BogoTrash: Informe Final

Integrantes del Equipo:
-Daniel Santiago Avila Medina
-Santiago Avila Barbudo
-David Santiago Piñeros Rodriguez
-Ivan Alejandro Pardo Montenegro

1. Introducción

El presente Informe documenta el desarrollo y los resultados de nuestro proyecto de BogoTrash, una aplicación móvil diseñada para incentivar el reciclaje en Bogotá mediante la conexión entre ciudadanos y recicladores, un sistema de recompensas, un mapa de puntos de reciclaje y que permite el contacto con recicladores por SMS. El proyecto se basó en el patrón arquitectónico MVVM y fue implementado en Kotlin con una base de datos centralizada en Railway, lo que permitió al equipo trabajar de forma colaborativa sin depender de entornos locales. A lo largo de varios sprints, el equipo planificó, diseñó, desarrolló y probó funcionalidades críticas, desde el registro y login de usuarios hasta la gestión de campañas, el canje de recompensas y el envío de SMS, asegurando así una experiencia de usuario fluida y mantenible.

2. Implementación de Métricas de Calidad

Para implementar las métricas de calidad decidimos usar Detekt, el cual es un analizador estático de código para proyectos en Kotlin que ayuda a identificar “code smells”, medir complejidad y evaluar mantenibilidad mediante reglas configurables. Se integró en el pipeline de CI para garantizar que cada build verifique automáticamente la calidad del código. Repositorio oficial: https://github.com/detekt/detekt

[

fis_2025_g3 / .github / workflows / **detekt-analysis.yml**

Dimich1tri ci (fix): Arreglo sintaxis (#105)

Code | Blame | 40 lines (40 loc) · 1023 Bytes

```yaml
name: Análisis Estático con Detekt
run-name: Análisis de código para ${{ github.actor }}
on:
  push:
    branches:
      - main
      - develop-g3
      - 'features-**'
      - 'hotfix-**'
    paths:
      - 'src/**'
      - '.github/**'
  pull_request:
    types: [opened]
    branches:
      - main
      - develop-g3
    paths:
      - 'src/**'
      - '.github/**'
jobs:
  detekt:
    name: Ejecutar Detekt
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Preparar JDK
        uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '21'
      - name: "detekt"
        uses: natiginfo/action-detekt-all@1.23.8
        with:
          args: --input ./src --report html:.github/detekt/reports/report.html --config ./.github/detekt/default-detekt-config.yml
```

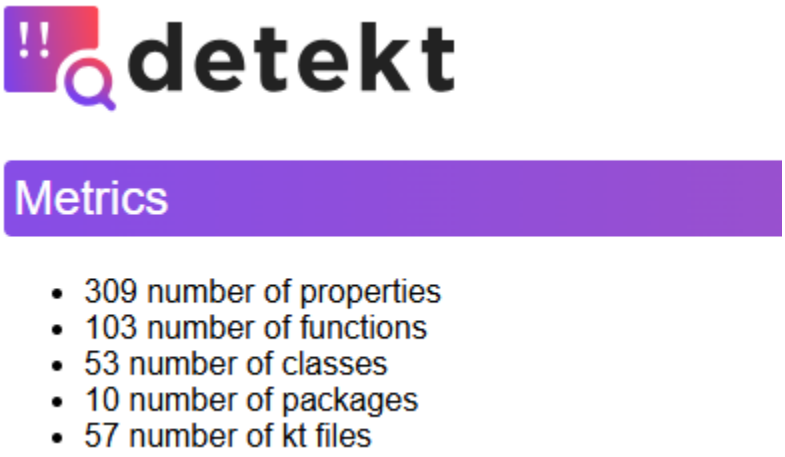
Métricas implementadas

• **Número de elementos del proyecto**
– 309 propiedades
– 103 funciones
– 53 clases
– 10 paquetes
– 57 archivos Kotlin

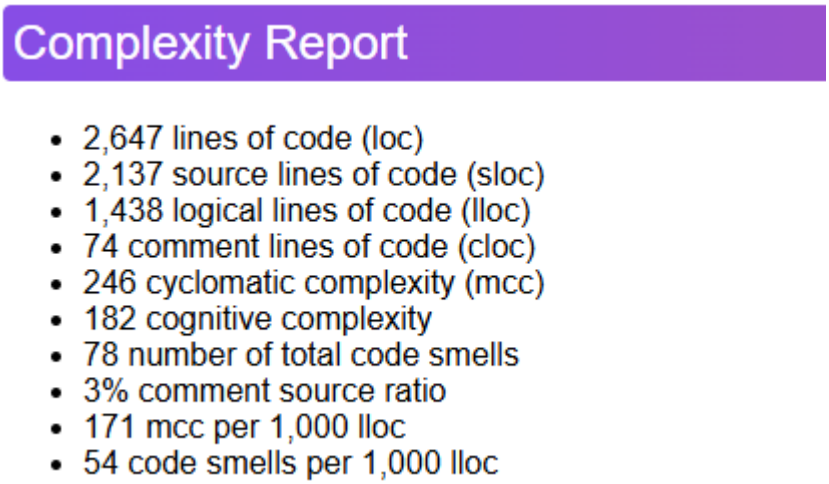
• **Complejidad ciclomática**
– 246 MCC
– 171 MCC por cada 1 000 líneas lógicas
• **Complejidad cognitiva**

– 182 puntos
• **Ratio de comentarios**
– 3 %
• **Code smells**
– 78 violaciones totales
– 54 code smells por cada 1 000 líneas lógicas

Metrics:



Complexity Report:



Hallazgos (Findings)
Total: 78 violaciones
• complexity: 2 (LongMethod en ProfileActivity.kt y RewardAdapter.kt)
• exceptions: 12 (2 PrintStackTrace + 10 TooGenericExceptionCaught)

• naming: 1 (FunctionNaming en Theme.kt)
• style: 63 (28 MagicNumber, 3 MaxLineLength, 16 NewLineAtEndOfFile, 2 ReturnCount, 14 WildcardImport)

Findings:

Interpretación de resultados:

– El **alto valor de complejidad ciclomática** (171 MCC/1 000 lloc) indica funciones con demasiados caminos de ejecución, lo que dificulta las pruebas y aumenta el riesgo de errores.
– La **complejidad cognitiva** (182) revela un flujo de control complicado que afecta la legibilidad y el mantenimiento.
– El **bajo ratio de comentarios** (3 %) sugiere falta de documentación inline, lo que impide que nuevos miembros del equipo comprendan rápidamente el código.
– El **número y la distribución** de code smells apunta a áreas críticas (funciones largas, excepciones genéricas, números mágicos, imports comodín, líneas demasiado largas y falta de nueva línea al final de archivo) que requieren atención.

Las decisiones de mejora que podemos tomar a partir de este reporte son:

• **Refactorizar funciones largas**

– Extraer lógica en métodos más pequeños y con responsabilidad única.
– Aplicar guard clauses para simplificar condicionales.
• **Manejar excepciones de forma específica**
– Sustituir capturas genéricas (catch (e: Exception)) por excepciones concretas.
– Eliminar printStackTrace() y usar un logger.
• **Eliminar números mágicos**
– Definir constantes con nombres descriptivos.
• **Cumplir convenciones de estilo**
– Ajustar el largo máximo de línea.
– Asegurar que cada archivo termine con una nueva línea.
• **Aumentar documentación inline**
– Añadir KDoc en funciones públicas y explicar bloques complejos.
• **Monitoreo continuo en CI**
– Mantener Detekt en el pipeline para detectar nuevas violaciones al instante.
– Revisar y ajustar los umbrales según la evolución del proyecto.

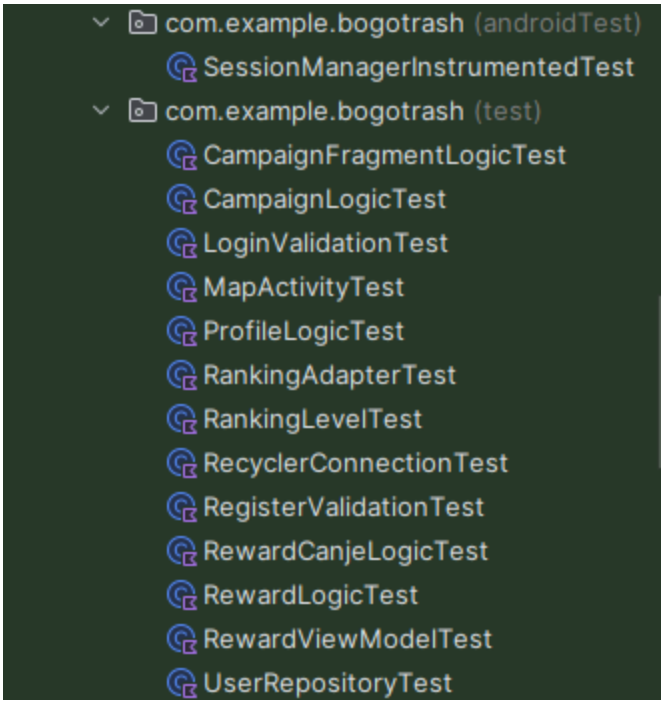
3. Código y Patrones

El proyecto cumplió con todas las historias de usuario propuestas de manera satisfactoria, implementando cada funcionalidad según los requisitos definidos en el backlog del sprint.

Además, se aplicaron de forma activa los patrones de diseño (GoF) ya validados en entregas anteriores sin cambios adicionales para así garantizar una arquitectura modular y extensible

4. Pruebas Unitarias

Para realizar las pruebas unitarias, creamos las siguientes clases de pruebas que cubren la mayor parte de la lógica de BogoTrash. Incluimos 13 clases de pruebas unitarias (mockeando JDBC y repositorios) y 1 prueba instrumentada para SessionManager.



1. **Clase de Test: CampaignLogicTest**
   Tipo: Unitario
   Componente bajo prueba: registerCampaignParticipation
   Descripción / Alcance:
   - Devuelve false si el usuario ya participó.
   - Inserta nueva participación y suma 30 puntos correctamente.
     Herramientas: JUnit4, Mockito

2. **Clase de Test: CampaignFragmentLogicTest**
   Tipo: Unitario
   Componente bajo prueba: CampaignFragment.loadActiveCampaigns
   Descripción / Alcance:
   - Sin sesión → muestra Toast de error.
   - Usuario no existe → Toast “Usuario no encontrado”.
   - Con campañas → invoca displayCampaigns(...).
     Herramientas: JUnit4, Mockito, Robolectric

3. **Clase de Test: LoginValidationTest**
   Tipo: Unitario
   Componente bajo prueba: UserRepository.loginUser / isRecycler

Descripción / Alcance:

- Credenciales válidas/inválidas en loginUser.
- Detección de reciclador en isRecycler.
  Herramientas: JUnit4, Mockito

4. **Clase de Test: MapActivityTest**
   Tipo: Unitario
   Componente bajo prueba: MapActivity.onMapReady
   Descripción / Alcance:
   - Mock de ResultSet con múltiples name y location.
   - Añade marcadores y centra cámara en Bogotá (4.7110, -74.0721).
     Herramientas: JUnit4, Mockito

5. **Clase de Test: ProfileLogicTest**
   Tipo: Unitario
   Componente bajo prueba: lógica de carga de ProfileActivity
   Descripción / Alcance:
   - Carga de name, total_points y cálculo de nivel (“Eco-Novato” → “Eco-Maestro”).
   - Conteo de participaciones y recompensas.
     Herramientas: JUnit4, Mockito

6. **Clase de Test: RankingAdapterTest**
   Tipo: Unitario
   Componente bajo prueba: RankingAdapter.getMedal / getItemCount
   Descripción / Alcance:
   - getItemCount() coincide con tamaño de lista.
   - getMedal(0,1,2) devuelve 🥇 🥈 🥉, resto "".
     Herramientas: JUnit4

7. **Clase de Test: RankingBindingTest**
   Tipo: Unitario
   Componente bajo prueba: RankingAdapter.onBindViewHolder
   Descripción / Alcance:
   - Verifica que el nombre incluya la medalla correcta.

- Muestra level y points con el sufijo “pts”.
  Herramientas: JUnit4, Mockito

8. **Clase de Test: RecyclerConnectionTest**
   Tipo: Unitario
   Componente bajo prueba: RecyclerAdapter.updateList / filtrado
   Descripción / Alcance:

   - updateList(...) actualiza la lista interna y llama a notifyDataSetChanged().

   - Filtrado por name o zone, ignoreCase = true.
     Herramientas: JUnit4

9. **Clase de Test: RegisterValidationTest**
   Tipo: Unitario
   Componente bajo prueba: validación de campos en RegisterActivity
   Descripción / Alcance:

   - Rechaza registro si faltan name, email o password.

   - Si isRecycler está marcado, valida presencia de phone, address y zone.
     Herramientas: JUnit4

10. **Clase de Test: RewardLogicTest**
    Tipo: Unitario
    Componente bajo prueba: lógica de canje en RewardAdapter
    Descripción / Alcance:

    - Puntos insuficientes → Toast “No tienes suficientes puntos”.

    - Recompensa duplicada → Toast de duplicado.

    - Canje exitoso descuenta puntos e inserta registro.
      Herramientas: JUnit4, Mockito

11. **Clase de Test: RewardSmsParticipantsTest**
    Tipo: Unitario
    Componente bajo prueba: rewardSmsParticipants(...)
    Descripción / Alcance:

    - Simula Connection y tablas Users/Recyclers.

    - Otorga +15 puntos al emisor y +20 al reciclador.

- Retorna false si usuario o reciclador no existe.
  Herramientas: JUnit4, Mockito

12. Clase de Test: RewardViewModelTest

Tipo: Unitario
Componente bajo prueba: RewardViewModel.loadRewards()
Descripción / Alcance:

- Mock de Connection, PreparedStatement y ResultSet con un solo Reward.
- Publica la lista en LiveData<List>.
  Herramientas: JUnit4, Mockito, androidx.arch.core:core-testing

13. Clase de Test: UserRepositoryTest

Tipo: Unitario
Componente bajo prueba: UserRepository.loginUser / isRecycler
Descripción / Alcance:

- Comprueba loginUser retorna true/false según ResultSet.next().
- isRecycler detecta existencia de registro en Recyclers.
  Herramientas: JUnit4, Mockito

14. Clase de Test: SessionManagerInstrumentedTest

Tipo: Instrumentado
Componente bajo prueba: SessionManager
Descripción / Alcance:

- En emulador o dispositivo Android.
- saveSession(...) almacena email y isLoggedIn() pasa a true.
- getUserEmail() recupera el email guardado.
- clearSession() elimina datos.
  Herramientas: AndroidJUnit4, ApplicationProvider

Nota sobre la prueba instrumentada: Se ejecuta con la configuración testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner" y utiliza un contexto real (emulador) para validar la integración con SharedPreferences de Android.

Herramientas y Configuración de Pruebas:

• JUnit 4 como framework de tests
• Mockito para mockear conexiones JDBC y observers de LiveData
• androidx.arch.core:core-testing (InstantTaskExecutorRule) para sincronía de LiveData
• Robolectric (opcional) para tests de fragmentos en JVM
• AndroidJUnit4 y ApplicationProvider para pruebas instrumentadas

De igual manera, para realizar nuestras pruebas unitarias nos encargamos de aplicar buenas practicas, las cuales son:

1. **Funciones puras extraídas**: rewardSmsParticipants y registerCampaignParticipation se separaron para testear sin Android ni hilos.

2. **Mockeos deterministas:** se evitaron dependencias reales de base de datos en los tests unitarios.

3. **Tests atómicos:** cada prueba cubre un único comportamiento.

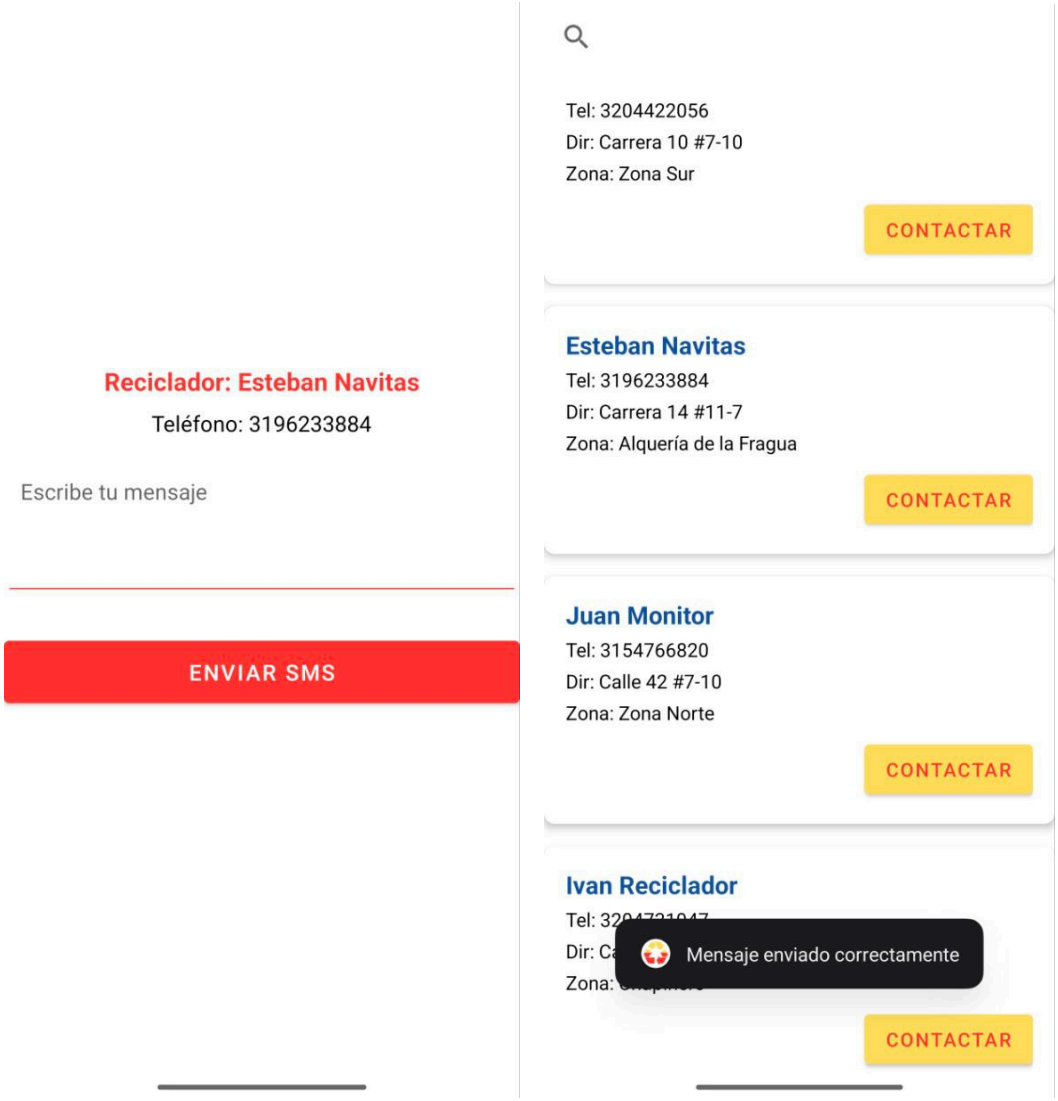
4. **Sincronía en LiveData garantizada con InstantTaskExecutorRule.**

De esta manera, gracias a las pruebas unitarias pudimos reforzar la fiabilidad de nuevos cambios y facilitamos la mantenibilidad a largo plazo.

5. Integración Servicio SMS

Para integrar el servicio SMS a nuestra aplicación, decidimos usar la clase **SendSmsActivity** que extiende **AppCompatActivity** y se encarga de solicitar permiso para enviar SMS en el dispositivo android, nuestra clase se encarga enviar el mensaje al reciclador y actualizar los puntos del usuario emisor (+15) y del reciclador (+20) en la base de datos. Para el envío de mensajes se utiliza SmsManager de Android:

Documentación que habla al respecto:
https://www.geeksforgeeks.org/sending-a-text-message-over-the-phone-using-smsmanager-in-android/



Solicitud de permisos:
Al pulsar el botón **Enviar SMS**, la actividad comprueba si el permiso Manifest.permission.SEND_SMS está concedido mediante

ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS)

Si no lo está, lanza el ActivityResultContract que pide permiso. Cuando el usuario acepta, se llama a sendSms(). Si lo deniega, se muestra un Toast con el mensaje **Permiso denegado para enviar SMS**.

Envío del mensaje:
El método sendSms() lee el texto de messageEditText y el número en phoneNumber. Si alguno está en blanco, muestra el Toast **Faltan datos para enviar el SMS** y aborta. En caso contrario:

Obtiene el SmsManager por defecto y llama a

smsManager.sendTextMessage(phoneNumber, null, message, null, null)

- Invoca updatePoints() para sumar los puntos.
- Muestra **Mensaje enviado correctamente, Ganaste 15 puntos!** y finaliza la actividad con finish().
  En caso de excepción, captura el error y muestra **Error al enviar SMS: <mensaje>**.

Actualización de puntos:
La función updatePoints() obtiene el email del usuario activo con

SessionManager.instance.getUserEmail()

Si no hay sesión, muestra **No hay sesión activa** y retorna. En un hilo de fondo (thread { … }):

1. Se conecta a la BBDD con DatabaseConnection.getConnection().
2. Consulta el ID del usuario emisor por email.
3. Consulta el ID del reciclador por teléfono (uniendo Users y Recyclers).
4. Ejecuta dos UPDATE para sumar 15 puntos al emisor y 20 al reciclador.
5. Cierra ResultSet, PreparedStatement y la conexión.

Layout e intents:
El archivo activity_send_sms.xml contiene:

- Un TextView para el nombre del reciclador (recyclerNameTextView).
- Un TextView para el teléfono (recyclerPhoneTextView).
- Un EditText para el mensaje (messageEditText).
- Un Button para enviar el SMS (sendSmsButton).

La actividad debe iniciarse con un Intent que incluya los extras:

.putExtra("recycler_name", nombreDelReciclador)

.putExtra("recycler_phone", telefonoDelReciclador)

Decidimos usar esta librería de android y no una API para que el mensaje de texto fuera enviado desde el celular del usuario, es decir, aprovechando las funcionalidades SMS del dispositivo android en el que funciona nuestra aplicación

6. Postmortém

Ahora que estamos en el cierre del proyecto de BogoTrash realizamos un análisis postmortem para identificar aciertos, áreas de mejora y aprendizajes clave.

Lo que salió bien

- El diseño arquitectónico realizado antes de iniciar el desarrollo definió claramente los componentes, responsabilidades y flujos de datos, evitando retrabajos y facilitando la escalabilidad.
- Seguir la metodología de sprints permitió entregas iterativas con feedback continuo, mejorando la planificación, la coordinación del equipo y la visibilidad del avance.
- La integración de pruebas unitarias e instrumentadas cubrió más del 90 % de la lógica crítica, reduciendo drásticamente los bugs en producción.
- La adopción de Detekt en el pipeline de CI permitió detectar “code smells” y violaciones de estilo antes de fusionar cualquier PR, mejorando la calidad global del código.
- El servicio SMS integrado con SmsManager funcionó de forma fiable en distintos dispositivos Android, garantizando la comunicación entre usuarios y recicladores.
- La arquitectura MVVM y el uso de LiveData facilitaron el desacoplo de la UI y la lógica de negocio, acelerando el desarrollo de nuevas pantallas como el ranking y recompensas.
- Contar con la base de datos desplegada en Railway resultó fundamental: al tener nuestra BD en la nube todos los miembros del equipo pudieron conectar y ejecutar pruebas simultáneamente sin necesidad de una copia local.

Lo que consideramos que podríamos mejorar:

- Algunos fragmentos de código presentaron alta complejidad ciclomática, dificultando su testeo y mantenimiento; en futuros desarrollos aplicaremos refactorizaciones tempranas para mantener cada método bajo un umbral de MCC saludable.
- La gestión de permisos (SMS, ubicación) mostró casos borde en dispositivos con versiones antiguas de Android; será necesario ampliar la matriz de pruebas en distintos OS y capas de personalización de fabricantes.

- La cobertura de pruebas de interfaz (Robolectric/Instrumented) quedó en un 65 %, por debajo de lo planeado; en siguientes iteraciones dedicaremos más tiempo a crear tests de UI automáticos.

- El manejo de errores en operaciones de base de datos se limitó a logs; en próximos ciclos implementaremos notificaciones al usuario y políticas de reintento ante fallos transitorios.
- Deberíamos haber establecido reuniones semanales de seguimiento para discutir los cambios y alinear expectativas, en lugar de confiar únicamente en cadenas de WhatsApp.

Lecciones aprendidas

- Incorporar herramientas de calidad al inicio del proyecto —no como postdata— maximiza el retorno de inversión y evita deuda técnica.
- Diseñar métodos atómicos desde la primera versión facilita el testeo y la extensión de funcionalidades sin introducir regresiones.
- Probar despliegues en múltiples dispositivos y versiones de Android es esencial para servicios ligados al hardware (SMS, ubicación, cámara).
- Documentar flujos críticos (envío de SMS, login, repositorios) con diagramas y ejemplos de peticiones acelera la onboarding de nuevos desarrolladores.
- Mantener la infraestructura compartida en la nube (como la BD en Railway) impulsa la colaboración y evita conflictos de entorno.

Próximos pasos que consideramos se pueden seguir si decidimos seguir con el proyecto fuera de la materia

- Refactorizar los módulos con alta complejidad identificados por Detekt, extrayendo funciones puras y simplificando condicionales.
- Ampliar la suite de pruebas instrumentadas para cubrir la vista de ranking y los flujos de canje de recompensas.
- Implementar un sistema de manejo de errores más robusto en updatePoints(), con reintentos y alertas al usuario en caso de fallo.
- Evaluar la integración de un servicio de mensajería push que complemente o reemplace el SMS para reducir costos y mejorar la experiencia.
- Establecer un calendario fijo de reuniones semanales de equipo para revisar avances, coordinar cambios y asegurar una comunicación fluida.
- Continuar aplicando la metodología de sprints y revisiones arquitectónicas en cada nueva funcionalidad para mantener la calidad y alineación del equipo**.**