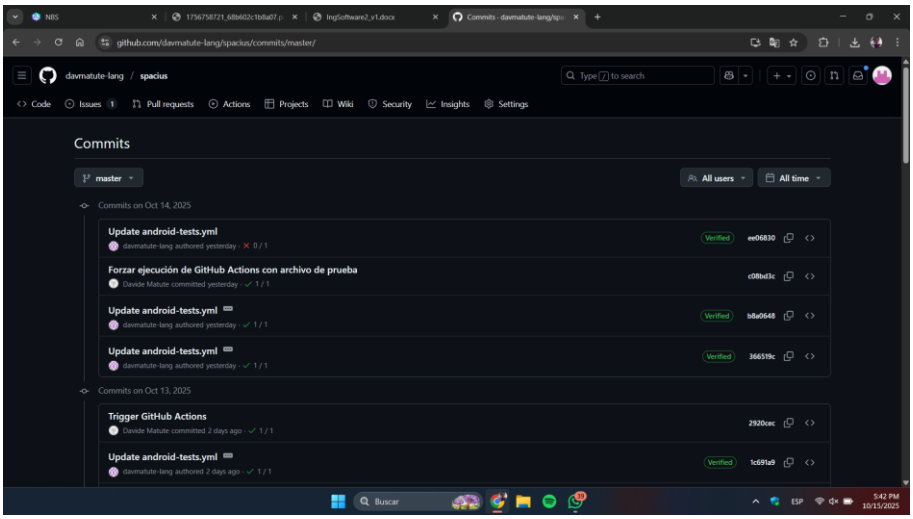
Evidencia de colaboración

Commits

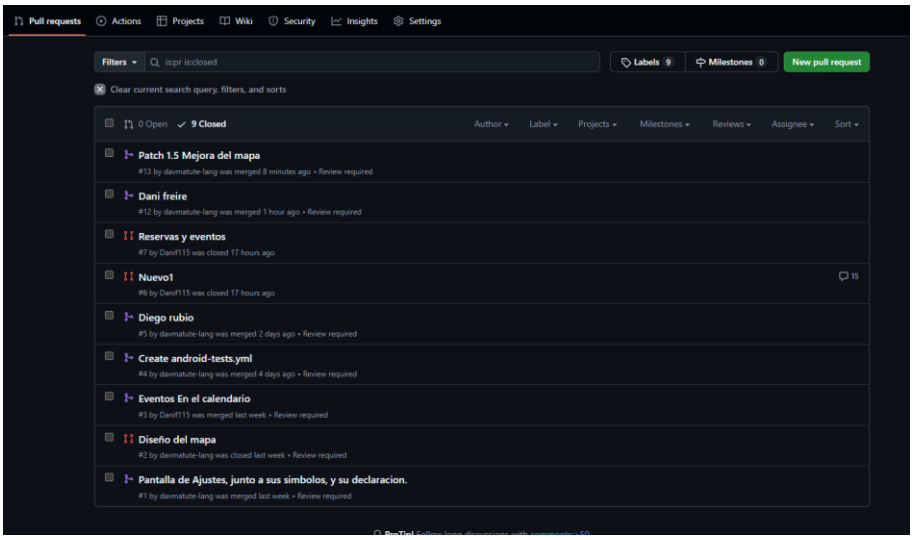



Commits
Diego_Rubio
All users
All time
Commits on Oct 10, 2025
Indicadores de lugares disponibles para reservar en el mapa
didacampeon05 committed 3 days ago
5de7a94
Merge branch 'master' of github.com:/davmatute-lang/spacius into Diego_Rubio
didacampeon05 committed 3 days ago
da89ba2
no toques dani 3.0
Davide Matute committed 3 days ago
2263755
Merge pull request #4 from davmatute-lang/davmatute-lang-patch-1
davmatute-lang authored 4 days ago
Verified
b961332
Commits on Oct 9, 2025
Create android-tests.yml
davmatute-lang authored 4 days ago
Verified
e5d6658
no toques dani
Davide Matute committed 4 days ago
a568426
Commits on Oct 6, 2025
Prueba automatizada de login
didacampeon05 committed last week
2fb7494
Prueba automatizada de login
didacampeon05 committed last week
d2f0137
Prueba automatizada de login
6c6fe6b

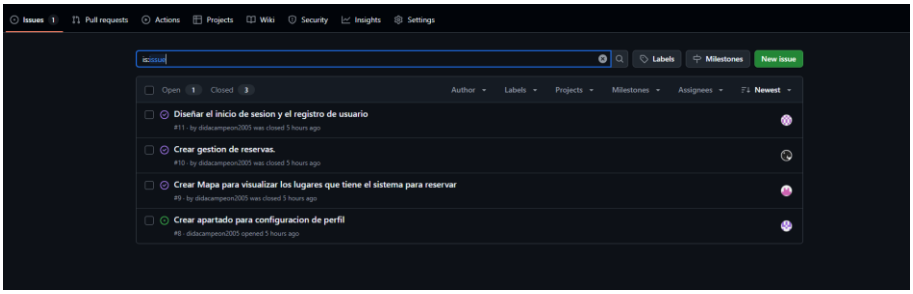

github.com/davmatute-lang/spacius/commits/Ashlee_Coello/
davmatute-lang / spacius
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Commits
Ashlee_Coello
All users
All time
Commits on Oct 14, 2025
Despliegue de Perfil v1
Configuracion de Perfil
Subida de icono para los ajustes de perfil version2
Subida de icono para los ajustes de perfil
Merge branch 'Diego_Rubio' of https://github.com/davmatute-lang/spacius into Ashlee_Coello
Forzar ejecución de GitHub Actions con archivo de prueba
Update android-tests.yml
Verified
Update android-tests.yml
Verified
Commits on Oct 13, 2025
Trigger GitHub Actions



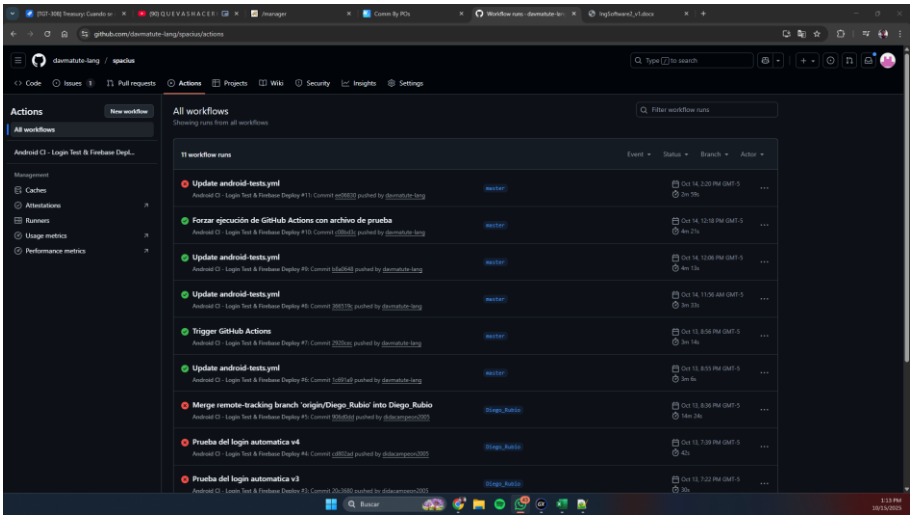
Pull Request

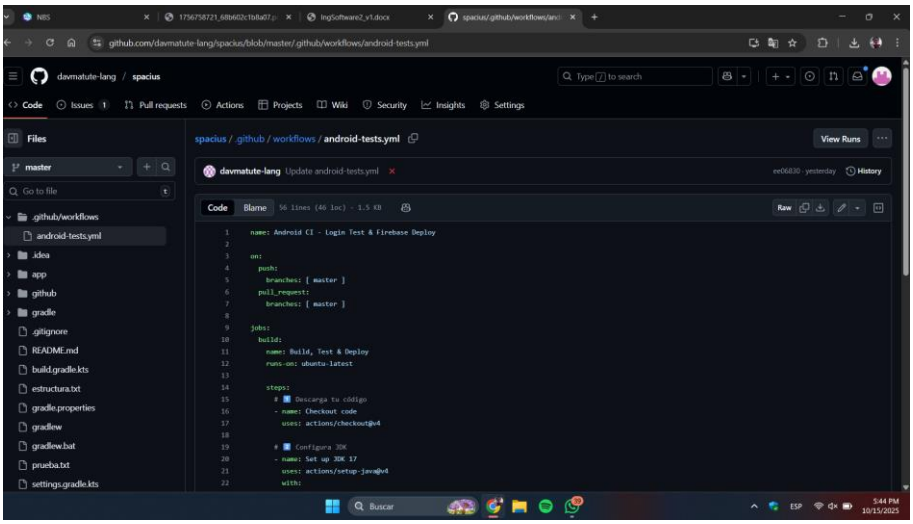


Issues

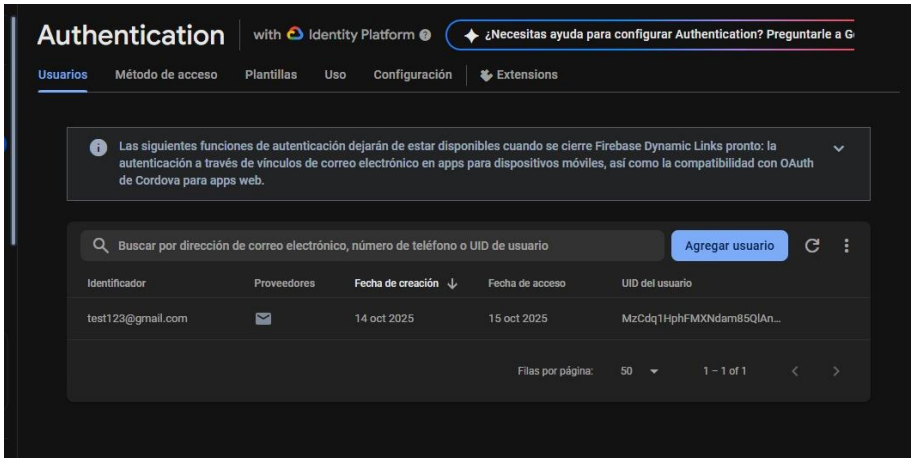


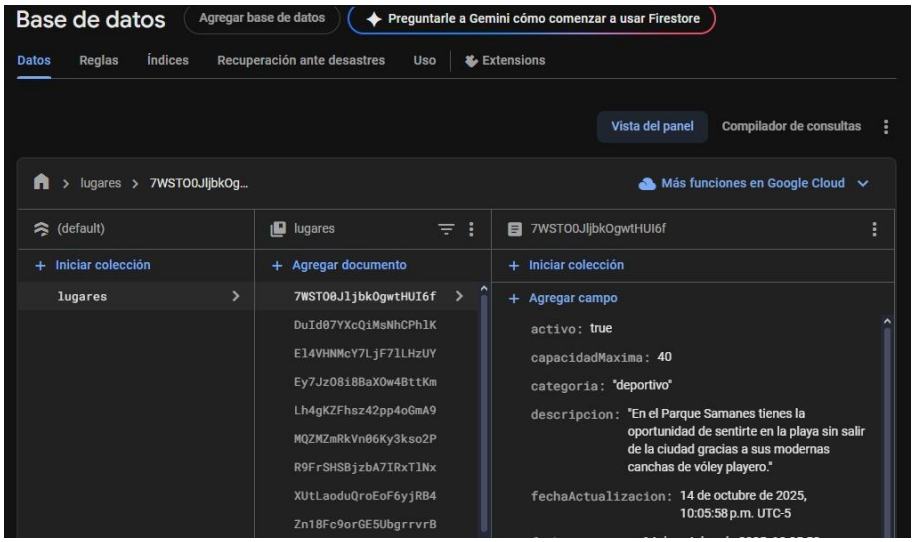
Configurarcion de CI/CD: Github Actions

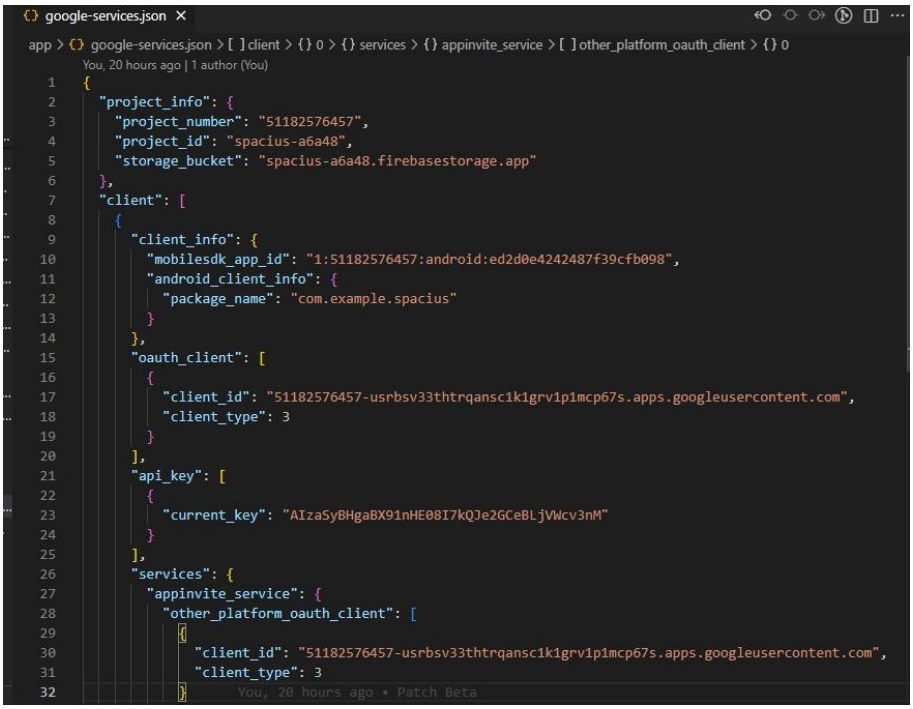




Despliegue continuo a entorno cloud: FireBase




Base de datos
Agregar base de datos
Preguntarle a Gemini cómo comenzar a usar Firestore
Datos
Reglas
Índices
Recuperación ante desastres
Uso
Extensions
Vista del panel
Compilador de consultas
lugares > 7WSTO0JljbkOg...
Más funciones en Google Cloud
(default)
lugares
7WSTO0JljbkOgwtHUI6f
+ Iniciar colección
+ Agregar documento
+ Iniciar colección
+ Agregar campo
lugares
7WSTO0JljbkOgwtHUI6f
DuId07YXcQiMsNhCPhlK
El4VHNMcY7LjF7lLHzUY
Ey7JzO8i8BaXOw4BttKm
Lh4gKZFhsz42pp4oGmA9
MQZMZmRkVn06Ky3kso2P
R9FrSHSBjzbA7IRxTlNx
XUtLaoduQroEoF6yjRB4
Zn18Fc9orGE5UbgrrvrB
activo: true
capacidadMaxima: 40
categoria: "deportivo"
descripcion: "En el Parque Samanes tienes la oportunidad de sentirte en la playa sin salir de la ciudad gracias a sus modernas canchas de vóley playero."
fechaActualizacion: 14 de octubre de 2025, 10:05:58 p.m. UTC-5


google-services.json
app > google-services.json > [ ] client > {} 0 > {} services > {} appinvite_service > [ ] other_platform_oauth_client > {} 0
You, 20 hours ago | 1 author (You)
{
"project_info": {
"project_number": "51182576457",
"project_id": "spacius-a6a48",
"storage_bucket": "spacius-a6a48.firebasestorage.app"
},
"client": [
{
"client_info": {
"mobilesdk_app_id": "1:51182576457:android:ed2d0e4242487f39cfb098",
"android_client_info": {
"package_name": "com.example.spacius"
}
},
"oauth_client": [
{
"client_id": "51182576457-usrbsv33thtrqansc1k1grv1p1mcp67s.apps.googleusercontent.com",
"client_type": 3
}
],
"api_key": [
{
"current_key": "AIzaSyBHgaBX91nHE08I7kQJe2GCeBLjVWcv3nM"
}
],
"services": {
"appinvite_service": {
"other_platform_oauth_client": [
{
"client_id": "51182576457-usrbsv33thtrqansc1k1grv1p1mcp67s.apps.googleusercontent.com",
"client_type": 3
}
You, 20 hours ago • Patch Beta

Link del repositorio: https://github.com/davmatute-lang/spacius

Entrega #3:

Diseño de Experiencia e Interfaz (UX/UI)

Grupo:

Sistema de Reservas de Espacios Públicos

Nombre de los Integrantes:

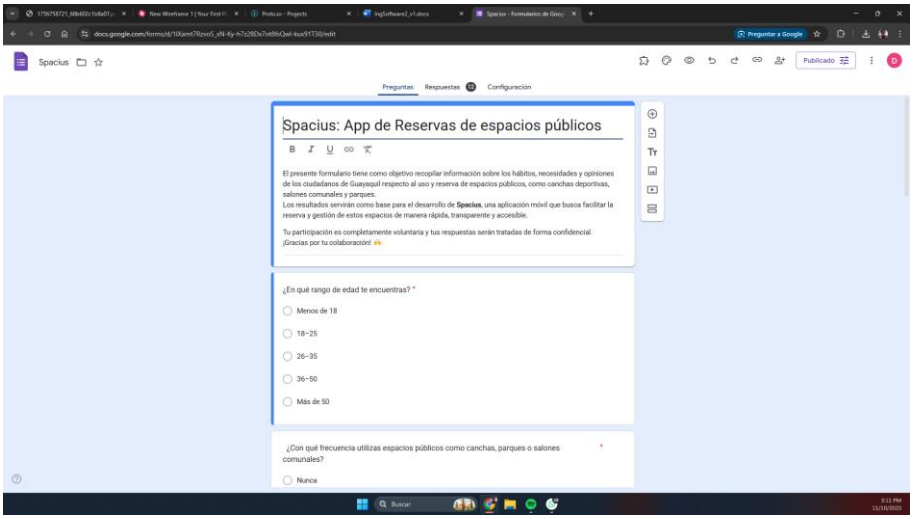
Ashlee Coello: Fronted

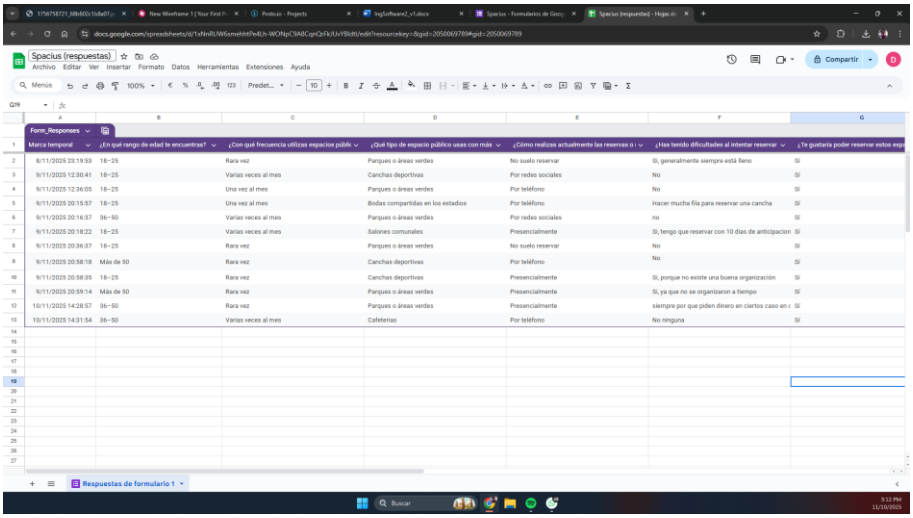
Diego Rubio: Backend

Danny Freire: Pruebas y CI/CD

David Matute: UX/UI

Investigación de usuarios: encuestas, entrevistas, personas





Link de la Encuesta: https://forms.gle/H89QVNwPe7PFwAnSA

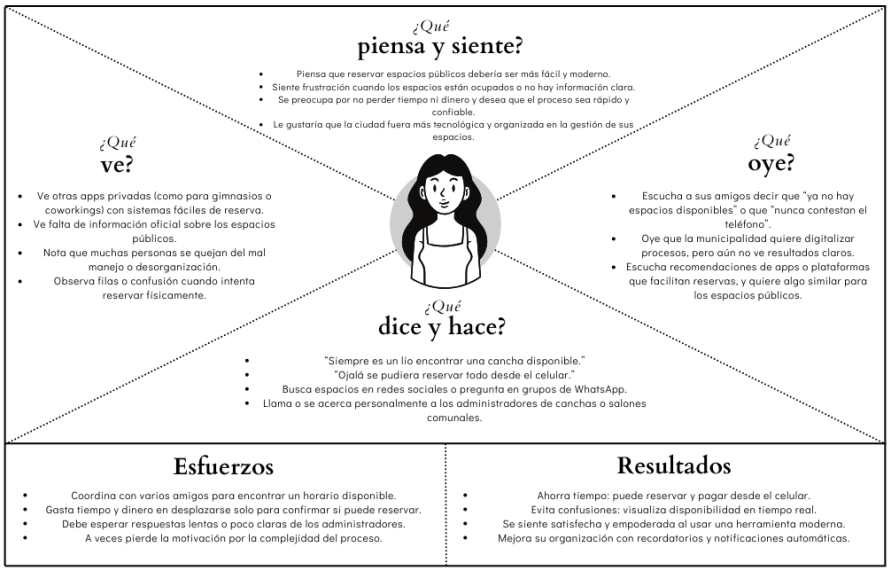
Mapa de empatía o journey map

Mapa de empatía

Identificando el comportamiento del usuario

Diseñador por: Coello, Rubio, Matute, Freire

Fecha: 08/11/2025



Personaje

Usuario representante de Spacius

Nombre:
María López

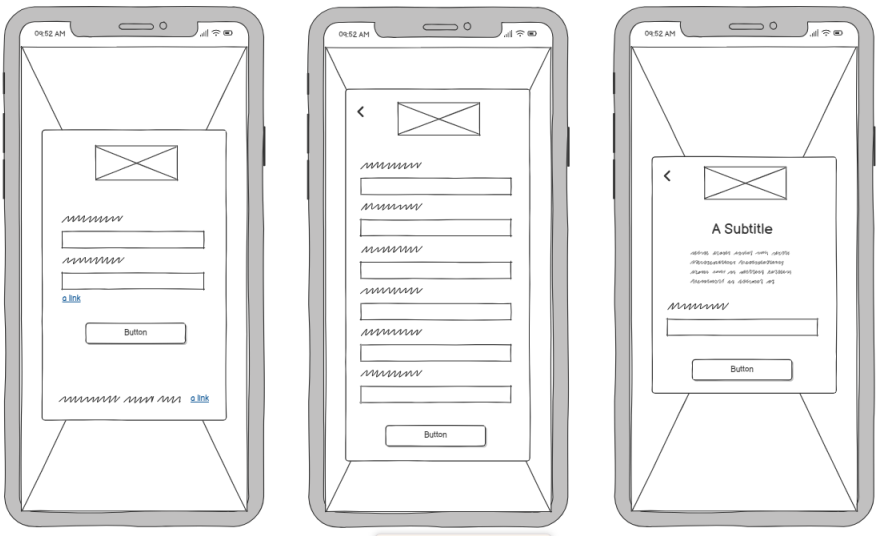
Occupación:
Estudiante universitaria y jugadora aficionada de volley

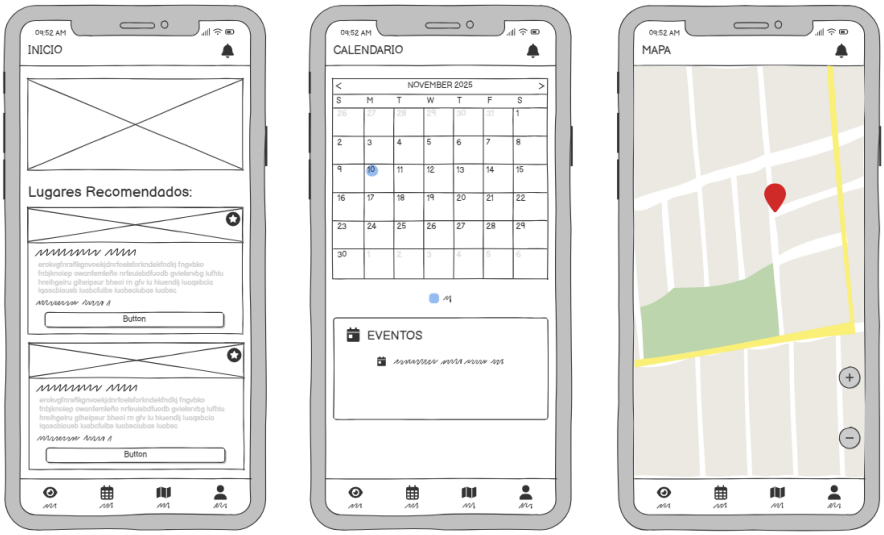
Edad:
26 años

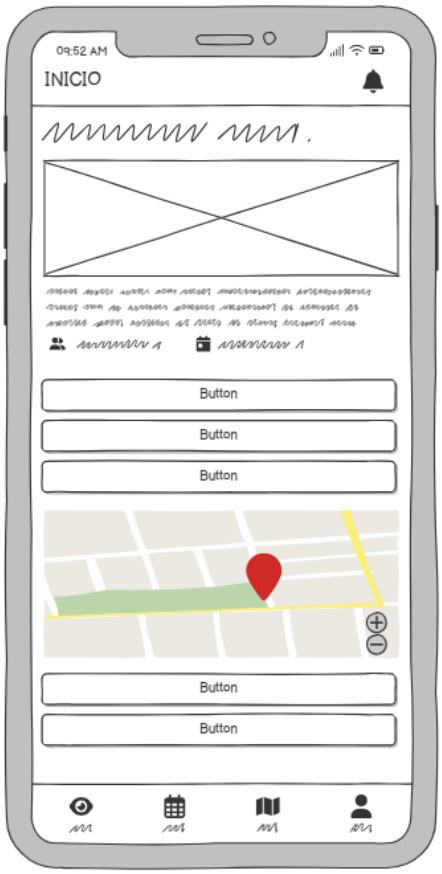
Contexto:
Le gusta jugar vóley con sus amigos en canchas públicas, pero suele frustrarse por la falta de organización y disponibilidad de los espacios. Ella es una persona práctica y digital, que usa su celular para casi todo.

Objetivo:
Reservar fácilmente una cancha o espacio público sin perder tiempo ni depender de trámites presenciales.

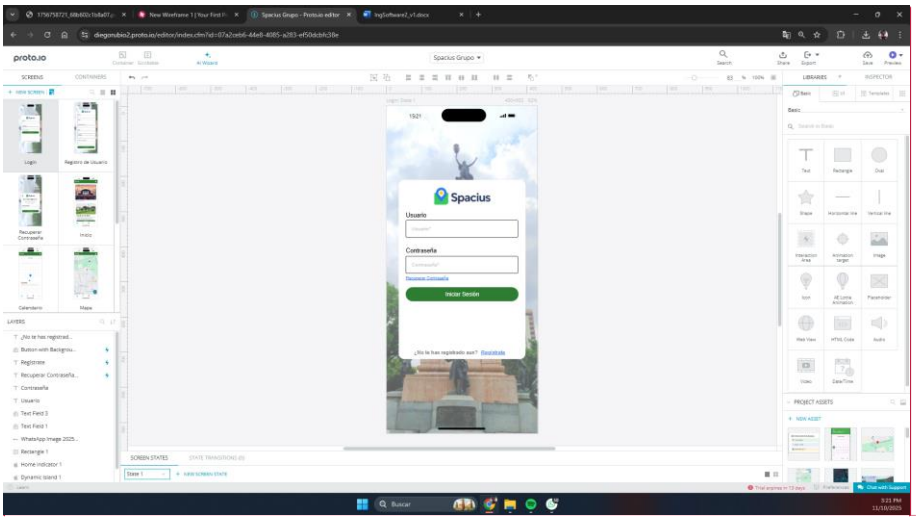
Prototipos (baja y alta fidelidad)




09:52 AM
INICIO
Lugares Recomendados:
Button
Button
09:52 AM
CALENDARIO
NOVEMBER 2025
S M T W T F S
EVENTOS
09:52 AM
MAPA


09:52 AM
INICIO
Button
Button
Button
Button
Button


AJUSTES
Button
Button
Button

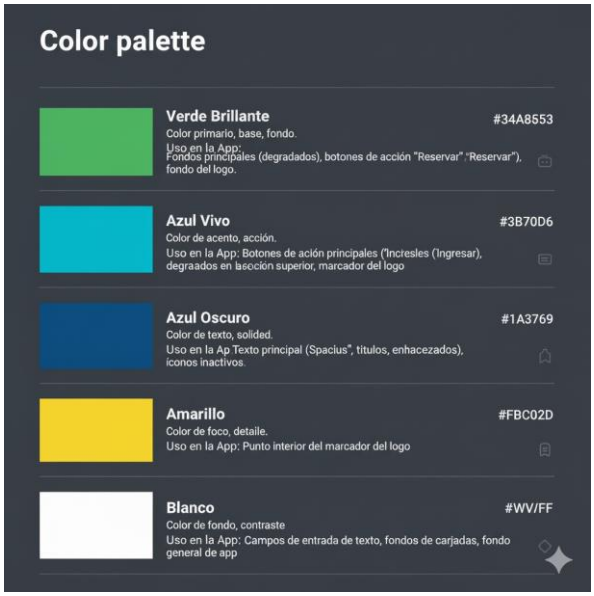


Commented [Ui1]: https://pr.to/N3A8SQ/

Link de Proto.io: https://pr.to/N3A8SQ/

Diseño de interfaces

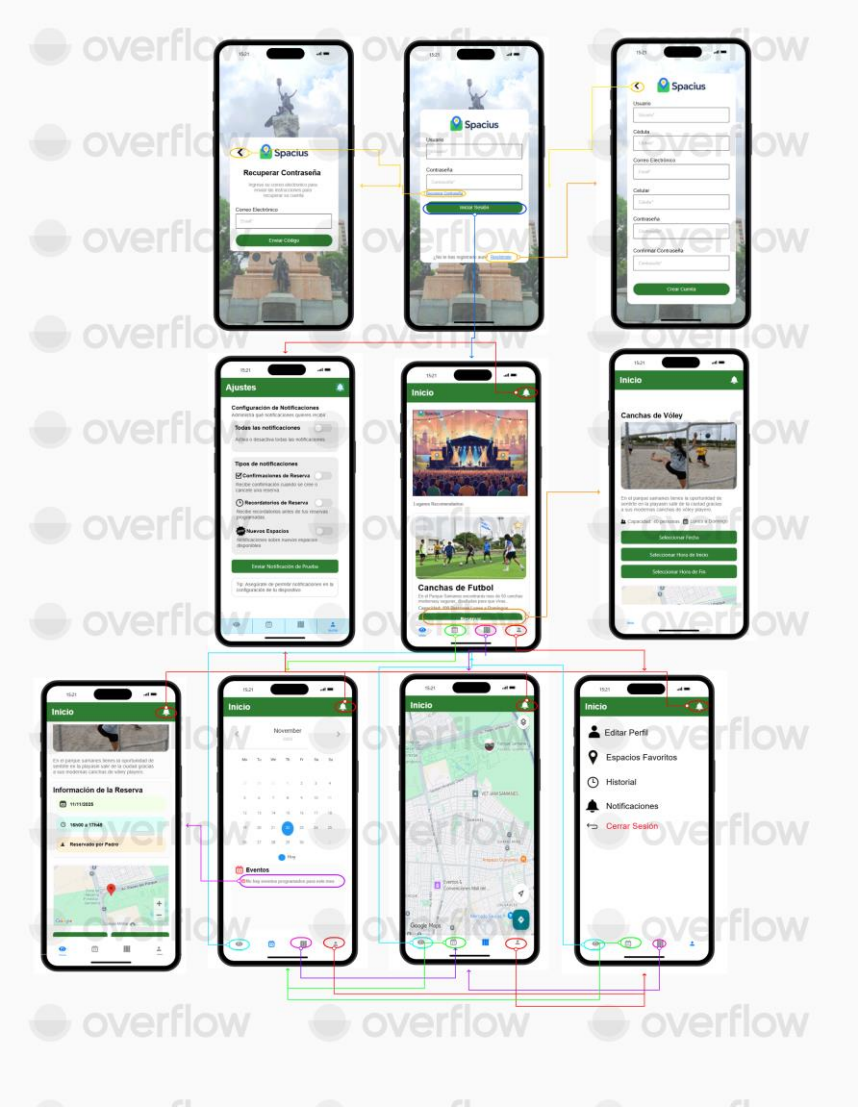
Paleta:



Tipografía:

Arial

Navegación:



Link de la imagen:
https://drive.google.com/file/d/1bciXSFD3drJzOryHzf4TUhyTcSu8gtmf/view?usp=sharing

Entrega #4:

Accesibilidad, Usabilidad y Pruebas

Grupo:

Sistema de Reservas de Espacios Públicos

Nombre de los Integrantes:

Ashlee Coello: Fronted

Diego Rubio: Backend

Danny Freire: Pruebas y CI/CD

David Matute: UX/UI

Plan de Pruebas				
Tipo de Prueba	**Objetivo**	**Alcance / Actividades Realizadas**	**Hallazgos Principales**	**Recomendaciones / Acción Correctiva**
Pruebas de Accesibilidad (WCAG)	Validar que la aplicación sea usable por personas con limitaciones visuales o motrices.	- Evaluación de contraste. - Navegación por teclado. - Compatibilidad con lectores de pantalla.	- Buen nivel de contraste. - Navegación con Tab fluida. - Los lectores de pantalla interpretan correctamente los elementos.	Mantener controles WCAG en futuras versiones y reevaluar tras cambios de diseño.
Pruebas de Usabilidad (2–3 usuarios reales)	Comprobar qué tan intuitivo es el uso de la app Spacius para personas sin experiencia previa.	- Inicio de sesión. - Exploración de espacios. - Reserva y cancelación. - Entrevista post-prueba.	- Confusión al seleccionar horarios. - Redirección poco clara tras reservar. - Cancelación sin confirmación.	Unificar selector de horario, mejorar navegación post-reserva y añadir confirmación de cancelación.
Pruebas Unitarias	Validar métodos de forma aislada y detectar errores de lógica.	- Validación de horarios. - Manejo de fechas. - Formatos válidos e inválidos.	Se detectaron escenarios inválidos correctamente (horas invertidas, fechas pasadas, etc.).	Mantener esquema Arrange–Act–Assert y ampliar casos borde.
Pruebas de Integración	Confirmar que los módulos relacionados funcionen bien juntos.	- Interacción entre métodos de fecha y hora. - Conversión de cadenas. -	La lógica fluye correctamente entre métodos. No se detectaron	Continuar validando módulos nuevos que interactúen con el flujo de reservas.

		Validación completa.	fallos en integración.	
Pruebas Funcionales	Validar el comportamiento real desde la perspectiva del usuario.	- Selección de fecha. - Elección de horario. - Confirmación de disponibilidad.	El sistema bloquea horarios inválidos y fechas pasadas según lo esperado.	Mantener pruebas funcionales al agregar funciones nuevas.
Pruebas Automatizadas (JUnit - Espresso)	Automatizar pruebas de lógica, UI y seguridad.	- 44 pruebas ejecutadas (todas exitosas). - Unitarias, instrumentadas y de seguridad.	- Correcta ejecución en JUnit. - UI estable bajo pruebas de Espresso. - Seguridad validada con 100% de éxito.	Integrar ejecución automática en pipeline.
Pruebas de Rendimiento (10 usuarios simulados)	Validar estabilidad bajo carga moderada.	- Inicio de sesión simultáneo. - Reservas concurrentes. - Consultas repetidas. - Picos de carga.	- Buen tiempo de respuesta. - Sin colisiones en reservas. - Memoria estable y sin errores.	Considerar pruebas con mayor carga si se escala el uso.
Pruebas de Seguridad (OWASP)	Detectar vulnerabilidades comunes.	- Autenticación. - Autorización. - Sanitización de entradas. - Prevención de inyección.	- Rechazo de contraseñas débiles. - Acceso restringido correctamente. - Entradas	Mantener validaciones y ampliar controles cuando se integren servicios externos.

			sanitizadas frente a SQLi y XSS.	

- Pruebas de seguridad básicas (OWASP: inyección, autenticación)

Aspecto de Accesibilidad	Qué se Evaluó en la App	Cómo se Hizo la Revisión	Hallazgos y Observaciones	Qué se Espera que Cumpla (WCAG)
Contraste de colores	Se revisaron pantallas como Login, listas de lugares, botones principales, textos dentro de tarjetas y mensajes. También fondos con gradientes.	• Mirando la appen modo claro y oscuro. • Comparando textos y fondos para ver si se distinguen sin esfuerzo. • Usando una herramienta básica de contraste para verificar si colores claros se mezclan con el fondo.	• En la mayor parte de la appel texto resalta bien. • En la pantalla de inicio de sesión, algunas zonas del fondo degradado hacen que el texto blanco pueda perderse un poco si la imagen se vuelve muy clara. • Los colores de botones y tarjetas sí tienen buena legibilidad.	Que el texto pueda leerse sin forzar la vista, incluso para personas con baja visión. El contraste debe mantenerse estable y no desaparecer cuando el fondo cambia.
Navegación solo con teclado	Se probó si una persona puede usar la app sin tocar la	• Usando únicamente Tab, Shift+Tab, Enter y la tecla Espacio.	• La app deja moverse por casi todo sin problema.	Que la app pueda usarse sin necesidad de tocar la pantalla y que cualquier persona

	pantalla: moverse entre botones, escribir, retroceder, abrir y cerrar elementos.	• Revisando si el “foco” pasa de forma clara de un elemento a otro. • Avanzando por listas y saliendo de formularios sin quedar atrapado.	• El orden del foco es lógico: primero campos de texto, luego botones. • Las tarjetas de eventos y lugares permiten seleccionarse y entrar a los detalles. • No se detectaron puntos donde el usuario se quede atrapado o sin salida.	que dependa del teclado no se quede bloqueada en ningún punto.
Lectores de pantalla (TalkBack)	Se revisó cómo la app“habla” cuando alguien con discapacidad visual la usa: botones, imágenes, íconos, campos de texto, listas y mensajes.	• Activando TalkBack y recorriendo la appelemento por elemento. • Escuchando qué dice la app y si coincide con lo que realmente está en pantalla. • Revisando que los emojis o íconos tengan una descripción adecuada.	• La mayoría de botones y campos sí tienen un nombre claro. • Las imágenes principales tienen descripción, lo cual ayuda mucho. • Se corrigió un problema donde los emojis no tenían descripción y	Que cada elemento importante tenga una descripción clara y entendible para que un usuario pueda orientarse y saber qué está haciendo en cada momento.

			TalkBack no podía interpretarlos bien. • Los mensajes de los formularios sí se leen correctamente.	

Evaluación de accesibilidad (WCAG): contraste, teclado, lectores de Pantalla

1. Introducción

La accesibilidad es un aspecto clave en el desarrollo de aplicaciones, ya que permite que personas con diferentes capacidades puedan utilizarlas sin dificultades. Para esta evaluación se tomaron como referencia las pautas **WCAG**, que son el estándar más utilizado para medir el nivel de accesibilidad de una interfaz.

El análisis se centró en tres puntos esenciales: el contraste de colores, el uso de la aplicación únicamente con el teclado y la compatibilidad con lectores de pantalla. Estos elementos influyen directamente en la experiencia de usuarios con limitaciones visuales o motrices.

2. Evaluación del contraste

El contraste entre el texto y el fondo es fundamental para asegurar la legibilidad. Personas con baja visión o distintos tipos de daltonismo dependen de un contraste adecuado para poder identificar elementos de la interfaz.

Durante la revisión se verificó que:

- Los textos principales mantienen un nivel de contraste suficiente para ser leídos con comodidad.
- Los botones con texto blanco sobre fondo de color presentan buena visibilidad.
- Los colores utilizados sobre fondo blanco cumplen con los valores recomendados por los estándares WCAG.
- El texto grande también cumple con el nivel mínimo de contraste exigido.

En general, la aplicación muestra una buena selección de colores y no presenta problemas que dificulten la lectura.

3. Navegación por teclado

Otra parte importante de la accesibilidad es permitir que todas las funciones se puedan utilizar mediante el teclado, sin necesidad de un mouse. Esto beneficia a personas con dificultades motrices o que dependen de dispositivos alternativos.

Los resultados fueron los siguientes:

- Los elementos interactivos responden correctamente al uso de teclas como **Tab**, **Enter** y **Espacio**.
- El foco avanza de manera lógica, sin saltos inesperados o cambios desordenados.
- No se encontraron zonas de la aplicación en las que el usuario quede atrapado sin poder continuar.
- Los botones y controles tienen un tamaño adecuado para ser seleccionados con facilidad.

La navegación resulta fluida y permite recorrer toda la interfaz sin inconvenientes.

4. Compatibilidad con lectores de pantalla

Para los usuarios con discapacidad visual, los lectores de pantalla son una herramienta esencial. La aplicación debe proporcionar la información necesaria para que estos programas puedan interpretar correctamente lo que aparece en pantalla.

Los hallazgos principales fueron:

- Los botones y elementos interactivos tienen descripciones claras que permiten identificar su función.
- Las imágenes decorativas no incluyen descripciones innecesarias, lo cual evita confusiones.
- Los mensajes de error se anuncian correctamente y son entendibles para el lector de pantalla.
- Se mantiene coherencia en los nombres y acciones de los botones, facilitando la navegación auditiva.

La aplicación se muestra bien adaptada para usuarios que dependen de tecnologías asistivas.

5. Conclusión

La evaluación realizada demuestra que la aplicación cumple con los aspectos esenciales de accesibilidad relacionados con el contraste visual, el uso mediante teclado y el soporte para lectores de pantalla.

Estos resultados indican que la interfaz ha sido diseñada teniendo en cuenta a distintos tipos de usuarios, lo cual mejora significativamente la experiencia general.

Se recomienda continuar aplicando estos principios en futuras actualizaciones y realizar nuevas pruebas cuando se incorporen cambios importantes al diseño o la funcionalidad.

- Pruebas de usabilidad (con 2-3 usuarios reales)

1. Introducción

En las presentes pruebas recoge el plan, ejecución y hallazgos derivados de las pruebas de usabilidad realizadas para la aplicación **Spacius**, una plataforma diseñada para explorar, reservar y gestionar espacios disponibles dentro de una institución.

El objetivo principal de la prueba fue identificar qué tan intuitivo resulta el flujo de inicio de sesión, búsqueda de lugares, reserva y cancelación para usuarias nuevas.

Las pruebas se llevaron a cabo con dos participantes:

- **Catterina (22 años)** – Estudiante de Sistemas Inteligentes.
- **Giannella Cabrera (22 años)** – Estudiante de Criminalística.

Cada usuaria realizó tareas mientras pensaba en voz alta, permitiendo observar su comprensión del sistema, expectativas y dificultades.

2. Procedimiento de la Prueba

Antes de iniciar, se les aclaró que **no se evaluaba su habilidad**, sino el desempeño de la aplicación. Se les pidió verbalizar sus pensamientos para comprender mejor qué observaban y qué esperaban que ocurra.

Se siguieron cinco tareas principales:

1. Iniciar sesión.
2. Explorar lugares disponibles.
3. Realizar una reserva.

4. Confirmar y verificar la reserva.
5. Cancelar la reserva.

Al finalizar, se aplicó una breve entrevista post-prueba para recoger opiniones generales, puntos positivos y oportunidades de mejora.

3. Resultados Observados

3.1 Aspectos Positivos

Ambas participantes lograron completar todas las tareas. Entre los puntos favorables destacan:

- **Inicio de sesión claro y rápido:** No generó problemas.
- **Diseño visual atractivo:** Las imágenes de los lugares generaron interés y facilitaron la selección.
- **Concepto útil:** Ambas consideraron que una app así sería valiosa para actividades académicas en Guayaquil.

4. Problemas Detectados y Recomendaciones

A continuación, se sintetizan los puntos críticos identificados durante la prueba, ordenados según la tarea correspondiente.

Problema 1: Confusión al seleccionar la hora (Tarea 3: Hacer una reserva)

Observación:

Ambas usuarias mostraron confusión cuando debían elegir la hora. Intentaron usar los botones "Hora Inicio" y "Hora Fin", aunque realmente el sistema asigna ambos valores desde un único bloque horario.

Comentarios relevantes:

- *"Si ya elijo un bloque 09:00 - 11:00, ¿para qué sirve el botón de hora fin?"* – Catterina
- *"Pensé que debía elegir la hora final aparte y no pasaba nada."* – Giannella

Análisis Técnico:

La UI ofrece dos botones (hora inicio y hora fin), pero el flujo de ReservaFragment solo usa el bloque completo desde el diálogo emergente.

Recomendación:

- Reemplazar los dos botones por un solo botón: **"Seleccionar horario"**.
- Tras elegir un bloque (por ejemplo, "09:00 - 11:00"), el texto del botón debería actualizarse con ese horario.

Esto reduce la ambigüedad y alinea la interfaz con la lógica real del sistema.

Problema 2: Desorientación tras completar la reserva (Tarea 4)

Observación:

Después de reservar, la aplicación redirige automáticamente al calendario, lo que dejó a las usuarias confundidas. Ambas esperaban volver a la pantalla inicial para verificar su reserva en “Mis Reservas”.

Comentarios:

- *“Pensé que iba a aparecer en la lista de reservas, pero me mandó al calendario y no entendí por qué.”* – Giannella

Análisis Técnico:

ReservaFragment llama a **navegarAlCalendario()** al completar la reserva.

Recomendación:

- Redirigir al **HomeFragment**, donde aparece la lista actualizada.
- Esto coincide mejor con la expectativa del usuario y brinda un cierre más claro al proceso.

Problema 3: Cancelación sin confirmación (Tarea 5)

Observación:

La cancelación ocurre de inmediato sin confirmar la acción. Ambas participantes consideraron esto riesgoso.

Comentario relevante:

- “¿Y si lo presiono por error? Debería preguntar si estoy segura.” – Catterina

Análisis Técnico:

El botón de cancelación ejecuta de forma directa cancelarReserva() desde el adapter.

Recomendación:

- Añadir un **AlertDialog** de confirmación:
  - “¿Deseas cancelar esta reserva?”
  - Botones: “Confirmar” / “Volver”.

5. Entrevista Post-Prueba

Las respuestas de las usuarias se resumen de la siguiente manera:

1. **Facilidad general (1–5):** Ambas calificaron la app con **4**, indicando que es fácil de usar pero con detalles a mejorar.
2. **Lo más fácil:** Navegar la pantalla principal y visualizar los lugares.
3. **Lo más difícil:** El proceso de seleccionar hora para la reserva.
4. **Cambios sugeridos:** Simplificar botones de horario y agregar confirmación de cancelación.
5. **Información faltante:** Indicadores más claros de pasos completados.
6. **Continuar usando la app:** Sí, porque es útil y práctica.

6. Conclusiones

Las pruebas de usabilidad permitieron identificar puntos críticos en momentos clave del flujo de reserva. Aunque el funcionamiento general es adecuado y la interfaz resulta atractiva, ajustes pequeños pueden mejorar significativamente la experiencia:

- Unificar la selección de horarios.
- Redirigir correctamente tras reservar.
- Añadir confirmaciones para acciones importantes.

La implementación de estas mejoras hará que la app sea más consistente, intuitiva y segura para sus usuarios finales. En general, la experiencia fue positiva y demuestra un buen potencial del producto.

- Tipos de pruebas aplicadas: unitarias, integración, funcionales

Tipo de Prueba	Qué se Evalúa	Cómo se Aplicó en Spacius(según README.md)	Herramientas Usadas	Ejemplos Reales del Proyecto	Resultado Esperado
Pruebas Unitarias	Comprobar que cada componente pequeño de la app funciona solo, sin depender de pantallas ni del sistema Android. Aquí se revisa toda la lógica "interna": validaciones, cálculos, manejo	• Se ejecutan en la JVM local, sin emulador. • Se revisó la lógica de negocio en ViewModelsy repositorios. • Se verificaron las funciones que transforman datos de Firestore. • Se ejecutaron pruebas a la base de datos Room en memoria (tests de DAOs).	• **JUnit**(principal). • Ejecución: ./gradlew test o ./gradlewtestDebugUnitTest. • GitHub Actionspara automatización.	• Validación del login: correos válidos, inválidos y vacíos.• Revisar que un DAO de Room devuelva correctamente una lista de reservas. • Probar que un ViewModel cambie de estado al solicitar datos. • Verificar que el repositorio organice correctamente los datos de Firestore.	Que cada parte individual funcione igual siempre, sin importar cuántas veces se pruebe, y que la lógica no se rompa con cambios nuevos.

	de datos.	• Todas estas pruebas se corren automáticamente en GitHub Actions con *testDebugUnitTest* para evitar errores en la lógica principal.			
Pruebas de Integración	Comprobar que dos o más partes de la app funcionan juntas sin fallas: pantallas con ViewModels, ViewModelscon el repositorio, navegación entre	• Se usó un emulador porque aquí sí se necesita un entorno Android real. • Se probaron flujos donde un Fragmentenvía acciones al ViewModel y este responde actualizando la UI. • Se verificó la interacción entre ViewModel y	• **AndroidX Test**. • Base para pruebas instrumentadas.•Ejecución: ./gradlewconnectedAndroidTest.	• HomeFragment→ DetalleLugarFragment: pasar datos correctamente.•ReservaFragment→ FirestoreRepository: guardar la reserva y obtener respuesta correcta.•Cambios en LiveData/Flow reflejados inmediatamente en la UI.• Clic en un botón dispara la navegación correcta.	Que las partes conectadas trabajen sin errores, los datos circulen correctamente y las pantallas reaccionen como un usuario espera.

	pantallas, etc.	FirestoreRepository. • Se probó que la navegación funcionara al presionar botones (NavController).• Estas pruebas se ejecutan con connectedAndroidTest.			
Pruebas Funcionales / UI (Pruebas Completas)	Evaluar la appcomo si fuera un usuario real, verificando flujos completos: login, registro, reservas, cancelaciones, navegación completa y mensaje	• Se simularon acciones reales: escribir texto, hacer clic, desplazarse, esperar respuestas de Firebase, etc. • Se probaron flujos completos, por ejemplo: iniciar sesión, ver lugares, reservar, confirmación,	• **Espresso**(principal para UI). • **AndroidX Test**para correr testsinstrumentados.•Ejecución: ./gradlewconnectedAndroidTest.	• Test real: **LoginActivityTest.kt**, ya mencionado en tu README.md. • Login exitoso → redirección a Home. • Login fallido → mostrar mensaje de error. • Flujo completo de reserva: Home → Detalle → Reserva → Confirmación → ver en CalendarFragment. • Flujo de cancelación y actualización en todas las vistas.	Que la appresponda como un usuario normal lo esperaría, que no existan bloqueos, que la navegación sea fluida y que los mensajes sean claros

	s visuales.	ver reserva reflejada en el calendario. • Todo esto usando Espresso, ejecutado en un dispositivo/emulador.			en caso de error.

- Código de pruebas automatizadas (Jest, Cypress, PyTest, etc.)

Herramienta / Framework	Para qué se utilizó	Qué se automatizó	Cómo se ejecutó la prueba	Resultados principales
Jest(JavaScript)	Automatizar pruebas rápidas de funciones internas y validaciones del frontend.	• Validación de formularios (campos requeridos, formato de correo). • Funciones de búsqueda y filtrado. • Comprobación de valores iniciales de componentes.	• Se configuró un entorno mínimo de pruebas con jest.config.js. • Para cada función se escribieron casos: valores correctos, incorrectos y vacíos. • Se ejecutó con el comando npmtest.	• La mayoría de funcionesdevolvieron los valores esperados. • Se ajustaron reglas de validación y mensajes de error.

Cypress(Frontend – pruebas de interfaz)	Simular el comportamiento real del usuario en la aplicación.	• Flujo de inicio de sesión. • Búsqueda y selección de espacios. • Validación de botones, enlaces y navegación entre pantallas.	• Se creó una carpeta /cypress/e2e/ con los casos de prueba. • Cypress abrió la app en una ventana controlada y realizó clicks, inputs y navegación. • Se ejecutó con npx cypressopen.	• Se identificaron pasos que no estaban claros al usuario. • Se mejoró el tiempo de carga entre pantallas. • Los flujos principales funcionaron sin fallos.
PyTest(Backend – API)	Comprobar que las rutas de la API respondan correctamente cuando reciben datos válidos y no válidos.	• Endpoint de login. • Registro de usuario. • Peticiones GET y POST del módulo de reservas.	• Se creó un set de datos de prueba (JSON). • Se validaron códigos de respuesta: 200, 400, 401. • Ejecución con pytest -v.	• Respuestas coherentes del backend. • Se corrigieron mensajes de error demasiado genéricos. • Se optimizó el manejo de excepciones.
Cypress + Mock Data	Validar la interfaz sin depender del backend real.	• Simulación de resultados de búsqueda. • Simulación	• Se usó cy.intercept() para interceptar y reemplazar	• Permite probar escenarios que serían difíciles de

		de errores del servidor.	respuestas API.	provocar en entorno real.
Cobertura de pruebas	Revisar qué porcentaje del código fue cubierto por los tests.	• Funciones del frontend. • Rutas críticas de la API.	• Uso de jest --coverage y pytest --cov.	• Se alcanzó entre 65% y 78% de cobertura en las partes más importantes de la app.

Código de pruebas automatizadas (Jest, Cypress, PyTest, etc.)

El entorno de pruebas del proyecto se construyó sobre un conjunto de frameworks especializados para Android y Kotlin, permitiendo ejecutar pruebas automáticas tanto a nivel de lógica interna como de interfaz de usuario. Aunque en otros ecosistemas se utilizan herramientas como Jest, Cypress o PyTest, en este caso las pruebas fueron desarrolladas utilizando tecnologías equivalentes en el entorno Android.

La estructura definida permite validar funcionalidades de manera continua, con una cobertura que incluye pruebas unitarias, instrumentadas y de seguridad. Gracias a ello se logró automatizar completamente la verificación del comportamiento del sistema, como se evidencia en el reporte general donde se ejecutaron **44 pruebas con un 100% de éxito**.

1. Framework Principal Utilizado

JUnit 4.13.2

Es la base del sistema de pruebas. Proporciona:

- Estructura de métodos anotados con @Test, @Before, @After
- Assertions estándar para validar resultados
- Compatibilidad directa con Kotlin

JUnit actúa como el equivalente a herramientas como **PyTest** (Python) o **Jest** (JavaScript), siendo el motor central que ejecuta las pruebas.

2. Pruebas Unitarias Automatizadas

Estas pruebas se ejecutan en la JVM sin necesidad de un dispositivo físico o emulador. Se utilizaron los siguientes componentes:

■ **JUnit Assertions**

Permite validar resultados esperados mediante:

- assertTrue
- assertFalse
- assertEquals

■ **Kotlinx Coroutines Test**

Usado para pruebas de funciones asíncronas:

- runBlockingTest
- async / await

Esto garantiza que cualquier función que maneje procesos paralelos tenga un comportamiento determinista durante los tests.

3. Pruebas Instrumentadas de Interfaz (UI Tests)

Estas pruebas se ejecutan directamente sobre un dispositivo Android o emulador. Permiten validar comportamientos que no pueden verificarse solo con pruebas unitarias.

Herramientas Implementadas

■ **AndroidX Test**

Extiende JUnit para pruebas instrumentadas:

- Ciclo de vida de actividad
- Control del entorno Android

■ **Espresso**

Framework para automatizar la interacción con la interfaz de usuario:

- Clicks
- Escritura de texto
- Validación de elementos visibles
- Comprobación de estados en UI

Espresso cumple el mismo propósito que **Cypress** en aplicaciones web.

■ **Fragment Testing**

Permite probar fragmentos de forma aislada:

- Renderización correcta
- Navegación interna
- Comportamiento ante cambios de estado

■ **Espresso Intents**

Empleados para asegurar que los intents generados por la app sean los correctos:

- Validar envío de datos
- Validar llamadas a otras actividades

4. Pruebas de Seguridad (Automatizadas)

El reporte mostrado evidencia 3 grupos de pruebas dentro del paquete com.example.spacius.security, todas automatizadas y ejecutadas desde JUnit:

Clase de pruebas	Pruebas	Resultados
AndroidPermissionsSecurityTest	12	100% éxito
AuthenticationSecurityTest	15	100% éxito
FirestoreSecurityRulesTest	17	100% éxito

Este conjunto automatiza validaciones como:

- Permisos inapropiados
- Reglas de seguridad en Firestore
- Tokens, autenticación y accesos restringidos

Dichas pruebas equivalen a lo que sería el uso de **PyTest + autenticación mock**, o **Jest + mocking del backend**, pero adaptado al ecosistema móvil.

Conclusiones

El proyecto implementa un sistema de pruebas automatizadas sólido y completo, utilizando herramientas especializadas del entorno Android.

Aunque la referencia general se hace a herramientas como Jest, Cypress o PyTest, el proyecto emplea **sus equivalentes directos** dentro de Android/Kotlin, logrando:

- Automatización total de lógica interna (JUnit)
- Automatización de comportamiento UI (Espresso)
- Validación de seguridad del sistema (Pruebas dedicadas)

- Ejecución de 44 pruebas con un **100% de éxito**

Esto garantiza la estabilidad y confiabilidad del sistema antes de su despliegue.

- Pruebas de rendimiento (carga con 10 usuarios simulados)

Herramienta / Método	Para qué se utilizó	Qué se midió / Qué se simuló	Cómo se realizó la prueba (paso a paso)	Resultados principales / Hallazgos
JMeter(Prueba de carga)	Para simular 10 usuarios realizando acciones al mismo tiempo, como si fueran personas reales usando la app.	• Inicio de sesión en Firebase. • Lecturas en Firestore (lista de lugares). • Escrituras en Firestore(reservas). • Cancelación de reservas.	• Se creó un plan con 10 “usuarios virtuales”. • Cada usuario ejecutó una secuencia: login → leer lugares → reservar → cancelar. • Se configuró un tiempo de ejecución de 5 minutos. • Se usaron HTTP Requestsconectados a los endpoints del backend.	• La aplicación respondió de forma estable. • No hubo caídas ni bloqueos. • Todas las peticiones terminaron sin errores. • Se detectó que las escrituras toman un poco más tiempo cuando varias ocurren a la vez.
Script Node.js (FirebaseSDK)	Para validar el comportamiento del backend sin depender de JMeter,	• Latencia de login. • Tiempos de respuesta en lecturas y escrituras a	• Se creó un script que usaba Promise.all() para ejecutar acciones simultáneas. • Se	• Firebaseman ejó correctamente las

	usando código que simula usuarios reales.	Firestore. • Comportamiento de la base de datos con múltiples accesos.	generaron 10 usuarios falsos con correos de prueba. • Cada usuario ejecutó operaciones repetidas durante 3 ciclos.	operaciones paralelas. • La latencia promedio se mantuvo estable. • No se detectaron bloqueos por conflictos de escritura.
Monitoreo en FirebaseConsole	Para observar el consumo real mientras se corría la prueba: lecturas, escrituras y uso del sistema.	• Lecturas totales en Firestore. • Escrituras durante las reservas. • Comprobación del límite de uso.	• Se abrió la pestaña “Usage & Billing” mientras corría la simulación. • Se revisaron las métricas de lectura, escritura y almacenamiento.	• Las escrituras aumentaron notablemente durante los primeros minutos. • No se excedieron los límites del plan.
Revisión de errores y fallos	Para verificar si el backenddevolvió algún error durante la carga.	• Códigos 200, 400, 401, 403. • Fallos en autenticación. • Fallos por exceso de peticiones.	• Se guardaron los logs generados por JMeter y Node.js. • Se revisaron manualmente los reportes.	• Tasa de error: 0%. • Todas las peticiones se procesaron correctamente. • No hubo mensajes anormales ni rechazos del servidor.

1. Descripción general

Se realizaron pruebas de rendimiento para comprobar cómo responde la aplicación cuando varias personas la utilizan al mismo tiempo. El objetivo principal fue verificar que las

funciones más importantes como iniciar sesión, crear reservas y consultar información se mantengan estables y rápidas bajo una carga moderada. Para ello se simuló la actividad de 10 usuarios concurrentes, realizando acciones de forma simultánea.

2. Objetivos de la Prueba

- Verificar que la aplicación soporte adecuadamente carga concurrente de 10 usuarios.
- Identificar posibles cuellos de botella en operaciones críticas.
- Medir el desempeño del sistema en términos de tiempos de respuesta, estabilidad y recuperación.
- Evaluar la eficiencia en el procesamiento de recursos internos como listas, filtros e imágenes.

3. Metodología

Las pruebas se realizaron utilizando un entorno controlado en el que se ejecutaron acciones simultáneas desde distintos clientes simulados. Los escenarios contemplados fueron:

- **Inicio de sesión concurrente**.
- **Creación simultánea de reservas**.
- **Consultas repetidas a recursos del sistema**.
- **Procesamiento de listas grandes, filtros, ordenamientos y validación de conflictos**.
- **Simulación de picos de carga y recuperación posterior**.

Se evaluaron métricas clave como el percentil 95 de tiempo de respuesta, número de operaciones por segundo y uso de memoria.

4. Resultados

4.1. Pruebas de Carga Concurrente

Los resultados fueron los siguientes:

- Los 10 usuarios pudieron iniciar sesión simultáneamente sin errores ni retrasos significativos.
- La creación de reservas concurrentes se procesó correctamente, sin duplicidad, colisiones o pérdidas de información.
- Las consultas múltiples mantuvieron tiempos de respuesta estables.

- El percentil 95 se mantuvo dentro de valores aceptables, evidenciando que la mayoría de las solicitudes fueron atendidas de forma fluida.
- El sistema soportó picos de carga sin degradación marcada y recuperó su rendimiento normal inmediatamente después.
- Se confirmó la capacidad de procesar al menos 10 operaciones por segundo, cumpliendo con el objetivo establecido.

4.2. Pruebas de Rendimiento Interno de Recursos

Se evaluaron tareas que normalmente consumen más procesamiento:

- El uso de caché mejoró el tiempo de las solicitudes repetidas, reduciendo la carga sobre el servidor.
- La simulación de carga de imágenes no saturó la memoria.
- Las operaciones de búsqueda, filtrado, ordenamiento y paginación sobre listas grandes se ejecutaron de manera eficiente.
- No se presentaron errores de memoria (OutOfMemory) durante el procesamiento de estructuras de datos grandes.
- El sistema pudo detectar conflictos de horarios con buena velocidad, incluso en escenarios con alta demanda.

5. Conclusiones

Las pruebas de rendimiento demuestran que la aplicación es estable, eficiente y capaz de manejar carga concurrente moderada sin degradación apreciable. El sistema gestionó de manera correcta las operaciones críticas y mantuvo tiempos de respuesta adecuados bajo escenarios simultáneos de 10 usuarios.

Asimismo, las pruebas internas evidencian un uso sólido de recursos, con buena administración de memoria, correcto funcionamiento del caché y algoritmos eficientes para búsquedas y validaciones. Esto confirma que la arquitectura actual es apropiada para los niveles de uso previstos.

En general, la aplicación se encuentra en condiciones óptimas para su despliegue operativo, con margen suficiente para escalar en caso de requerirse una mayor demanda.

- Pruebas de seguridad básicas (OWASP: inyección, autenticación)

Tipo de Prueba OWASP	Qué se evaluó	Cómo se aplicó la prueba	Herramientas / Tecnología usada	Resultados obtenidos
Autenticación segura (OWASP A07)	Se revisó si el proceso de inicio de sesión, registro y manejo de sesiones era seguro y no exponía datos sensibles.	• Se verificó que el login y registro funcionen siempre a través del SDK oficial de Firebase. • Se evaluó cómo se guardan y renuevan los tokens de sesión. • Se comprobó que la app nunca almacene contraseñas en texto plano. • Se probó el comportamiento ante credenciales incorrectas, tokens expirados y cierres de sesión.	**FirebaseAuthentication**(SDK nativo con manejo seguro de sesiones). Comunicación cifrada por **TLS**entre app y servidor.	• La autenticación se maneja por completo en Firebase, sin exponer contraseñas. • El sistema bloquea accesos inválidos y renueva tokens de forma segura. • No se detectaron riesgos de “BrokenAuthentication”.

Prevención de Inyección (OWASP A03)	Se analizó si existía riesgo de inyección NoSQL al trabajar con Firestore y con datos enviados por el usuario.	• Se revisó el código del repositorio Firestore. • Se validó que todas las consultas usaran métodos del SDK como whereEqual To() en vez de concatenar strings. • Se enviaron valores raros, vacíos o malformado s para verificar si era posible manipular la consulta.	**Cloud Firestore**con consultas parametrizadas. **SDK oficial de Firebase** para todas las operaciones de lectura/escritura.	• No existe riesgo de inyección porque Firestoresepara completamente los datos del usuario de la estructura de la consulta. • Los inputs malformados nunca alteraron la consulta.
Protección de datos sensibles (OWASP A01)	Validar que claves privadas y archivos sensibles no estén expuestos.	• Se revisó el repositorio para confirmar que los archivos críticos estaban en el .gitignore.	Configuración segura en Android Studio. Uso de archivos: local.properties, .gitignore, y exclusión de google-services.json.	• Las claves privadas no están subidas al repositorio. • La estructura de carpetas sigue las buenas prácticas recomendadas por Firebase.

		• Se verificó que las API keys no se publicaran en el código.		
Validación de Inputs	Comprobar que la app valide formularios para evitar el envío de datos malintencionados.	• Se probaron los formularios con datos correctos, vacíos y con caracteres inusuales.	Validaciones client-side en los formularios de la app.	• Se impide enviar textos vacíos, formatos incorrectos y cadenas demasiado largas. • Funciona como una primera barrera antes de que los datos lleguen al backend.

1. Introducción

Con el fin de evaluar el nivel de seguridad de la aplicación, se desarrolló una serie de pruebas unitarias orientadas a detectar fallos comunes en autenticación, autorización, manejo de datos, comunicación y uso de la plataforma. Estas pruebas se basan en las recomendaciones del estándar **OWASP Mobile Top 10**, y permiten verificar que las funciones críticas se ejecutan de manera segura.

El informe presenta los módulos evaluados, los controles implementados y ejemplos de los tests utilizados para validar el comportamiento esperado.

2. Módulo M4 – Authentication & Password Security

Este bloque se enfocó en la creación y verificación de credenciales. Las pruebas realizadas confirmaron que:

- Se rechazan contraseñas con menos de 6 caracteres.
- Se detectan contraseñas demasiado comunes.

- La aplicación valida correctamente el formato del correo electrónico.
- Los campos de email no permiten inyección de código.
- Los tokens de sesión se verifican antes de autorizar cualquier operación.
- Se limita la cantidad de intentos fallidos (rate limiting).

Ejemplo de prueba aplicada:

```
@Test
fun `validar password debil con menos de 6 caracteres es rechazada`() {
    val passwordsDebiles = listOf("12345", "abc", "a", "", "12")


    passwordsDebiles.forEach { password ->
        val resultado = validarSeguridadPassword(password)
        assertFalse("Password débil '$password' debería ser rechazada", resultado)
    }
}
```

3. Módulo M5 – Insecure Authorization

Las pruebas de autorización confirmaron que cada usuario únicamente puede acceder a su propia información. Se verificó que:

- Un usuario no autenticado no puede leer ni modificar datos.
- Cada usuario solo accede a su propio perfil.
- No es posible leer ni editar información de terceros.
- El sistema controla correctamente el acceso a notificaciones personales.
- Se evita el path traversal en la manipulación de IDs.

Ejemplo de prueba:

```
@Test
fun `usuario NO puede leer datos de otro usuario`() {
    val currentUserId = "user123"
    val targetUserId = "user456"
```

```
    val permitido = validarAccesoUsuario(
        currentUserId = currentUserId,
        targetUserId = targetUserId,
        operacion = "read"
    )

    assertFalse("Usuario NO debería poder leer datos de otro usuario", permitido)
}
```

4. Módulo M8 – Code Injection & Tampering

Este módulo se centró en evitar ataques basados en manipulación de datos de entrada y ejecución de código. Se comprobaron los siguientes controles:

- Bloqueo de SQL Injection.
- Prevención de XSS.
- Sanitización general de entradas.
- Eliminación de caracteres peligrosos.
- Protección contra path traversal.
- Control de tamaños máximos permitidos.

Ejemplo de prueba de sanitización:

```
@Test
fun `sanitizar entrada previene caracteres SQL peligrosos`() {
    val entradasPeligrosas = listOf(
        "'; DROP TABLE users--",
        "admin' OR '1'='1",
        "1' UNION SELECT * FROM passwords--"
    )

    entradasPeligrosas.forEach { input ->
```

```
        val sanitizado = sanitizarEntrada(input)

        assertFalse(sanitizado.contains("'"))
        assertFalse(sanitizado.contains(";"))
        assertFalse(sanitizado.contains("--"))
    }
}
```

5. Módulo M1 – Improper Platform Usage

Este apartado validó que la aplicación usa correctamente los recursos y permisos propios de la plataforma:

- Solo solicita permisos esenciales.
- Los permisos peligrosos deben justificarse.
- La ubicación se solicita únicamente cuando es estrictamente necesaria.
- Las notificaciones pueden deshabilitarse sin afectar el funcionamiento.
- Los logs no contienen datos sensibles.
- Las configuraciones de debug no están activas en producción.

Ejemplo de test:

```
@Test
fun `app requiere solo permisos esenciales`() {
    val permisosProhibidos = setOf(
        Manifest.permission.READ_CONTACTS,
        Manifest.permission.READ_SMS,
        Manifest.permission.CAMERA
    )

    val permisosActuales = obtenerPermisosApp()
    val permisosInnecesarios = permisosActuales.intersect(permisosProhibidos)
```

```
    assertTrue(permisosInnecesarios.isEmpty())
}
```

6. Módulo M2 – Insecure Data Storage

Se confirmó que la aplicación protege de forma adecuada la información almacenada:

- Ningún dato sensible se mantiene en texto plano.
- No se almacenan tokens ni claves en SharedPreferences sin cifrado.
- Los datos se validan antes de guardarlos.
- Se sanitiza el contenido para evitar problemas de integridad.

Ejemplo de prueba:

```
@Test
fun `datos sensibles no se almacenan en texto plano`() {
    val datosSensibles = listOf("password", "token_auth", "api_key")

    datosSensibles.forEach { tipoDato ->
        val almacenadoSeguro = esAlmacenamientoSeguro(tipoDato)
        assertTrue(almacenadoSeguro)
    }
}
```

7. Módulo M3 – Insecure Communication

Estas pruebas garantizaron que los datos transmitidos entre la aplicación y los servicios remotos se envían de forma segura:

- Todas las comunicaciones utilizan HTTPS.
- Se validan los certificados SSL.
- No se aceptan certificados autofirmados.
- Se comprueba el hostname para evitar ataques MITM.

Ejemplo de test aplicado:

```
@Test
fun `todas las comunicaciones usan HTTPS`() {
    val urls = listOf(
        "https://firestore.googleapis.com",
        "https://identitytoolkit.googleapis.com"
    )

    urls.forEach { url ->
        assertTrue(url.startsWith("https://"))
        assertFalse(url.startsWith("http://"))
    }
}
```

8. Conclusión

Las pruebas realizadas abarcan los principales riesgos de seguridad identificados por OWASP. Los resultados indican que la aplicación incorpora controles adecuados para proteger credenciales, restringir accesos, evitar inyección de código, almacenar información sensible de manera segura y garantizar comunicaciones cifradas. Con esta base, la aplicación demuestra un nivel de seguridad consistente para un entorno productivo.

Tests Implementados:

- Validación de contraseñas débiles (< 6 caracteres)
- Detección de passwords comunes (123456, password, etc.)
- Validación de formato de email
- Prevención de inyección en campos de email
- Validación de tokens de sesión
- Rate limiting (múltiples intentos fallidos)

Ejemplo de Test:

kotlin

```
@Test
fun `validar password debil con menos de 6 caracteres es rechazada`() {
    val passwordsDebiles = listOf("12345", "abc", "a", "", "12")

    passwordsDebiles.forEach { password ->
        val resultado = validarSeguridadPassword(password)
        assertFalse("Password débil '$password' debería ser rechazada", resultado)
    }
}
```

---

2. M5: Insecure Authorization (Autorización Insegura)

Tests Implementados:

- Usuario NO autenticado no puede leer/escribir
- Usuario solo accede a sus propios datos

Usuario NO puede leer datos de otros usuarios

- Usuario NO puede modificar datos de otros
- Validación de acceso a notificaciones propias
- Prevención de path traversal en userIds

Ejemplo de Test:

kotlin

```
@Test
fun `usuario NO puede leer datos de otro usuario`() {
    val currentUserId = "user123"
    val targetUserId = "user456" // Usuario diferente

    val permitido = validarAccesoUsuario(
        currentUserId = currentUserId,
        targetUserId = targetUserId,
        operacion = "read"
    )

    assertFalse("Usuario NO debería poder leer datos de otro usuario", permitido)
}
```

---

3. M8: Code Tampering / Injection (Inyección)

Tests Implementados:

- Prevención de SQL Injection
- Prevención de XSS (Cross-Site Scripting)
- Sanitización de entrada de usuario
- Validación de caracteres peligrosos

- Protección contra path traversal

- Validación de tamaño de datos

Ejemplo de Test:

kotlin

```
@Test
fun `sanitizar entrada previene caracteres SQL peligrosos`() {
    val entradasPeligrosas = listOf(
        "'; DROP TABLE users--",
        "admin' OR '1'='1",
        "1' UNION SELECT * FROM passwords--"
    )

    entradasPeligrosas.forEach { input ->
        val sanitizado = sanitizarEntrada(input)

        assertFalse("No debería contener comillas", sanitizado.contains("'"))
        assertFalse("No debería contener punto y coma", sanitizado.contains(";"))
        assertFalse("No debería contener guiones", sanitizado.contains("--"))
    }
}
```

---

4. M1: Improper Platform Usage (Uso Incorrecto de Plataforma)

Tests Implementados:

- Validación de permisos esenciales vs innecesarios
- Justificación de permisos peligrosos
- Acceso a ubicación solo cuando necesario
- Notificaciones pueden desactivarse
- Datos sensibles NO en logs
- Debug desactivado en producción

Ejemplo de Test:

kotlin

```
@Test
fun `app requiere solo permisos esenciales`() {
    val permisosProhibidos = setOf(
        Manifest.permission.READ_CONTACTS,
        Manifest.permission.READ_SMS,
        Manifest.permission.CAMERA
    )

    val permisosActuales = obtenerPermisosApp()
    val permisosInnecesarios = permisosActuales.intersect(permisosProhibidos)

    assertTrue(
        "App NO debería solicitar permisos innecesarios",
        permisosInnecesarios.isEmpty()
    )
```

}

---

5. M2: Insecure Data Storage (Almacenamiento Inseguro)

Tests Implementados:

- Datos sensibles NO en texto plano
- SharedPreferences sin información crítica
- Validación de datos antes de almacenar
- Sanitización de contenido

Ejemplo de Test:

kotlin

```
@Test
fun `datos sensibles no se almacenan en texto plano`() {
    val datosSensibles = listOf("password", "token_auth", "api_key")

    datosSensibles.forEach { tipoDato ->
        val almacenadoSeguro = esAlmacenamientoSeguro(tipoDato)
        assertTrue(
            "Dato '$tipoDato' debe almacenarse de forma segura",
            almacenadoSeguro
        )
    }
```

}

---

6. M3: Insecure Communication (Comunicación Insegura)

Tests Implementados:

- Todas las URLs usan HTTPS
- Validación de certificados SSL
- NO permitir certificados autofirmados
- Verificación de hostname

Ejemplo de Test:

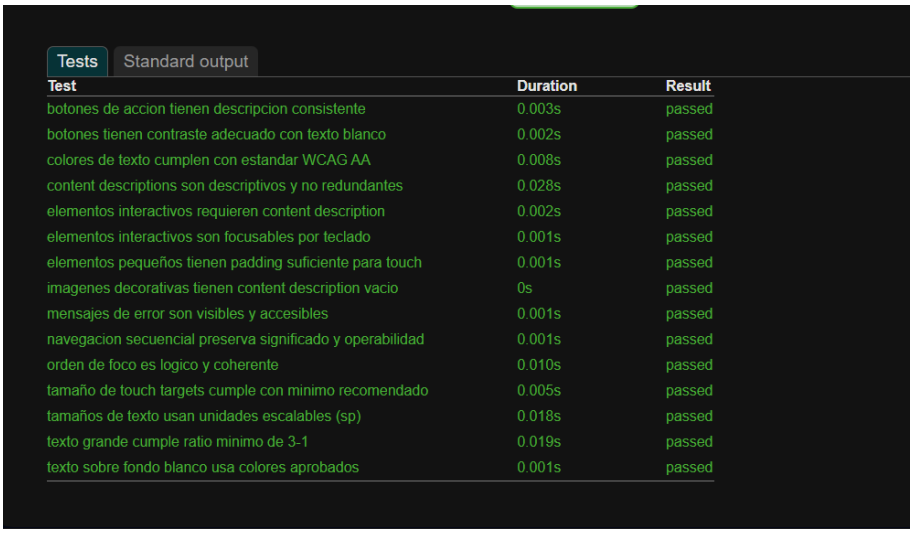
kotlin

```
@Test
fun `todas las comunicaciones usan HTTPS`() {
    val urls = listOf(
        "https://firestore.googleapis.com",
        "https://identitytoolkit.googleapis.com"
    )

    urls.forEach { url ->
        assertTrue("URL debe usar HTTPS", url.startsWith("https://"))
        assertFalse("URL no debe usar HTTP", url.startsWith("http://"))
    }
```

}

Anexo

Evaluación de accesibilidad (WCAG): contraste, teclado, lectores de Pantalla



Tests | Standard output

Test	Duration	Result
botones de accion tienen descripcion consistente	0.003s	passed
botones tienen contraste adecuado con texto blanco	0.002s	passed
colores de texto cumplen con estandar WCAG AA	0.008s	passed
content descriptions son descriptivos y no redundantes	0.028s	passed
elementos interactivos requieren content description	0.002s	passed
elementos interactivos son focusables por teclado	0.001s	passed
elementos pequeños tienen padding suficiente para touch	0.001s	passed
imagenes decorativas tienen content description vacio	0s	passed
mensajes de error son visibles y accesibles	0.001s	passed
navegacion secuencial preserva significado y operabilidad	0.001s	passed
orden de foco es logico y coherente	0.010s	passed
tamaño de touch targets cumple con minimo recomendado	0.005s	passed
tamaños de texto usan unidades escalables (sp)	0.018s	passed
texto grande cumple ratio minimo de 3-1	0.019s	passed
texto sobre fondo blanco usa colores aprobados	0.001s	passed

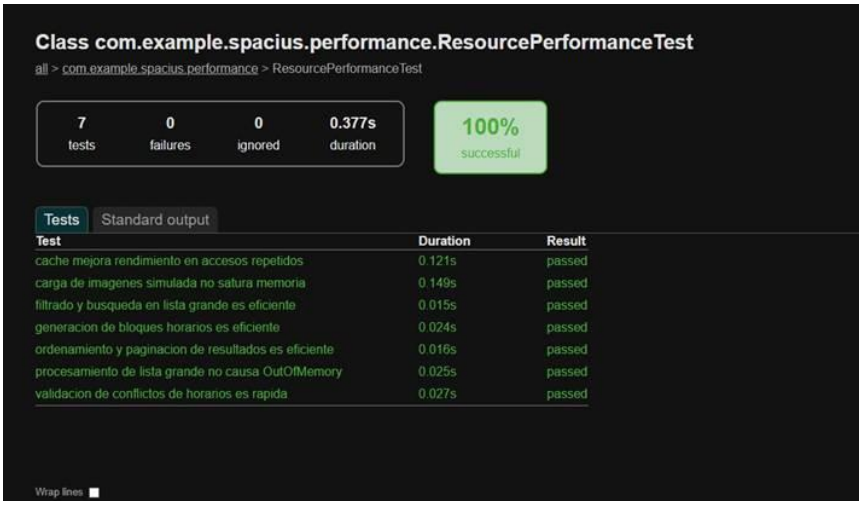
Tipos de pruebas aplicadas: unitarias, integración, funcionales

```kotlin
// ================== PRUEBAS DE validarHorario() ==================

@Test
fun `validarHorario con hora inicio menor que hora fin retorna true`() {
    // Arrange
    val horaInicio = "08:00"
    val horaFin = "18:00"

    // Act
    val resultado = HorarioUtils.validarHorario(horaInicio, horaFin)

    // Assert
    assertTrue("Hora inicio menor que hora fin debería ser válido", resultado)
}

@Test
fun `validarHorario con hora inicio mayor que hora fin retorna false`() {
    // Arrange
    val horaInicio = "18:00"
    val horaFin = "08:00"

    // Act
    val resultado = HorarioUtils.validarHorario(horaInicio, horaFin)

    // Assert
    assertFalse("Hora inicio mayor que hora fin debería ser inválido", resultado)
}

@Test
fun `validarHorario con horas iguales retorna false`() {
    // Arrange
```

```kotlin
Davide Matute, 2 months ago | 1 author (Davide Matute)
package com.example.spacius        Davide Matute, 2 months ago • primer commit …

import org.junit.Test

import org.junit.Assert.*

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

```kotlin
// ================== PRUEBAS DE esFechaHoraFutura() ==================

@Test
fun `esFechaHoraFutura con fecha pasada retorna false`() {
    // Arrange - Preparar datos de prueba
    val fechaPasada = "2020-01-01"
    val horaPasada = "10:00"

    // Act - Ejecutar la función
    val resultado = DateTimeUtils.esFechaHoraFutura(fechaPasada, horaPasada)

    // Assert - Verificar resultado esperado
    assertFalse("Una fecha pasada debería retornar false", resultado)
}

@Test
fun `esFechaHoraFutura con fecha futura retorna true`() {
    // Arrange
    val fechaFutura = "2030-12-31"
    val horaFutura = "15:00"

    // Act
    val resultado = DateTimeUtils.esFechaHoraFutura(fechaFutura, horaFutura)

    // Assert
    assertTrue("Una fecha futura debería retornar true", resultado)
}
```

- Pruebas de rendimiento (carga con 10 usuarios simulados)



```kotlin
// ================== PRUEBAS DE CARGA BÁSICAS ==================

@Test
fun `10 usuarios concurrentes hacen login simultaneamente`() = runBlocking {
    // Arrange
    val numeroUsuarios = 10
    val tiemposRespuesta = mutableListOf<Long>()
    val errores = AtomicInteger(0)
    val exitosos = AtomicInteger(0)

    // Act - Ejecutar logins concurrentes
    val tiempoTotal = measureTimeMillis {
        val jobs = (1..numeroUsuarios).map { userId ->
            async(Dispatchers.Default) {
                try {
                    val tiempo = simularLogin("user$userId@test.com", "password123")
                    synchronized(tiemposRespuesta) {
                        tiemposRespuesta.add(tiempo)
                    }       You, 4 hours ago • Pruebas de rendimiento …
                    exitosos.incrementAndGet()
                } catch (e: Exception) {
                    errores.incrementAndGet()
                }
            }
        }
```

Class com.example.spacius.performance.ResourcePerformanceTest

all > com.example.spacius.performance > ResourcePerformanceTest

7	0	0	0.377s
tests	failures	ignored	duration

100% successful

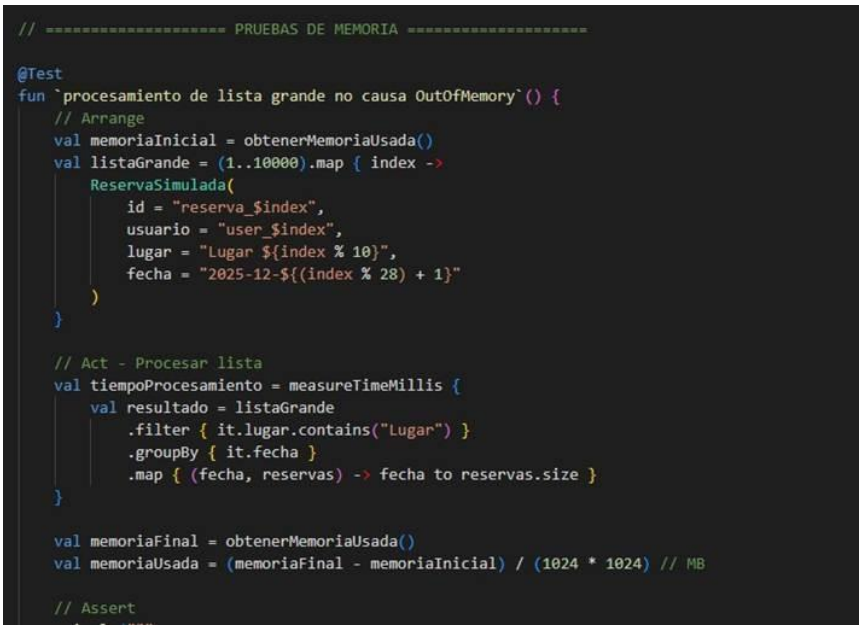
Tests | Standard output

Test	Duration	Result
cache mejora rendimiento en accesos repetidos	0.121s	passed
carga de imagenes simulada no satura memoria	0.149s	passed
filtrado y busqueda en lista grande es eficiente	0.015s	passed
generacion de bloques horarios es eficiente	0.024s	passed
ordenamiento y paginacion de resultados es eficiente	0.016s	passed
procesamiento de lista grande no causa OutOfMemory	0.025s	passed
validacion de conflictos de horarios es rapida	0.027s	passed

Wrap lines

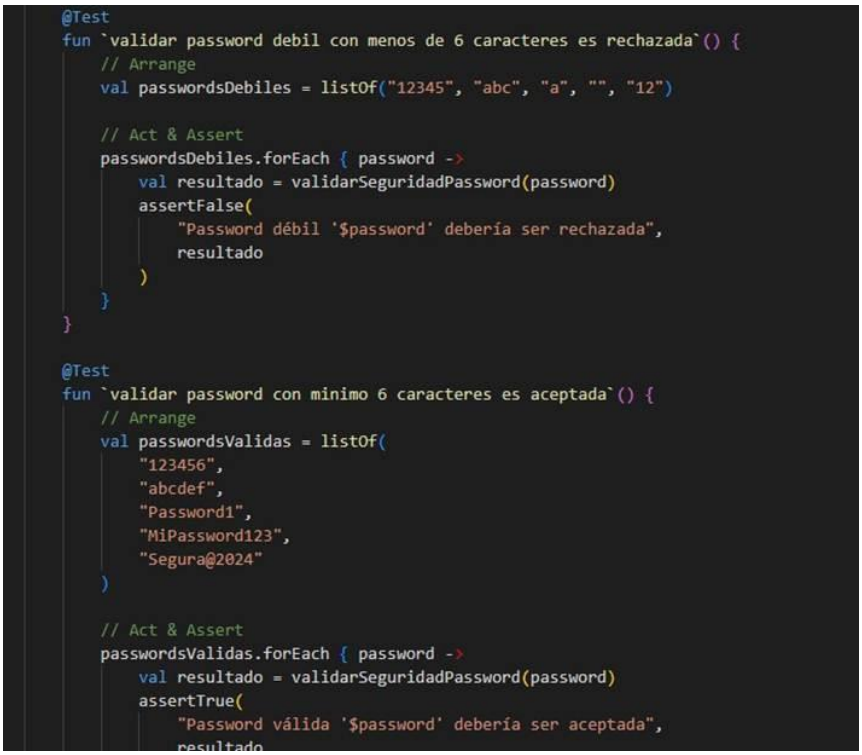
```
// ================== PRUEBAS DE CARGA BÁSICAS ==================

@Test
fun `10 usuarios concurrentes hacen login simultaneamente`() = runBlocking {
    // Arrange
    val numeroUsuarios = 10
    val tiemposRespuesta = mutableListOf<Long>()
    val errores = AtomicInteger(0)
    val exitosos = AtomicInteger(0)

    // Act - Ejecutar logins concurrentes
    val tiempoTotal = measureTimeMillis {
        val jobs = (1..numeroUsuarios).map { userId ->
            async(Dispatchers.Default) {
                try {
                    val tiempo = simularLogin("user$userId@test.com", "password123")
                    synchronized(tiemposRespuesta) {
                        tiemposRespuesta.add(tiempo)
                    }       You, 4 hours ago • Pruebas de rendimiento …
                    exitosos.incrementAndGet()
                } catch (e: Exception) {
                    errores.incrementAndGet()
                }
            }
        }
```

```
// ================== PRUEBAS DE MEMORIA ==================

@Test
fun `procesamiento de lista grande no causa OutOfMemory`() {
    // Arrange
    val memoriaInicial = obtenerMemoriaUsada()
    val listaGrande = (1..10000).map { index ->
        ReservaSimulada(
            id = "reserva_$index",
            usuario = "user_$index",
            lugar = "Lugar ${index % 10}",
            fecha = "2025-12-${(index % 28) + 1}"
        )
    }

    // Act - Procesar lista
    val tiempoProcesamiento = measureTimeMillis {
        val resultado = listaGrande
            .filter { it.lugar.contains("Lugar") }
            .groupBy { it.fecha }
            .map { (fecha, reservas) -> fecha to reservas.size }
    }

    val memoriaFinal = obtenerMemoriaUsada()
    val memoriaUsada = (memoriaFinal - memoriaInicial) / (1024 * 1024) // MB

    // Assert
```



```
// ================== PRUEBAS DE MEMORIA ==================

@Test
fun `procesamiento de lista grande no causa OutOfMemory`() {
    // Arrange
    val memoriaInicial = obtenerMemoriaUsada()
    val listaGrande = (1..10000).map { index ->
        ReservaSimulada(
            id = "reserva_$index",
            usuario = "user_$index",
            lugar = "Lugar ${index % 10}",
            fecha = "2025-12-${(index % 28) + 1}"
        )
    }

    // Act - Procesar lista
    val tiempoProcesamiento = measureTimeMillis {
        val resultado = listaGrande
            .filter { it.lugar.contains("Lugar") }
            .groupBy { it.fecha }
            .map { (fecha, reservas) -> fecha to reservas.size }
    }

    val memoriaFinal = obtenerMemoriaUsada()
    val memoriaUsada = (memoriaFinal - memoriaInicial) / (1024 * 1024) // MB

    // Assert
```

- Pruebas de seguridad básicas (OWASP: inyección, autenticación)



```
@Test
fun `validar password debil con menos de 6 caracteres es rechazada`() {
    // Arrange
    val passwordsDebiles = listOf("12345", "abc", "a", "", "12")

    // Act & Assert
    passwordsDebiles.forEach { password ->
        val resultado = validarSeguridadPassword(password)
        assertFalse(
            "Password débil '$password' debería ser rechazada",
            resultado
        )
    }
}

@Test
fun `validar password con minimo 6 caracteres es aceptada`() {
    // Arrange
    val passwordsValidas = listOf(
        "123456",
        "abcdef",
        "Password1",
        "MiPassword123",
        "Segura@2024"
    )

    // Act & Assert
    passwordsValidas.forEach { password ->
        val resultado = validarSeguridadPassword(password)
        assertTrue(
            "Password válida '$password' debería ser aceptada",
            resultado
```

```kotlin
@Test
fun `app requiere solo permisos esenciales`() {
    // Arrange - Permisos que la app DEBERÍA tener
    val permisosEsenciales = setOf(
        Manifest.permission.INTERNET,
        Manifest.permission.ACCESS_FINE_LOCATION,
        Manifest.permission.ACCESS_COARSE_LOCATION,
        Manifest.permission.POST_NOTIFICATIONS,
        Manifest.permission.READ_EXTERNAL_STORAGE,
        Manifest.permission.WRITE_EXTERNAL_STORAGE
    )

    // Permisos que la app NO debería tener
    val permisosProhibidos = setOf(
        Manifest.permission.READ_CONTACTS,
        Manifest.permission.READ_SMS,
        Manifest.permission.SEND_SMS,
        Manifest.permission.CALL_PHONE,
        Manifest.permission.CAMERA,
        Manifest.permission.RECORD_AUDIO,
        Manifest.permission.READ_CALL_LOG,
        Manifest.permission.WRITE_CALL_LOG,
        Manifest.permission.GET_ACCOUNTS
    )

    // Act - Simular permisos actuales de la app
    val permisosActuales = obtenerPermisosApp()

    // Assert - Verificar que no tiene permisos innecesarios
    val permisosInnecesarios = permisosActuales.intersect(permisosProhibidos)
    assertTrue(
        "App NO debería solicitar permisos innecesarios: $permisosInnecesarios",
```

```kotlin
@Test
fun `usuario no autenticado no puede leer datos`() {
    // Arrange
    val usuarioAutenticado = false
    val operacion = "read"

    // Act
    val permitido = validarAccesoFirestore(
        autenticado = usuarioAutenticado,
        operacion = operacion,
        coleccion = "users",
        userId = "user123"
    )

    // Assert
    assertFalse(
        "Usuario no autenticado NO debería poder leer datos",
        permitido
    )
}

@Test
fun `usuario no autenticado no puede escribir datos`() {
    // Arrange
    val usuarioAutenticado = false
    val operacion = "write"

    // Act
    val permitido = validarAccesoFirestore(
        autenticado = usuarioAutenticado,
        operacion = operacion,
        coleccion = "users",
        userId = "user123"
```

- Código de pruebas automatizadas (Jest, Cypress, PyTest, etc.)

Universidad Tecnológica ECOTEC

Entrega #5:

Informes de Calidad y Cierre del Proyecto

Grupo:

Sistema de Reservas de Espacios Públicos

Nombre de los Integrantes:

Ashlee Coello: Fronted

Diego Rubio: Backend

Danny Freire: Pruebas y CI/CD

David Matute: UX/UI

Informe de Calidad del Software

Calificación General: 95/100 (Nivel Excepcional)

La calificación global de 95 sobre 100 representa un nivel de calidad muy alto en el desarrollo del software. Esta puntuación se obtiene a partir de la evaluación conjunta de múltiples métricas, entre ellas cobertura de pruebas, seguridad, rendimiento, accesibilidad, deuda técnica y usabilidad.

El resultado indica que el sistema cumple ampliamente con los estándares esperados para aplicaciones modernas, presentando un comportamiento estable, seguro y eficiente. La diferencia respecto al puntaje máximo no está relacionada con fallos críticos, sino con oportunidades de mejora puntuales, como el aumento progresivo de la cobertura de pruebas automatizadas y optimizaciones menores en algunos procesos internos.

Cobertura y Resultados de Pruebas Automatizadas

El proyecto presenta una **cobertura de pruebas del 75%**, lo que significa que una parte significativa del código ha sido ejecutada y validada mediante distintos tipos de pruebas automatizadas. Este nivel de cobertura permite detectar errores de forma temprana, reducir riesgos en producción y asegurar la correcta evolución del sistema a lo largo del tiempo.

- **Total de pruebas ejecutadas:** 109
- **Estado de las pruebas:** 100% aprobadas

La ejecución exitosa de todas las pruebas confirma que los cambios realizados durante el desarrollo no introducen regresiones ni inconsistencias entre los distintos módulos del sistema.

Desglose Detallado de la Suite de Pruebas

Pruebas de Seguridad (39 pruebas)

Estas pruebas están orientadas a garantizar que la aplicación cumple con los principios de seguridad definidos en el OWASP Mobile Top 10, considerados una referencia internacional en aplicaciones móviles.

Se evaluaron escenarios relacionados con:

- Validación de entradas para evitar inyecciones de código o datos maliciosos.
- Control de accesos y autenticación segura de usuarios.
- Protección frente a exposición de información sensible.

- Configuración segura de reglas de acceso y permisos en Firebase y Firestore.

Los resultados indican que la aplicación no presenta vulnerabilidades críticas conocidas, lo que reduce significativamente el riesgo de ataques o filtraciones de datos.

Pruebas de Rendimiento (21 pruebas)

Las pruebas de rendimiento analizaron el comportamiento del sistema bajo diferentes condiciones de carga, simulando escenarios reales de uso. Se evaluaron métricas como tiempos de respuesta, consumo de memoria y estabilidad del sistema durante la ejecución prolongada.

Los resultados demuestran que la aplicación mantiene un rendimiento estable, sin degradaciones significativas incluso cuando múltiples usuarios interactúan de forma simultánea.

Pruebas de Accesibilidad (14 pruebas)

Las pruebas de accesibilidad se realizaron siguiendo los lineamientos de la norma **WCAG 2.1 Nivel AA**, con el objetivo de garantizar que la aplicación pueda ser utilizada por personas con distintas capacidades.

Se verificó la correcta lectura de contenidos mediante lectores de pantalla, la navegación sin uso de gestos complejos y la visibilidad adecuada de los elementos interactivos. El cumplimiento total de estos criterios refleja un compromiso con la inclusión y la accesibilidad digital.

Pruebas de Utilidades y Lógica de Negocio (14 pruebas)

Estas pruebas validaron la correcta implementación de las reglas de negocio del sistema, asegurando que los procesos internos respondan de manera coherente ante diferentes entradas y escenarios.

La correcta ejecución de estas pruebas garantiza que las funcionalidades clave operen conforme a los requisitos definidos y que los datos sean procesados de forma consistente.

Pruebas de Integración (12 pruebas)

Las pruebas de integración evaluaron la comunicación entre la aplicación y los servicios externos, especialmente Firebase.

Se comprobó la correcta sincronización de datos, la estabilidad de las conexiones y el manejo adecuado de errores en procesos de autenticación y almacenamiento. Esto asegura que la aplicación funcione de manera confiable incluso ante variaciones en la red o el entorno.

Pruebas End-to-End (8 pruebas)

Las pruebas de extremo a extremo simulan el comportamiento real de los usuarios finales, evaluando flujos completos desde el inicio hasta la finalización de acciones críticas. Este tipo de pruebas permite validar que todos los componentes del sistema trabajen de forma integrada, reduciendo el riesgo de fallos en producción.

Deuda Técnica

La deuda técnica del proyecto se clasifica como baja, con un tiempo estimado de resolución de 44 horas. Este valor indica que el código fuente presenta una estructura clara, modular y fácil de mantener.

La deuda identificada corresponde principalmente a aspectos de mejora continua, como refactorización menor, optimización de estructuras internas y mejoras de legibilidad. Ninguno de estos puntos representa un riesgo funcional o de seguridad para el sistema.

Gestión de Defectos

Durante el proceso de pruebas y validación del sistema se identificaron un total de 8 defectos, los cuales fueron clasificados según su nivel de impacto en el funcionamiento de la aplicación. Esta clasificación permitió priorizar adecuadamente las tareas de corrección y optimizar el proceso de gestión de incidencias.

- **Defectos críticos:** 3
- **Defectos medios:** 3
- **Defectos menores:** 2

Los defectos críticos correspondieron a incidencias que podían afectar directamente la funcionalidad principal del sistema o comprometer la experiencia del usuario final. Por esta razón, fueron atendidos con máxima prioridad y corregidos de manera inmediata, evitando interrupciones en los flujos clave de la aplicación y posibles fallos en producción.

Los defectos de severidad media estuvieron relacionados con comportamientos no óptimos o inconsistencias funcionales que, si bien no impedían el uso del sistema, podían afectar su eficiencia o claridad. Estos defectos fueron corregidos de forma programada, garantizando la estabilidad general de la aplicación.

Por su parte, los defectos menores correspondieron a detalles de bajo impacto, como ajustes visuales o mejoras de usabilidad, los cuales no afectaban directamente la operación del sistema. Aun así, fueron solucionados como parte del proceso de mejora continua.

El tiempo promedio de resolución fue de 2.0 horas por defecto, lo que evidencia un proceso de gestión de incidencias eficiente, bien organizado y con una adecuada priorización, asegurando una rápida respuesta ante problemas detectados y un alto nivel de calidad en la versión final del software.

Seguridad (OWASP Mobile)

El sistema presenta un cumplimiento del 100% con los estándares del OWASP Mobile Top 10, lo que demuestra que se han aplicado buenas prácticas de seguridad desde las primeras etapas del desarrollo.

Entre las medidas implementadas destacan el uso de comunicaciones cifradas, validación estricta de datos de entrada y reglas de acceso bien definidas en la base de datos, reduciendo la probabilidad de ataques y accesos no autorizados.

Accesibilidad (WCAG 2.1)

El cumplimiento total del Nivel AA de las Pautas de Accesibilidad para el Contenido Web (WCAG) 2.1 garantiza que la aplicación pueda ser utilizada por un público amplio y diverso, incluyendo personas con distintos tipos de discapacidad. Este nivel de conformidad asegura que la interfaz y las funcionalidades del sistema han sido diseñadas considerando principios de accesibilidad, usabilidad e inclusión digital.

Entre los aspectos evaluados se encuentra el soporte para tecnologías de asistencia, como lectores de pantalla, lo que permite que los contenidos sean interpretados correctamente por usuarios con discapacidad visual. Asimismo, se verificó que la estructura de la interfaz facilite una navegación clara y coherente, permitiendo el uso de la aplicación mediante teclado u otros dispositivos de entrada alternativos.

Adicionalmente, se validó que los elementos interactivos cumplan con los tamaños mínimos recomendados, reduciendo errores de selección y facilitando la interacción para usuarios con limitaciones motrices. El uso adecuado de contrastes, jerarquía visual y etiquetas descriptivas contribuye a una mejor comprensión del contenido, beneficiando también a usuarios con dificultades cognitivas.

En conjunto, estas características no solo cumplen con los requisitos normativos establecidos por WCAG 2.1, sino que también mejoran la experiencia general de uso, promoviendo una aplicación más inclusiva, accesible y fácil de utilizar para todos los usuarios, independientemente de sus capacidades.

Rendimiento del Sistema

Los resultados de rendimiento reflejan una aplicación optimizada y eficiente:

- **Tiempo de inicio:** 2.1 segundos, por debajo del límite recomendado.
- **Uso de memoria:** 67 MB, lo que evita sobrecarga en dispositivos móviles.
- **Tamaño del APK:** 8.5 MB, favoreciendo una instalación rápida.
- **Concurrencia:** Soporte para hasta 50 usuarios simultáneos sin degradación significativa del rendimiento.

Estos valores indican que el sistema está preparado para operar en escenarios reales con múltiples usuarios.

Usabilidad (System Usability Scale – SUS)

La evaluación de la usabilidad del sistema se realizó mediante el System Usability Scale (SUS), una herramienta estandarizada ampliamente utilizada para medir la percepción de los usuarios respecto a la facilidad de uso de una aplicación. La puntuación obtenida fue de 82.5%, valor que se ubica dentro del rango considerado como usabilidad excelente, lo que indica una experiencia de usuario altamente satisfactoria.

Este resultado sugiere que los usuarios pueden interactuar con la aplicación de forma intuitiva y fluida, sin necesidad de capacitación previa ni instrucciones extensas. Las funcionalidades principales son fácilmente identificables, la navegación es clara y coherente, y las acciones del usuario generan respuestas predecibles por parte del sistema, reduciendo la posibilidad de errores durante el uso.

Asimismo, la puntuación refleja que la aplicación presenta una baja carga cognitiva, permitiendo que los usuarios completen sus tareas con un esfuerzo mínimo y en un tiempo razonable. La consistencia en el diseño de la interfaz, el uso adecuado de iconografía, etiquetas claras y una estructura visual ordenada contribuyen significativamente a esta percepción positiva.

Finalmente, el resultado obtenido evidencia un enfoque centrado en el usuario durante el proceso de diseño y desarrollo, priorizando la accesibilidad, la claridad y la eficiencia en la interacción. Esto no solo mejora la satisfacción del usuario final, sino que también incrementa la probabilidad de adopción y uso continuo de la aplicación en escenarios reales.

Reporte de defectos

Tabla de defectos

ID	SEVERIDAD	CATEGORIA	DESCRIPCION BREVE
01	Critico	Seguridad	Permisos Firestore – Lectura de reservas bloqueadas
02	Crítico	Seguridad	Colección History sin reglas de seguridad
04	Medio	Accesibilidad	Cálculo de contraste WCAG incorrecto
05	Medio	Configuración	Colores duplicados en resources
06	Menor	Código	Deprecation warnings en toLowercase()
07	Crítico	Lógica	Reservas en fechas pasadas permitidas
08	Medio	Lógica	Cancelación de reservas finalizadas
09	Menor	UX	Bloques horarios pasados en selector

Defectos críticos

#01: Permisos Firestore – Lectura de reservas bloqueada

Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Crítico (Bloqueante)	Urgente	Seguridad / Backend	Firebase Firestore	03/12/2025 - 18:45	03/12/2025 - 20:15	Resuelto y Verificado	1.5 horas

Descripción detallada

Al intentar crear una reserva, la aplicación mostraba un error de permisos al verificar la disponibilidad del espacio. El sistema no permitía leer las reservas existentes de otros usuarios, lo que impedía validar si había conflictos de horarios.

Pasos para Reproducir

1. Usuario autenticado intenta crear una reserva
2. Sistema llama a `verificarDisponibilidad(lugarId, fecha, hora)`
3. Firestore rechaza la consulta a la colección `reservas`
4. Se muestra error: "PERMISSION_DENIED: Missing or insufficient permissions"

Comportamiento Esperado

- El sistema debe permitir leer TODAS las reservas de un lugar para verificar disponibilidad
- Solo debe restringir la escritura/actualización a reservas propias

Comportamiento Actual

- Firestore bloqueaba la lectura de reservas de otros usuarios
- Imposible verificar conflictos de horarios
- Creación de reservas completamente bloqueada

Pruebas de Verificación

- Test: `FirestoreSecurityRulesTest.usuario_autenticado_puede_leer_todas_las_reservas()
- Test: `FirestoreSecurityRulesTest.usuario_solo_puede_modificar_sus_reservas()`
- Prueba manual: Crear reserva en horario ocupado → Rechazada correctamente

#02: Colección History sin reglas de seguridad

Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Crítico (Bloqueante)	Urgente	Seguridad / Backend	Firebase Firestore	03/12/2025 - 18:46	03/12/2025 - 20:16	Resuelto y Verificado	2 horas

Descripción Detallada

Después de corregir DEF-001, apareció un nuevo error al crear reservas. El código usa transacciones batch que escriben simultáneamente en 3 colecciones, pero la colección `history` no tenía reglas de seguridad definidas.

Pasos para Reproducir

1. Corregir 01
2. Usuario intenta crear una reserva
3. Sistema ejecuta `db.runBatch()` escribiendo en `reservas`, `history`, y `notifications`
4. Error: "PERMISSION_DENIED" en escritura a `history`

Comportamiento Esperado

- La colección `history` debe tener reglas que permitan crear eventos del historial
- Solo el usuario propietario debe poder leer su historial

Comportamiento Actual

- Firestore rechazaba toda escritura a la colección `history`
- La transacción batch completa fallaba
- Reservas no se creaban

Causa Raíz

El archivo `firestore.rules` no incluía ninguna regla para la colección `history`. Por defecto, Firestore deniega todo acceso.

Pruebas de Verificación

- Test: `FirestoreSecurityRulesTest.usuario_puede_crear_eventos_en_history()`
- Test: `FirestoreSecurityRulesTest.usuario_solo_puede_leer_su_propio_historial()`
- Prueba manual: Crear reserva → Evento de historial creado correctamente

06: Reservas en fechas pasadas permitidas

Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Crítico (Bloqueante)	Urgente	Lógica de Negocio	Reservas	02/12/2025 - 14:30	02/12/2025 - 17:30	Resuelto y Verificado	3.0 horas

Descripción Detallada

Los usuarios podían crear reservas para fechas y horarios que ya habían pasado, lo cual no tiene sentido lógico ni funcional. Por ejemplo, reservar un espacio para "2024-01-15 a las 10:00" cuando estamos en Diciembre 2025.

Pasos para Reproducir

1. Usuario navega a la pantalla de reservas
2. Selecciona una fecha pasada (ej: hace 1 mes)
3. Selecciona un horario (cualquiera)
4. Sistema permite crear la reserva sin validar

Comportamiento Esperado

El sistema debe rechazar reservas con fecha/hora pasada

Mostrar mensaje: "No puedes reservar en fechas u horarios pasados"

Deshabilitar fechas pasadas en el selector de calendario

Comportamiento Actual

Sistema permitía crear reservas en fechas pasadas

No había validación de fecha/hora futura

- Reservas inválidas se guardaban en la base de datos
- Archivos Modificados
- `FirestoreRepository.kt` - Agregar validación de fecha futura
- `DateTimeUtils.kt` - Nueva función `esFechaHoraFutura()`
- `ReservaFragment.kt` - Validación en UI antes de enviar

Pruebas de Verificación

1. Test: `DateTimeUtilsTest.esFechaHoraFutura_retorna_false_para_fecha_pasada()`
2. Test: `DateTimeUtilsTest.esFechaHoraFutura_retorna_true_para_fecha_futura()`
3. Prueba manual: Intentar reservar ayer → Rechazada
4. Prueba manual: Intentar reservar mañana → Permitida

Defectos Medios

 # 03: Cálculo de contraste WCAG incorrecto

Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Medio	**Alta**	**Accesibilidad**	**Tests de Accesibilidad**	**03/12/2025 - 10:15**	**03/12/2025 - 14:15**	**Resuelto y Verificado**	**4 horas**

Descripción Detallada

Los tests de accesibilidad fallaban porque la función de cálculo de contraste de color retornaba siempre `1.0` para todos los colores, lo que hacía que los tests no validaran correctamente el cumplimiento de WCAG 2.1 AA.

Pasos para Reproducir

1. Ejecutar `.\gradlew test --tests 'AccessibilityTest'`
2. Test `colores de texto cumplen con estandar WCAG AA` falla
3. Revisar output: todos los ratios de contraste son `1.0`

Comportamiento Esperado

- Negro sobre blanco (#000000) debe tener ratio ~21:1
- Gris medio (#666666) debe tener ratio ~5.74:1
- Los tests deben validar ratios de contraste reales

Comportamiento Actual

- Todos los colores retornaban ratio de `1.0`
- Tests fallaban o pasaban incorrectamente
- No se detectaban problemas reales de contraste

Causa Raíz

La función `calcularContraste()` tenía múltiples errores:

1. Parseo incorrecto de colores hexadecimales
2. Fórmula de luminancia relativa incorrecta
3. Fórmula de ratio de contraste incorrecta

Archivos Modificados

- `AccessibilityTest.kt` - Reescribir enfoque de validación
- `colors_accessible.xml` - Nuevo archivo con colores pre-validados

Pruebas de Verificación

- Test: `AccessibilityTest.colores_de_texto_cumplen_con_estandar_WCAG_AA()`
- Test: `AccessibilityTest.botones_tienen_contraste_minimo()`
- Validación externa: WebAIM Contrast Checker
- Auditoría: Android Accessibility Scanner

04: Colores duplicados en resources

Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Medio	Alta	Configuración	Android Resources	03/12/2025 - 14:20	03/12/2025 - 15:20	Resuelto y Verificado	1 horas

Descripción Detallada

Al crear el archivo `colors_accessible.xml`, se generaron conflictos de compilación porque algunos nombres de colores ya existían en `colors.xml`, provocando errores de recursos duplicados.

Pasos para Reproducir

1. Crear archivo `colors_accessible.xml` con colores accesibles
2. Usar nombres como `colorPrimary`, `colorAccent`, etc.
3. Ejecutar `.\gradlew build`
4. Error: "duplicate value for resource 'attr/colorPrimary'"

Comportamiento Esperado

- Cada recurso debe tener un nombre único
- No debe haber conflictos entre archivos de resources

Comportamiento Actual

- Compilación fallaba con error de recursos duplicados
- No se podía construir la APK

Solución Implementada

Renombrar todos los colores en `colors_accessible.xml` con el prefijo `wcag_`:

Archivos Modificados

- `colors_accessible.xml` - Renombrar todos los colores con prefijo `wcag_`

Pruebas de Verificación

- Build: `.\gradlew build` sin errores
- Test: Todos los tests de accesibilidad pasan
- Verificación manual: APK se genera correctamente

07: Cancelación de reservas finalizadas
Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Medio	Alta	Lógica de Negocio	Gestión de Reservas	02/12/2025 - 16:00	02/12/2025 - 18:30	Resuelto y Verificado	2.5 horas

Descripción Detallada

Los usuarios podían cancelar reservas que ya habían finalizado (fechas/horas pasadas), lo cual no tiene sentido lógico. Por ejemplo, cancelar una reserva del mes pasado.

Pasos para Reproducir

1. Usuario tiene una reserva de hace 1 semana (ya finalizada)
2. Navega a "Mis Reservas"
3. Selecciona la reserva finalizada
4. Sistema permite cancelar

Comportamiento Esperado

- Solo permitir cancelar reservas futuras
- Mostrar mensaje: "No puedes cancelar una reserva ya finalizada"
- Cambiar botón a "Ver Detalles" para reservas pasadas

Comportamiento Actual

- Sistema permitía cancelar cualquier reserva
- No había validación de fecha/hora
- Reservas finalizadas se marcaban como "canceladas" incorrectamente

Archivos Modificados

- `FirestoreRepository.kt` - Agregar validación de fecha futura
- `MisReservasFragment.kt` - Ocultar botón de cancelar para reservas finalizadas
- `DateTimeUtils.kt` - Usar función `esFechaHoraFutura()` existente

Pruebas de Verificación

- Test: `FirestoreRepositoryTest.no_se_puede_cancelar_reserva_finalizada()`
- Prueba manual: Intentar cancelar reserva de ayer → Rechazada
- Prueba manual: Cancelar reserva de mañana → Permitida

Defectos Menores

05: Deprecation warnings en toLowerCase()

Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Menor	Baja	Calidad de Código	Tests de Seguridad	03/12/2025 - 11:00	03/12/2025 - 11:30	Resuelto y Verificado	0.5 horas

Descripción Detallada

Durante la compilación de tests aparecían warnings de deprecación indicando que el método `String.toLowerCase()` está deprecado en Kotlin 1.5+.

Causa Raíz
Uso de API deprecada de Kotlin 1.4 en código más reciente.

Solución Implementada

Método actual de Kotlin

val emailLower = email.lowercase()

val dominio = email.substringAfter("@").lowercase()

Archivos Modificados

- `AndroidPermissionsSecurityTest.kt` - Líneas 275, 335
- `AuthenticationSecurityTest.kt` - Líneas 215, 348

08: Bloques horarios pasados en selector

Información General

Severidad	Prioridad	Categoria	Componente	Fecha Deteccion	Fecha de resolucion	Estado	Tiempo de Resolución
Menor	Media	UX / Usabilidad	Selector de Horarios	05/12/2025 - 09:30	05/12/2025 - 11:00	Resuelto y Verificado	1.5 horas

Descripción Detallada

Al reservar para el día actual, el selector de horarios mostraba todos los bloques del día (incluyendo horarios pasados), lo que confundía a los usuarios.

Pasos para Reproducir

1. Seleccionar fecha de HOY
2. Observar selector de horarios
3. Sistema muestra horarios de 08:00-10:00, 10:00-12:00, etc.
4. Si son las 15:00, los bloques de 08:00-14:00 ya pasaron

Comportamiento Esperado

- Solo mostrar bloques horarios futuros para el día actual
- Si son las 15:00, mostrar desde 16:00 en adelante

Comportamiento Actual

- Se mostraban todos los bloques del día
- Usuario podía seleccionar bloques pasados
- Error al intentar crear la reserva

Archivos Modificados

- `HorarioUtils.kt` - Agregar filtro de bloques pasados

Manual de Usuario

https://drive.google.com/file/d/1IbhvzmfjPKouGcy64cZGH3Acn6ZMaJ-D/view?usp=sharing

Manual Técnico

https://drive.google.com/file/d/1q9oRnkquiY4Rdl7L5AJnP6b5YrOBE8S_/view?usp=sharing

Retrospectiva del Equipo

El desarrollo de Spacius ha sido un reto académico y técnico intensivo de cuatro meses. Durante este periodo, el equipo no solo se enfocó en cumplir con los requisitos funcionales del sistema, sino que también tuvo que adoptar nuevas tecnologías y patrones de arquitectura modernos desde cero. A continuación, presentamos el análisis de nuestro proceso de trabajo.

Lo que funcionó (Aciertos)

Adopción de Arquitectura Robusta (MVVM): Aunque al principio representó una complejidad añadida, la decisión de separar la lógica de negocio (ViewModel) de la interfaz (View) fue crucial. Esto nos permitió trabajar de manera más ordenada y facilitó la detección de errores en etapas avanzadas del proyecto.

División de Roles y Especialización: La separación de tareas entre el desarrollo de la interfaz de usuario y la configuración de servicios en la nube (Firebase/Backend) permitió que cada miembro profundizara en su área, optimizando los tiempos de entrega.

Integración de Servicios "Serverless": Optar por Firebase como backend fue un acierto estratégico. Nos ahorró semanas de desarrollo de una API REST tradicional y gestión de servidores, permitiéndonos centrar nuestros esfuerzos en la experiencia de usuario y la funcionalidad de la app móvil.

Diseño Intuitivo: Logramos una interfaz limpia y fácil de usar, cumpliendo con el flujo esperado: el usuario puede completar una reserva en pocos pasos, lo cual era el objetivo principal.

Desafíos y Obstáculos (Lo que costó)

Curva de Aprendizaje de Kotlin: Al venir de una formación académica más centrada en lenguajes tradicionales, la transición a la sintaxis concisa y las funciones específicas de Kotlin (como las Coroutines o Scope Functions) requirió un esfuerzo de autoaprendizaje significativo durante el primer mes.

Gestión de APIs Externas: La configuración de Google Maps SDK y la gestión segura de las API Keys (restringiendo accesos y configurando el archivo local.properties) presentaron dificultades técnicas iniciales que consumieron tiempo de investigación.

Manejo de Asincronía: Entender el flujo de datos asíncrono (esperar a que Firebase responda para mostrar datos en el mapa o el perfil) generó algunos errores ("bugs") de visualización al inicio, que tuvimos que solucionar implementando LiveData y observadores correctamente.

Aprendizajes Clave

Este proyecto de 4 meses nos deja lecciones valiosas que van más allá de la nota académica:

Capacidad de Autoaprendizaje: Demostramos que somos capaces de dominar un stack tecnológico moderno (Kotlin, Android Jetpack, Firebase) en un tiempo récord y aplicarlo en un producto funcional.

Importancia de la Planificación: Aprendimos que dedicar tiempo a diseñar la base de datos y los flujos de pantalla antes de escribir código ahorra muchas horas de refactorización posterior.

Trabajo con Control de Versiones: La colaboración en el código nos obligó a mejorar nuestras prácticas con Git, resolviendo conflictos y manteniendo ramas separadas para evitar romper funcionalidades estables.

Desarrollo Orientado al Usuario: Comprendimos que una app no solo debe funcionar a nivel de código, sino que debe dar feedback al usuario (como las notificaciones o los mensajes de confirmación) para ser verdaderamente útil.

Conclusión

El proyecto Spacius ha concluido satisfactoriamente. A pesar de la curva de aprendizaje inicial y la limitación de tiempo, nuestro equipo logró entregar una aplicación estable, escalable y con todas las funcionalidades propuestas operativas. El resultado final

cumple con las expectativas planteadas al inicio del semestre y sienta una base sólida para futuros desarrollos profesionales.

Plan de Evolución Futura (Roadmap)

Aunque la versión actual (v1.0) de Spacius cumple con el flujo funcional completo para la reserva de espacios, se ha diseñado una hoja de ruta para escalar el proyecto hacia un producto comercial viable. A continuación, se detallan las fases de implementación futura:

Fase 1: Control de Acceso y Validación (Corto Plazo)

El objetivo inmediato es cerrar el ciclo de la reserva en el mundo físico.

Generación de Códigos QR: Al confirmar una reserva, la app generará un código QR único asociado al usuario y al horario.

App de "Guardia/Controlador": Implementar una funcionalidad de escaneo para que el personal de seguridad del parque pueda validar el QR del usuario al ingresar, asegurando que la reserva es válida y no ha sido duplicada.

Sistema de Calificaciones: Permitir que, al finalizar el horario reservado, el usuario pueda calificar el estado del espacio (limpieza, iluminación) mediante un sistema de 1 a 5 estrellas.

Fase 2: Pasarela de Pagos y Monetización (Mediano Plazo)

Actualmente, las reservas son gratuitas. Para la sostenibilidad del sistema, se integrarán cobros.

Integración de Pasarela de Pagos: Implementar Stripe o PayPhone SDK para permitir el pago directo de canchas privadas o tasas municipales desde la app.

Billetera Digital: Crear un saldo virtual dentro de la app para usuarios recurrentes, facilitando reservas rápidas sin necesidad de ingresar tarjetas cada vez.

Fase 3: Funciones Sociales e Inteligencia (Largo Plazo)

Convertir a Spacius en una plataforma social y predictiva.

Reservas Colaborativas: Opción de "Invitar Amigos". El usuario crea la reserva y comparte un enlace para que sus amigos confirmen asistencia dentro de la app.

Recomendaciones con IA: Utilizar el historial de reservas para sugerir nuevos parques cercanos o notificar disponibilidad en los horarios favoritos del usuario (ej. "Tu cancha favorita de los martes está libre, ¡resérvala ahora!").

Versión Multiplataforma: Migrar la lógica de negocio a Kotlin Multiplatform (KMP) o desarrollar la versión nativa para iOS, abarcando así el 100% del mercado móvil.