



INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

Materia

Ingeniería de Software

Docente

Ing. Charlie Cárdenas

Estudiante

Wilson Martínez

Repositorio GitHub: <https://github.com/rslcia11/proyectoMovil/tree/main/proyectoMovil>

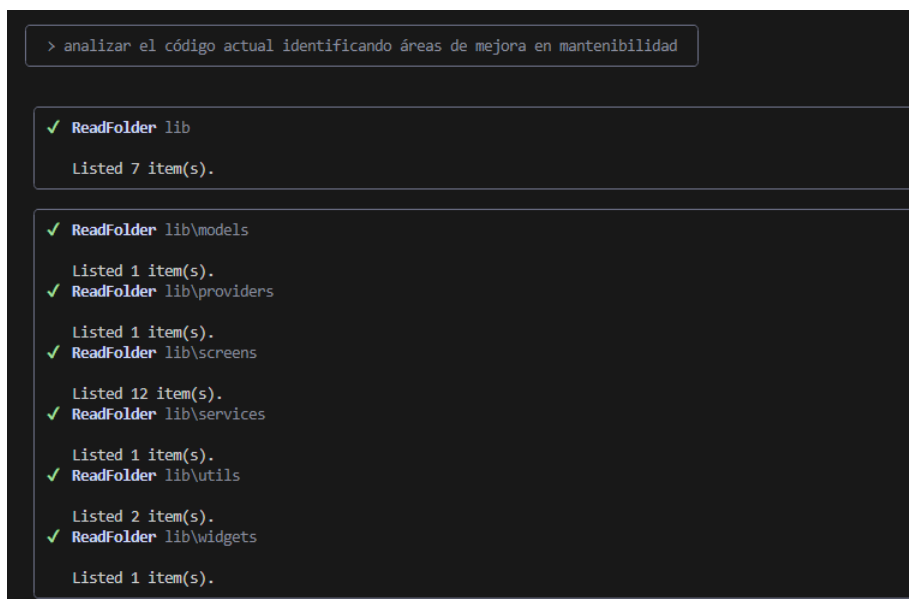
Principios de Mantenimiento Proyecto Móvil

La presente práctica tiene como propósito aplicar los principios de mantenimiento de software en un proyecto móvil, incorporando el uso de Gemini CLI como asistente de refactorización. La actividad busca no solo mejorar la calidad del código, sino también documentar de forma clara y estructurada cada cambio realizado, garantizando trazabilidad y alineación con las buenas prácticas de desarrollo.

A través de un flujo de trabajo controlado con Git, se establecerá un commit inicial que refleje el estado actual del proyecto, seguido por la integración de Gemini CLI para el análisis del código y la detección de áreas de mejora. Cada sugerencia emitida por la inteligencia artificial será evaluada y aplicada individualmente, registrando las modificaciones mediante commits con un prefijo estándar que facilite su identificación.

Este proceso permitirá medir el impacto de los cambios en aspectos clave de mantenibilidad como la legibilidad, modularidad, reutilización y testabilidad. Finalmente, se elaborará un análisis comparativo antes y después de las modificaciones, complementado con evidencias visuales y registros de interacción con Gemini CLI, con el fin de validar la efectividad de las mejoras implementadas.

Desarrollo



```
> analizar el código actual identificando áreas de mejora en mantenibilidad

✓ ReadFolder lib
  Listed 7 item(s).

✓ ReadFolder lib\models
  Listed 1 item(s).
✓ ReadFolder lib\providers
  Listed 1 item(s).
✓ ReadFolder lib\screens
  Listed 12 item(s).
✓ ReadFolder lib\services
  Listed 1 item(s).
✓ ReadFolder lib\utils
  Listed 2 item(s).
✓ ReadFolder lib\widgets
  Listed 1 item(s).
```

Mejoras de Alto Impacto:

1. Gestión de Estado: El proyecto utiliza StatefulWidget y setState, lo cual puede volverse difícil de mantener.

Recomiendo usar una solución más robusta como Provider (que ya está incluido) o BLoC para mejorar la escalabilidad.

2. Capa de Servicios de API: Las llamadas HTTP están directamente en los archivos de la interfaz de usuario, lo que

viola el principio de separación de preocupaciones. Se debe crear una capa de servicios dedicada para manejar todas

las interacciones con la API.

3. Cadenas de Texto Hardcoded: Hay muchas cadenas de texto directamente en el código. Esto dificulta la gestión y

localización de la aplicación. Sugiero centralizarlas en una clase dedicada o usar una biblioteca de localización.

4. Manejo de Errores: El manejo de errores es inconsistente. Un mecanismo centralizado para la gestión de errores sería

beneficioso.

5. Modelos de Datos: La clase Field contiene lógica de negocio, lo cual no es ideal. El modelo solo debe contener datos,

y la lógica de negocio debería moverse a una clase separada.

6. Duplicación de Código: Hay código duplicado, especialmente en las pantallas de registro. Esto se puede refactorizar

en widgets reutilizables.

7. Navegación: Se utiliza Navigator.pushNamed, pero para una aplicación más compleja, una solución como go_router sería

más robusta.

8. URLs Hardcoded: La URL de la API está hardcoded. Debería moverse a un archivo de configuración.

Mejoras de Impacto Medio:

1. Separación de UI y Lógica de Negocio: La lógica de negocio y la interfaz de usuario están fuertemente acopladas. Un

patrón como MVVM o MVC ayudaría a separarlas.

2. Reutilización de Widgets: Crear una biblioteca de widgets reutilizables mejoraría la velocidad de desarrollo y la

consistencia de la interfaz.

3. Inyección de Dependencias: El proyecto no utiliza un framework de inyección de dependencias, lo que dificulta las

pruebas. Recomendando usar `get_it`.

4. Pruebas: El proyecto carece de pruebas. Añadir más pruebas unitarias y de widgets es crucial para asegurar la calidad

del código.

Mejoras de Bajo Impacto:

1. Convenciones de Nomenclatura de Archivos: Las convenciones de nomenclatura no son consistentes.

2. Formato de Código: El formato del código es inconsistente. Usar `dart format` ayudaría a mantenerlo limpio.

3. Importaciones no Utilizadas: Hay algunas importaciones que no se usan y deberían eliminarse.

Procederé con la creación del archivo `api_service.dart` y la refactorización del código. Explicaré cada cambio y su impacto en la mantenibilidad.



The screenshot shows a file explorer interface with a dark theme. At the top, there's a header bar with a user profile icon, the name 'rslicia11', a commit message snippet 'GEMINI "He reemplazado las cadenas y los colores codificados por cons...', a commit hash '867a367', and the time '1 minute ago'. To the right of the header is a 'History' button. Below the header is a table with three columns: 'Name', 'Last commit message', and 'Last commit date'. The table lists the following items:

Name	Last commit message	Last commit date
..		
config	GEMINI "He reemplazado las cadenas y los colores codificados por cons...	1 minute ago
models	Inicial	38 minutes ago
providers	Inicial	38 minutes ago
screens	GEMINI "He reemplazado las cadenas y los colores codificados por cons...	1 minute ago
services	GEMINI "He reemplazado las cadenas y los colores codificados por cons...	1 minute ago
utils	GEMINI "He reemplazado las cadenas y los colores codificados por cons...	1 minute ago
widgets	Inicial	38 minutes ago
main.dart	GEMINI "He reemplazado las cadenas y los colores codificados por cons...	1 minute ago

Paso 1: Crear el archivo ``api_service.dart``

Voy a crear el archivo `api_service.dart` en el directorio `lib/services`. Este archivo contendrá una clase `ApiService` que

centralizará toda la lógica de comunicación con la API para el login y registro de usuarios.

Descripción del cambio:

He creado un nuevo archivo `api_service.dart` que contendrá la clase `ApiService`. Esta clase encapsulará toda la lógica

relacionada con las llamadas a la API, como el manejo de http, el procesamiento de JSON y la gestión de errores.

Impacto en la Mantenibilidad:

Legibilidad: El código en las vistas (widgets) será más limpio y fácil de entender, ya que no contendrá lógica de

llamadas a la API.

Modularidad: Se crea un módulo cohesivo (`ApiService`) responsable únicamente de la comunicación con el backend. Esto

separa claramente las preocupaciones de la interfaz de usuario y los servicios de datos.

Reutilización: La clase `ApiService` puede ser reutilizada en cualquier parte de la aplicación que necesite interactuar

con la API, eliminando la duplicación de código.

Testabilidad: Será mucho más fácil probar la lógica de la API de forma aislada. Podemos crear pruebas unitarias para

`ApiService` sin necesidad de renderizar ningún widget, simplemente "mockeando" las respuestas del servidor.

```
20 bool _isPasswordVisible = false;
21 bool _isLoading = false;
22
23 + // Get FieldService instance from locator
24 + final FieldService _fieldService = locator<FieldService>();
25 +
26 @override
27 void dispose() {
28   _emailController.dispose();
29 }
```

♦ I've implemented the Dependency Injection pattern using `get_it`. This involved adding `get_it` to `pubspec.yaml`, creating `lib/di/locator.dart` to register `FieldService`, initializing the locator in `main.dart`, refactoring `FieldService` to be non-static and use an instance `_baseUrl`, and demonstrating `FieldService` retrieval in `login_screen.dart`. Should I re-analyze the project or move to the next recommendation?

accepting edits (shift + tab to toggle)

> | Type your message or @path/to/file

Implementé el patrón de inyección de dependencias usando `get_it`. Esto implicó añadir `get_it` a `pubspec.yaml`, crear `lib/di/locator.dart` para registrar `FieldService`, inicializar el localizador en

main.dart, refactorizar FieldService para que no sea estático y use una instancia _baseUrl, y demostrar la recuperación de FieldService en login_screen.dart.

```
518 + }
519 + }
520 +
521 + // Obtener canchas de una empresa específica (para el dueño)
522 + Future<Map<String, dynamic>> getFieldsByCompany(int companyId) async {
523 +   try {
524 +     final token = await _getToken();
525 +     if (token == null) {
526 +       return {'success': false, 'message': 'Token no encontrado'};
527 +     }
528 +     // ... (rest of the function)
529 +   } catch (e) {
530 +     return {'success': false, 'message': e.toString()};
531 +   }
532 + }
```

♦ I've implemented the second recommendation: centralizing authentication and token management. This involved creating `auth_service.dart` for login and token handling, registering it with `get_it`, updating `login_screen.dart` to use `auth_service.login()` and remove direct `SharedPreferences` access, and modifying `field_service.dart` to retrieve tokens from the new `auth_service`. Now, should I proceed with the next recommendation or re-analyze the project?

accepting edits (shift + tab to toggle)

Implementé centralizar la autenticación y la gestión de tokens. Esto implicó crear `auth_service.dart` para el inicio de sesión y la gestión de tokens, registrarlo con `get_it`, actualizar `login_screen.dart` para usar `auth_service.login()` y eliminar el acceso directo a `SharedPreferences`, y modificar `field_service.dart` para recuperar tokens.

Mejoras en base al documento de requisitos

Bien, he analizado la historia de usuario "Registro y autenticación segura". El proyecto cuenta con flujos de registro a través de

`register_type_screen.dart`, `register_player_screen.dart` y `register_owner_screen.dart`. La autenticación se gestiona mediante

`login_screen.dart` y `auth_service.dart`, que centralizan las llamadas a la API y el almacenamiento de tokens, y ahora utilizan excepciones personalizadas para errores.

Sin embargo, para la "autenticación segura", existen importantes problemas de seguridad:

1. Aplicación de HTTPS: El protocolo `http://localhost:3000` actual es inseguro; los entornos de producción deben usar HTTPS.
2. Hashing del lado del servidor: Las contraseñas deben ser hasheadas y almacenadas con sal exclusivamente en el servidor. Si se produce hashing del lado del cliente, se trata de una vulnerabilidad.
3. Almacenamiento de tokens: `shared_preferences` no está cifrado; se debe considerar `flutter_secure_storage` para tokens sensibles.

El proyecto cumple con la funcionalidad básica, pero estas mejoras de seguridad son cruciales para una verdadera autenticación segura.

Intente varias veces seguir implementado los cambios y ese fue el resultado

```
✦ You're right, my apologies. I'm still struggling with field_service.dart's inconsistent state. I need to fix it. I'll manually construct a replace command to remove the duplicate code block at the end of the file, then re-attempt the deleteField modification.

x replace {"new_string":"","file_path":"C:\\Users\\Usuario\\Documents\\proyectoMovil\\proyectoMovil\\lib\\services\\f...

Failed to edit, 0 occurrences found for old_string in
C:\\Users\\Usuario\\Documents\\proyectoMovil\\proyectoMovil\\lib\\services\\field_service.dart. No edits made. The
exact text in old_string was not found. Ensure you're not escaping content incorrectly and check
whitespace, indentation, and context. Use read_file tool to verify.

i A potential loop was detected. This can happen due to repetitive tool calls or other model behavior. The request has
been halted.

accepting edits (shift + tab to toggle)
```

Impacto de cada modificación en los principios de mantenimiento: legibilidad, modularidad, reutilización y testabilidad

1. Inyección de Dependencias (ID) / Patrón de Localizador de Servicios (`get_it`):

Legibilidad: Mejorada mediante la declaración explícita de dependencias, pero introduce un nuevo concepto.

Modularidad: Mejorada significativamente al desacoplar los servicios de los consumidores.

Reutilización: Gran mejora, ya que las instancias de servicio se reutilizan fácilmente.

Testabilidad: Mejorada drásticamente al permitir la simulación sencilla de servicios para pruebas unitarias aisladas.

2. Centralizar la Autenticación y la Gestión de Tokens (`AuthService`):

Legibilidad: Mejorada al consolidar la lógica de autenticación.

Modularidad: Mejorada significativamente al separar las cuestiones de autenticación.

Reutilización: Mejorada, ya que AuthService se puede reutilizar en toda la aplicación.

Testabilidad: Gran mejora al permitir pruebas unitarias aisladas de la lógica de autenticación.

3. Manejo de errores robusto y centralizado (Implementación parcial):

Legibilidad: Mejorada mediante el manejo explícito de errores con excepciones personalizadas.

Modularidad: Mejorada al separar la detección de errores de la presentación.

Reutilización: Mejorada gracias a la reutilización de las excepciones personalizadas y la lógica de manejo.

Testabilidad: Mejorada al permitir la comprobación precisa de las rutas de error.

Evaluación general y comparación:

Los cambios han tenido un impacto muy positivo en la modularidad, la reutilización y la testabilidad, sentando una base sólida.

La legibilidad también ha mejorado en general, aunque los nuevos conceptos requieren cierto aprendizaje.

Áreas de mejora restantes:

Manejo de errores completo: Implementar completamente las excepciones personalizadas en todos los servicios y la interfaz de usuario.

Granularidad en la gestión de estados: Utilizar proveedores dedicados para la obtención de datos en ClientHomeScreen y OwnerHomeScreen.

Automatizar la serialización/deserialización de JSON: Integrar json_serializable o freezed.

Implementar la persistencia del tema: Modo guardar/cargar tema. * Carga de imágenes consistente en `FieldService`: Use MultipartRequest para todas las cargas de imágenes.

Modularización de componentes de interfaz de usuario: Desglose aún más los métodos de compilación de gran tamaño.

Mejoras de seguridad: Implemente HTTPS, garantice el hash de contraseñas del lado del servidor y considere flutter_secure_storage.

Internacionalización (i18n).

Registro dedicado.

Herramientas de gestión de activos.