Documentação da Arquitetura – Aplicativo *Faça a Festa*

1. Visão Geral

O **Faça a Festa** é um aplicativo multiplataforma (Android, Web) desenvolvido em **Flutter**, com suporte a funcionamento **online e offline**, utilizando o **Firebase** como backend principal.

A arquitetura segue os princípios do **Clean Architecture**, aplicando **TDD (Test-Driven Development)**, **Design Patterns**, os **princípios do SOLID**, e boas práticas de versionamento com **Git (commits pequenos e pontuais)**.

O gerenciamento de estado é realizado com o **GetX**, permitindo uma abordagem reativa e desacoplada entre camadas.

---

2. Objetivos da Arquitetura

- Garantir **separação clara de responsabilidades** (UI, regras de negócio e persistência de dados).
- Permitir **testabilidade** de todas as camadas, com TDD.
- Funcionar de forma **offline-first**, com sincronização automática quando a internet estiver disponível.
- Utilizar **design patterns** para resolver problemas recorrentes (Repository, Strategy, Adapter, Facade, Observer).
- Seguir **SOLID** para maior flexibilidade e extensibilidade.
- Ter um processo de **Git organizado**, com commits granulares e rastreáveis.

---

3. Camadas da Arquitetura (Clean Architecture)

a) Domain Layer

- Contém as **entidades** (regras de negócio puras).
- **Use Cases**: casos de uso da aplicação (ex.: `ListarServicos`, `AgendarEvento`).
- Não depende de nenhuma tecnologia externa (Firebase, Flutter, etc.).
- **Testada primeiro** no TDD.

b) Data Layer

- Implementa os contratos definidos no *Domain*.
- Usa **DataSources** (local e remoto):
  - **Local**: Drift/SQLite (desktop/offline), cache interno ou Firestore offline.
  - **Remoto**: Firebase Firestore, Auth e Storage.
- Faz o mapeamento entre **Models (DTOs) ↔ Entities**.

c) Presentation Layer

- Implementada em Flutter.
- Usa **GetX** para gerenciar estado e dependências.
- **Controllers** consomem *Use Cases* e expõem estados para a UI.
- **Bindings** cuidam da injeção de dependências.
- **Views** (Widgets) reagem às mudanças do estado.

4. Fluxo de Dados

1. O usuário interage com a **UI**.
2. O **Controller** chama o **Use Case**.
3. O **Use Case** usa um **Repository (interface)**.
4. O **Repository** decide (via Delegate/Strategy) se consulta o **LocalDataSource** ou o **RemoteDataSource**.
5. Dados são convertidos para **Entities** e devolvidos para a **UI**.

5. Offline-First e Sincronização

- Toda **leitura** parte do **repositório local** (single source of truth).
- Escritas são registradas no local e marcadas como **pendentes de sync**.
- Um **SyncService** roda periodicamente (ou ao detectar internet) e replica dados para o Firebase.
- Conflitos são resolvidos com **estratégias de merge** (ex.: `last-write-wins` ou merge de campos).

6. Padrões de Projeto (Design Patterns)

- **Repository** → abstrai as fontes de dados.
- **Strategy** → decide entre `local` e `remoto`.
- **Adapter/Mapper** → converte entre `Model` e `Entity`.
- **Facade** → simplifica acesso ao Firebase (Auth, Storage).
- **Observer** → `Stream<List<T>>` com GetX/Firestore.

7. Princípios do SOLID aplicados

- **S (Single Responsibility)** → cada classe com uma única responsabilidade.
- **O (Open/Closed)** → novos repositórios ou estratégias podem ser adicionados sem alterar código existente.
- **L (Liskov Substitution)** → repositórios respeitam contratos de interfaces.
- **I (Interface Segregation)** → interfaces específicas, evitando contratos inchados.
- **D (Dependency Inversion)** → Controllers dependem de interfaces, não de implementações.

8. Gerenciamento de Estado (GetX)

- **Bindings**: definem dependências da feature.
- **Controllers**: usam `Rx<T>` para reatividade.

- **UI**: widgets com `Obx()` reagem automaticamente.

9. Banco de Dados

- **Firebase Firestore** (remoto) para persistência online.
- **Drift/SQLite ou cache Firestore** para funcionamento offline.
- **Firebase Auth** para autenticação de usuários.
- **Firebase Storage** para imagens e arquivos.

10. Testes (TDD)

Pirâmide de Testes

- **Domain** → testes unitários de Use Cases.
- **Data** → testes de repositórios e mapeamentos (com mocks).
- **Presentation** → testes de Controller e Widgets.
- **Integração** → com Firebase Emulator Suite.

Ciclo TDD

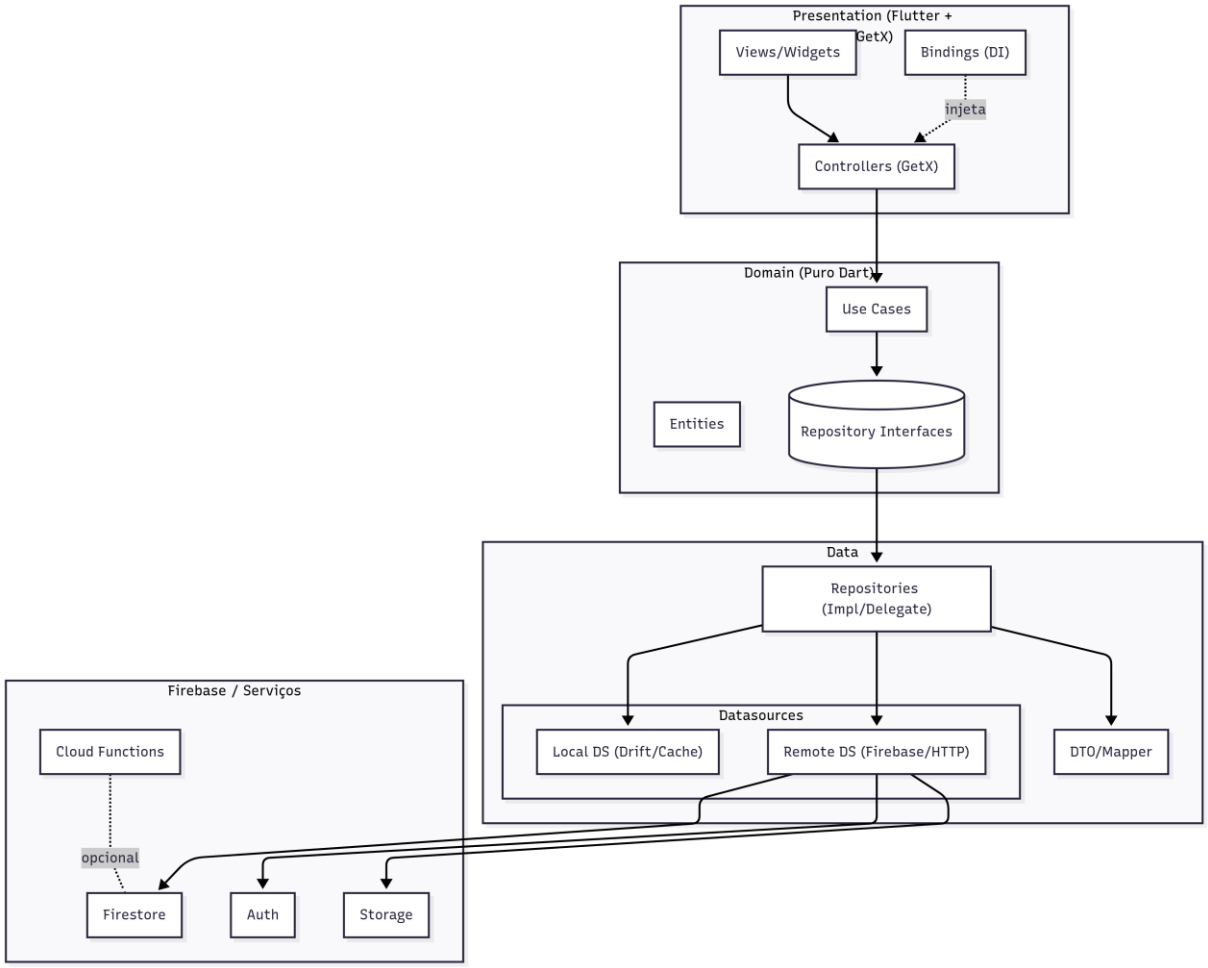
1. **Red** → escrever teste falhando.
2. **Green** → implementar mínimo necessário.
3. **Refactor** → melhorar código mantendo verde.

11. Benefícios da Arquitetura

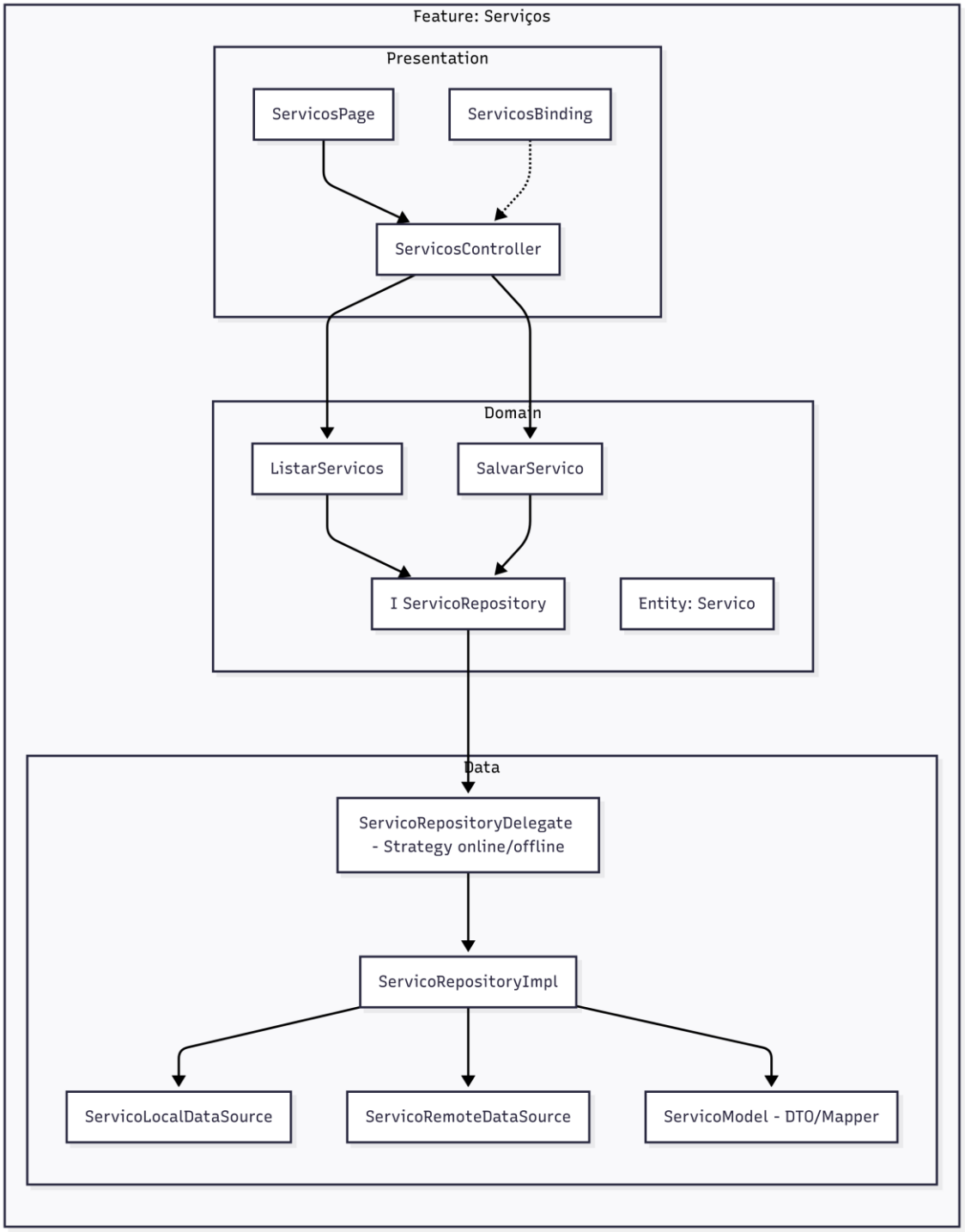
- **Testabilidade** → cada camada é testada isoladamente.
- **Escalabilidade** → fácil incluir novas features sem quebrar código existente.
- **Offline-first** → experiência fluida mesmo sem internet.
- **Organização** → equipe entende papéis de cada camada.
- **Boas práticas** → alinhado a Clean Architecture, SOLID e TDD.

Arquitetura Faça a Festa – Diagramas

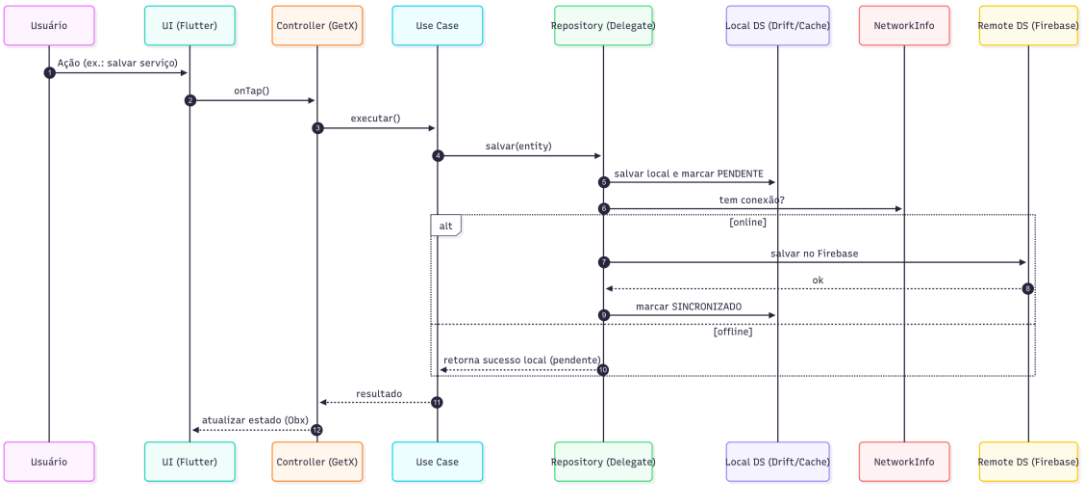
1) Clean Architecture – Visão em Camadas



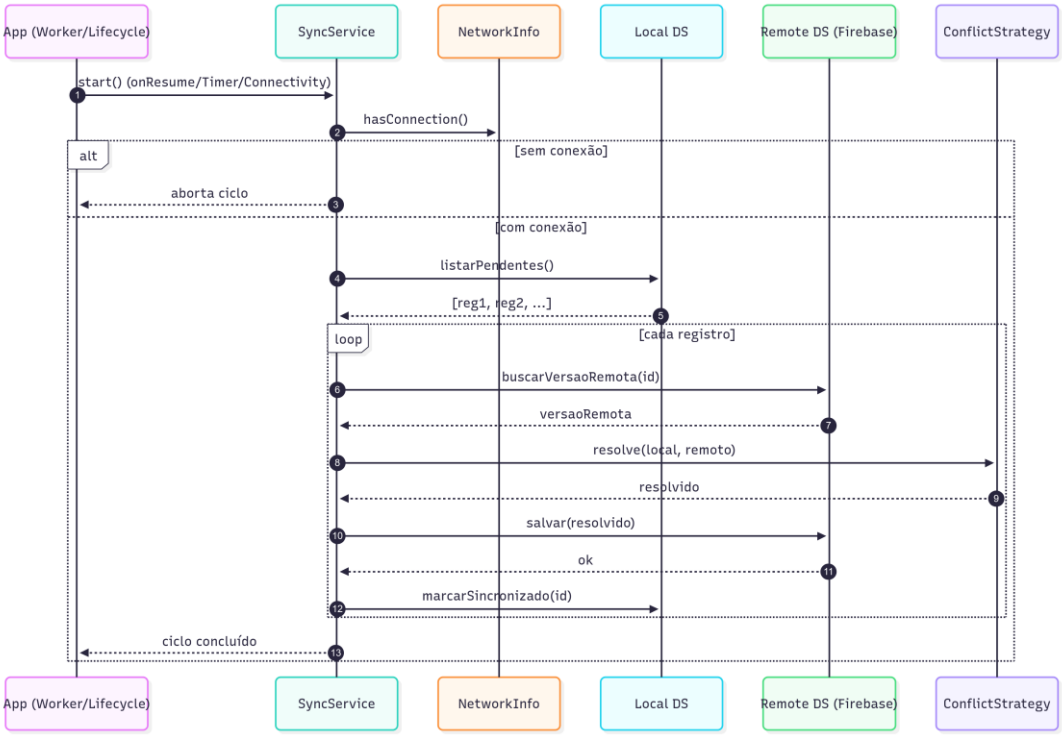
2) Componentes por Feature



3) Sequência – Fluxo **Offline-first** (Leitura/Escrita)



4) Serviço de Sincronização (Background)



5) Deployment / Infra (Ambientes & Ferramentas)