# Documentação da Arquitetura – Aplicativo Faça a Festa

## 1. Visão Geral

O **Faça a Festa** é um aplicativo **multiplataforma** (Android, iOS, Web e Desktop) desenvolvido em **Flutter**, projetado para operar tanto em **modo online** quanto **offline-first**, garantindo usabilidade mesmo em cenários de baixa conectividade.

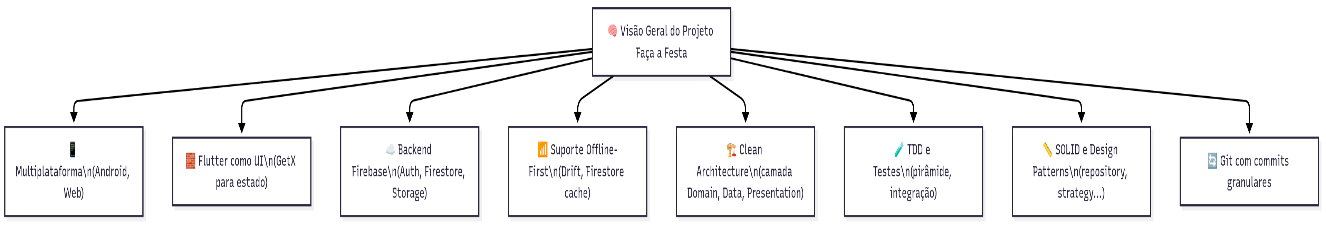
O backend principal é estruturado sobre o **Firebase** (Firestore, Authentication, Storage, Cloud Functions), complementado por banco local (**SQLite/Drift**) para sincronização de dados.

A arquitetura adota os princípios da **Clean Architecture**, promovendo separação de responsabilidades e alta testabilidade. Entre as práticas utilizadas destacam-se:

* **TDD (Test-Driven Development)** para garantir confiabilidade do código;
* **Design Patterns** aplicados (Repository, Strategy, Singleton e Observer);
* **Princípios SOLID** para manter coesão e baixo acoplamento;
* **Boas práticas de versionamento com Git**, priorizando commits pequenos, descritivos e frequentes.

O gerenciamento de estado é realizado com **GetX**, oferecendo reatividade, simplicidade na injeção de dependências e maior desacoplamento entre as camadas de apresentação, domínio e infraestrutura.

Imagem 01:

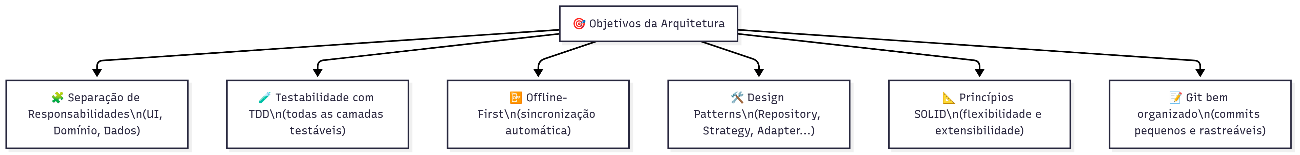


## 2. Objetivos da Arquitetura

A arquitetura do Faça a Festa foi projetada para atender a requisitos de **qualidade, manutenibilidade e escalabilidade**, buscando alinhar boas práticas de engenharia de software com as necessidades do negócio. Os principais objetivos são:

* **Separação de responsabilidades**: manter uma divisão clara entre **camada de apresentação (UI)**, **regras de negócio (Domínio/Aplicação)** e **persistência de dados (Infraestrutura)**.
* **Testabilidade**: possibilitar a cobertura de testes automatizados em todas as camadas, aplicando **TDD** como prática de desenvolvimento.
* **Suporte a offline-first**: garantir funcionamento mesmo sem conectividade, com **sincronização automática** assim que a internet estiver disponível.
* **Uso de Design Patterns**: empregar padrões como **Repository**, **Strategy**, **Adapter**, **Facade** e **Observer** para resolver problemas recorrentes e aumentar a reutilização de código.
* **Aderência aos princípios SOLID**: assegurar **cohesão, baixo acoplamento, extensibilidade e flexibilidade**.
* **Controle de versionamento eficiente**: adotar um processo de **Git organizado**, com **commits pequenos, granulares e rastreáveis**, facilitando auditoria e colaboração em equipe.

Imagem 02:



## 3. Camadas da Arquitetura (Clean Architecture)

A arquitetura segue os princípios da **Clean Architecture**, organizando o código em camadas independentes, de forma que cada uma tenha responsabilidades bem definidas e baixo acoplamento.

### a) Domain Layer (Domínio)

* Contém as **entidades centrais** e **regras de negócio puras**, independentes de qualquer tecnologia ou framework.
* Define os **Use Cases** (casos de uso da aplicação), que representam as operações principais do sistema, como ListarServiços, AgendarEvento e GerenciarConvidados.
* Não possui dependência direta de bibliotecas externas (como Firebase, Flutter ou bancos de dados).
* É a primeira camada a ser validada no **TDD**, garantindo que a lógica de negócio esteja correta antes da implementação técnica.

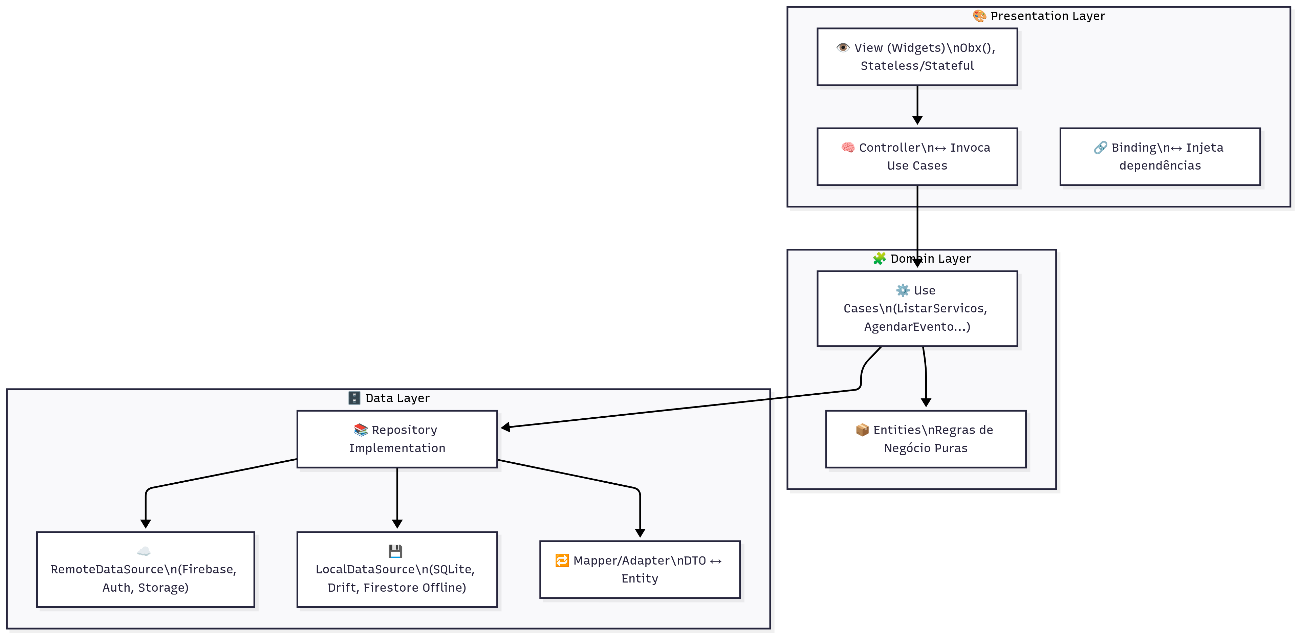
### b) Data Layer (Dados)

* Responsável por **implementar os contratos** definidos no Domain Layer.
* Utiliza **DataSources** para comunicação com os dados:
  + **Local**: Drift/SQLite (para desktop e modo offline), cache interno ou Firestore em modo offline.
  + **Remoto**: Firebase Firestore (banco principal), Firebase Auth (autenticação) e Firebase Storage (armazenamento de arquivos e imagens).
* Realiza o **mapeamento bidirecional** entre **Models/DTOs ↔ Entities**, permitindo que a camada de domínio permaneça independente da tecnologia.

### c) Presentation Layer (Apresentação)

* Desenvolvida em **Flutter**, sendo a interface direta com o usuário.
* Utiliza **GetX** para **gerenciamento de estado** e **injeção de dependências**, proporcionando reatividade e simplicidade no controle da UI.
* Estrutura-se em:
  + **Controllers**: consomem Use Cases do Domain Layer e expõem estados observáveis para a interface.
  + **Bindings**: responsáveis pela **injeção automática de dependências**, garantindo desacoplamento entre módulos.
  + **Views (Widgets)**: reagem a mudanças de estado e exibem os dados na interface do usuário.

Imagem 03:



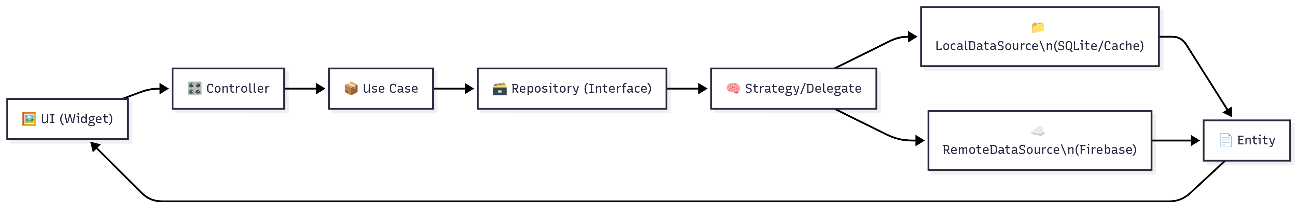
## 4. Fluxo de Dados

O **fluxo de dados** no Faça a Festa segue um ciclo bem definido, garantindo isolamento entre camadas, testabilidade e suporte ao modo **offline-first**.

1. **Interação do Usuário (UI)**
   * O usuário realiza uma ação na interface (ex.: clicar em Confirmar Presença ou Adicionar Serviço).
   * A View (Widget) notifica o **Controller**, que está observando os eventos.
2. **Controller → Use Case**
   * O **Controller** invoca o **Use Case** correspondente no Domain Layer.
   * Exemplo: AgendarEventoUseCase.execute(...).
3. **Use Case → Repository**
   * O **Use Case** depende apenas de uma **interface de Repository**, definida no domínio.
   * Essa abstração garante que a regra de negócio não dependa da implementação técnica.
4. **Repository → DataSources (Local/Remoto)**
   * O **Repository** aplica **Strategy/Delegate** para decidir se a consulta será feita no:
     + **LocalDataSource**: base SQLite/Drift, cache interno ou Firestore offline.
     + **RemoteDataSource**: Firebase Firestore, Auth ou Storage.
   * Essa decisão pode variar conforme: disponibilidade de rede, política de sincronização ou preferências de configuração.
5. **Conversão → Entities → Retorno**
   * Os dados brutos (DTOs/Models) vindos da fonte de dados são **mapeados para Entities**.
   * As **Entities** são devolvidas ao **Use Case**, que retorna ao **Controller**, e este atualiza a **UI** de forma reativa.

🔗 **Resumo:** O usuário aciona a UI → Controller → Use Case → Repository → DataSource (Local/Remoto) → Entities → UI.  
Esse fluxo garante **baixo acoplamento**, **facilidade de testes unitários** e **sincronização transparente entre offline e online**.

Imagem 04:



## 5. Offline-First e Sincronização

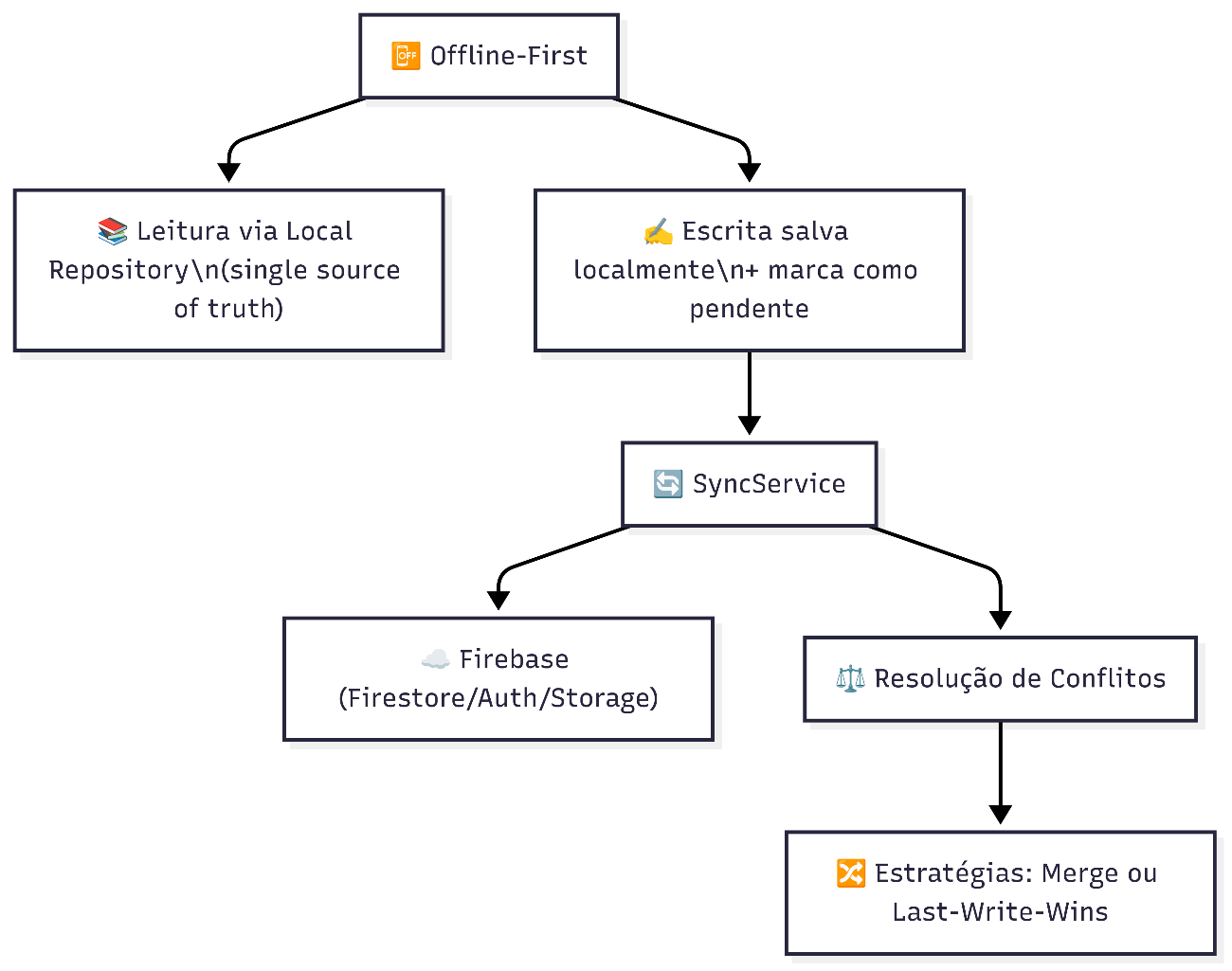
O Faça a Festa adota a estratégia **offline-first**, garantindo que o sistema funcione de forma estável mesmo sem conexão com a internet. O modelo de dados considera o **repositório local como a fonte única da verdade (single source of truth)**.

### Estratégia de Operação

* **Leituras**
  + Toda consulta parte do **banco local** (SQLite/Drift ou Firestore em modo offline).
  + A aplicação exibe os dados imediatamente, sem depender da rede.
* **Escritas**
  + Qualquer alteração (ex.: adicionar convidado, criar evento, registrar pagamento) é **persistida no local**.
  + Os registros são marcados com **status de sincronização pendente** (pending\_sync = true).
* **Serviço de Sincronização (SyncService)**
  + Executa em intervalos configurados ou assim que a conexão com a internet for detectada.
  + Responsável por **replicar as alterações locais para o Firebase** (Firestore/Auth/Storage).
  + Atualizações externas também são puxadas para o banco local, garantindo consistência.
* **Resolução de Conflitos**
  + Estratégias aplicadas conforme o contexto da entidade:
    - **Last-Write-Wins (LWW):** prevalece a última modificação registrada.
    - **Merge de Campos:** para estruturas mais complexas (ex.: detalhes de evento), combina campos de diferentes versões.
  + Logs de sincronização são mantidos para auditoria e rastreabilidade.

🔗 **Resumo:**  
O sistema funciona **mesmo offline**, armazenando todas as ações localmente e sincronizando automaticamente assim que houver internet, evitando perda de dados e garantindo consistência entre dispositivos.

Imagem 05:

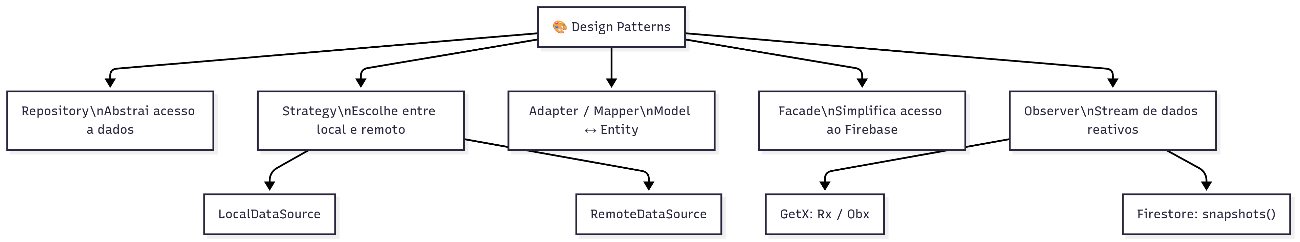


## 6. Padrões de Projeto (Design Patterns)

A arquitetura do Faça a Festa aplica **Design Patterns consagrados** para resolver problemas recorrentes de forma elegante, sustentável e de fácil manutenção:

* **Repository**
  + Abstrai as fontes de dados, expondo apenas contratos claros para os Use Cases.
  + Isola a lógica de persistência, permitindo alternar entre banco local e Firebase sem impactar as regras de negócio.
* **Strategy**
  + Define a política de escolha da fonte de dados.
  + Exemplo: ao buscar convidados, o repositório pode decidir entre **LocalDataSource** (SQLite/Firestore offline) ou **RemoteDataSource** (Firestore online).
* **Adapter / Mapper**
  + Converte entre **Models (DTOs)** e **Entities (Domain)**.
  + Garante que a camada de domínio trabalhe com objetos puros, desacoplados da tecnologia de persistência.
* **Facade**
  + Fornece uma interface simplificada para serviços externos complexos, como **Firebase Auth**, **Firestore** e **Firebase Storage**.
  + Reduz o acoplamento e facilita a manutenção e evolução do backend.
* **Observer**
  + Implementado via **Streams** (ex.: Stream<List<T>>) e integrado com o **GetX** e o **Firestore**.
  + Permite que a UI seja atualizada automaticamente em tempo real quando houver alterações nos dados.

Imagem 06:

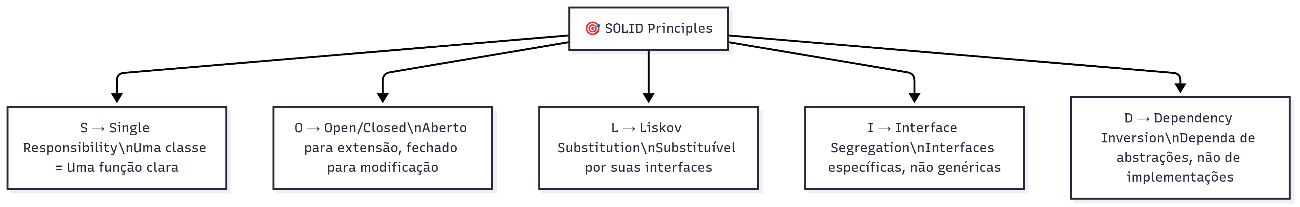


## 7. Princípios do SOLID aplicados

A arquitetura do Faça a Festa segue de forma consistente os **princípios do SOLID**, garantindo maior flexibilidade, testabilidade e facilidade de manutenção do sistema:

* **S – Single Responsibility Principle (Princípio da Responsabilidade Única)**
  + Cada classe tem apenas **uma responsabilidade clara**.
  + Exemplo: EventoRepository cuida apenas da persistência de eventos, enquanto AgendarEventoUseCase concentra a regra de negócio do agendamento.
* **O – Open/Closed Principle (Aberto/Fechado)**
  + O sistema é **aberto para extensão e fechado para modificação**.
  + Novos repositórios, estratégias de sincronização ou validações podem ser adicionados sem alterar código existente.
* **L – Liskov Substitution Principle (Substituição de Liskov)**
  + Todas as implementações respeitam os **contratos de interfaces**.
  + Exemplo: qualquer IEventoRepository pode substituir outra implementação (offline ou online) sem quebrar a aplicação.
* **I – Interface Segregation Principle (Segregação de Interfaces)**
  + São definidas **interfaces específicas e coesas**, evitando contratos genéricos ou “inchados”.
  + Exemplo: IUsuarioAuthRepository é separado de IUsuarioPerfilRepository, para não misturar responsabilidades.
* **D – Dependency Inversion Principle (Inversão de Dependência)**
  + Os **Controllers** e **Use Cases** dependem de **interfaces**, nunca de implementações concretas.
  + Isso facilita a aplicação de **TDD**, permitindo mocks e substituições em testes sem impactar o código real.

Imagem 07:



## 8. Gerenciamento de Estado (GetX)

O Faça a Festa utiliza o **GetX** como solução principal para **gerenciamento de estado, injeção de dependências e navegação**, proporcionando uma arquitetura enxuta e altamente reativa.

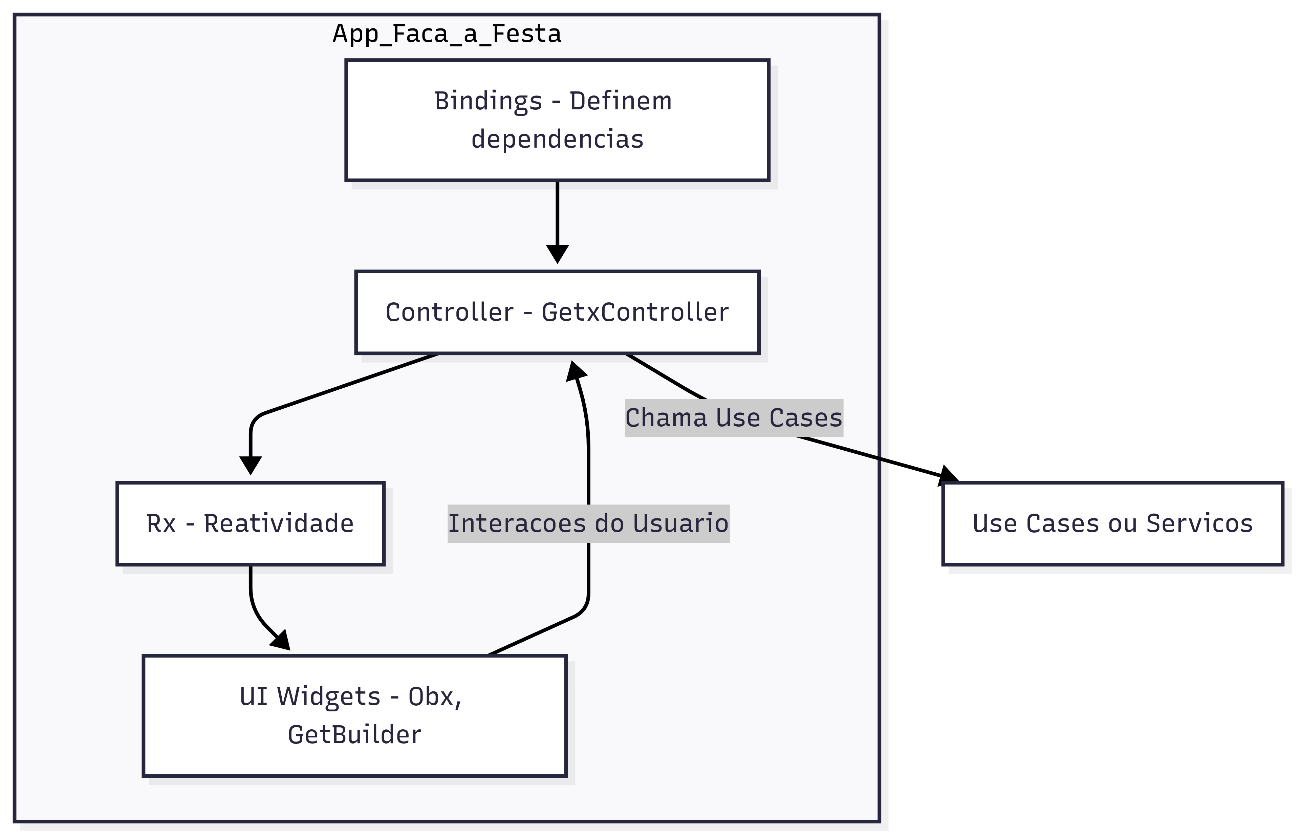
### Componentes-Chave

* **Bindings**
  + Definem as **dependências de cada feature**.
  + São executados automaticamente na inicialização da rota/tela.
  + Exemplo: ao abrir a tela de Cadastro de Evento, o EventoBinding injeta EventoController, EventoRepository e Use Cases relacionados.
* **Controllers**
  + Centralizam a lógica da apresentação.
  + Utilizam **observáveis (Rx<T>)** para armazenar estados reativos.
  + Consomem Use Cases da camada de domínio e expõem dados prontos para a UI.
  + Exemplo: EventoController contém RxList<Evento> que é atualizado a partir dos repositórios.
* **UI (Views/Widgets)**
  + Construída em Flutter.
  + Usa **Obx()** ou **GetBuilder** para reagir automaticamente às mudanças de estado.
  + A interface é atualizada em tempo real sem necessidade de setState().

### Benefícios do GetX no Projeto

* **Reatividade completa**: dados atualizados instantaneamente na tela.
* **Menos boilerplate**: código mais limpo comparado a soluções mais verbosas.
* **Desacoplamento**: UI, Controller e Repositórios trabalham de forma independente.
* **Escalabilidade**: cada módulo tem seus próprios bindings e controllers, facilitando manutenção e evolução.

Imagem 08:



## 9. Banco de Dados

A arquitetura do Faça a Festa adota uma abordagem **híbrida e offline-first**, integrando Firebase como principal backend e bases locais para suporte em situações sem conectividade.

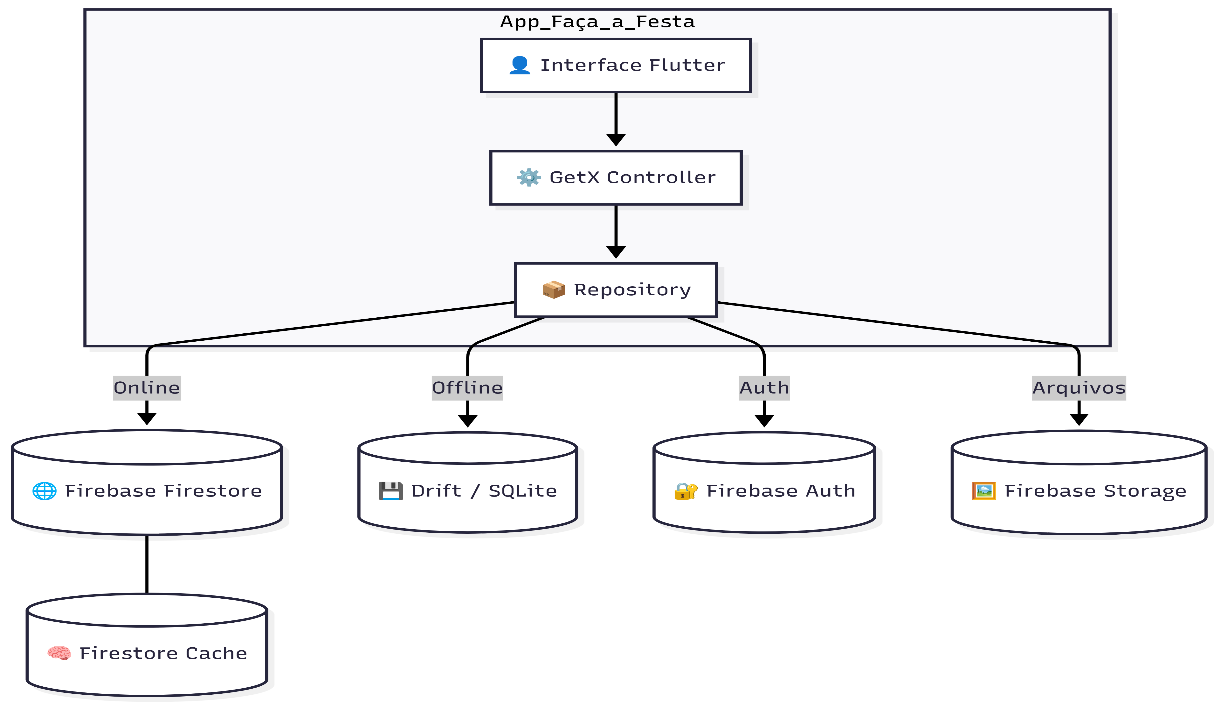
### Componentes Utilizados

* **Firebase Firestore (Remoto)**
  + Base de dados **NoSQL em nuvem**.
  + Persistência **online e em tempo real**, com sincronização automática entre dispositivos.
  + Estrutura flexível, organizada em coleções (ex.: usuarios, eventos, fornecedores, orcamentos).
* **Drift/SQLite (Local)**
  + Banco **relacional embarcado** utilizado em modo offline.
  + Serve como **single source of truth** para leitura e escrita inicial.
  + Posteriormente, os dados são sincronizados com o Firestore pelo SyncService.
* **Cache Offline do Firestore**
  + Como alternativa ao Drift, pode-se habilitar o **cache nativo do Firestore**, garantindo consultas rápidas sem conexão.
  + Indicado para dispositivos com restrição de memória.
* **Firebase Auth (Autenticação)**
  + Gerencia usuários e credenciais.
  + Suporte a **email/senha, Google, Facebook e Apple Sign-In**.
  + Tokens de autenticação integram-se ao Firestore para controle de permissões.
* **Firebase Storage (Arquivos e Imagens)**
  + Utilizado para armazenamento de **fotos de eventos, fornecedores, convites e referências visuais**.
  + Integrado com regras de segurança baseadas em autenticação.

### Benefícios

* **Escalabilidade**: Firestore lida com alto volume de leituras/escritas.
* **Offline-first**: toda operação pode ser feita sem internet, com posterior sincronização.
* **Segurança**: integração entre Auth, Firestore e Storage garante controle de acesso granular.
* **Flexibilidade**: combinação entre dados relacionais (SQLite) e NoSQL (Firestore).

Imagem 09:



## 10. Testes (TDD)

A qualidade do Faça a Festa é garantida através de uma abordagem **orientada a testes (Test-Driven Development – TDD)**. Isso assegura que cada camada da arquitetura seja validada de forma isolada e integrada.

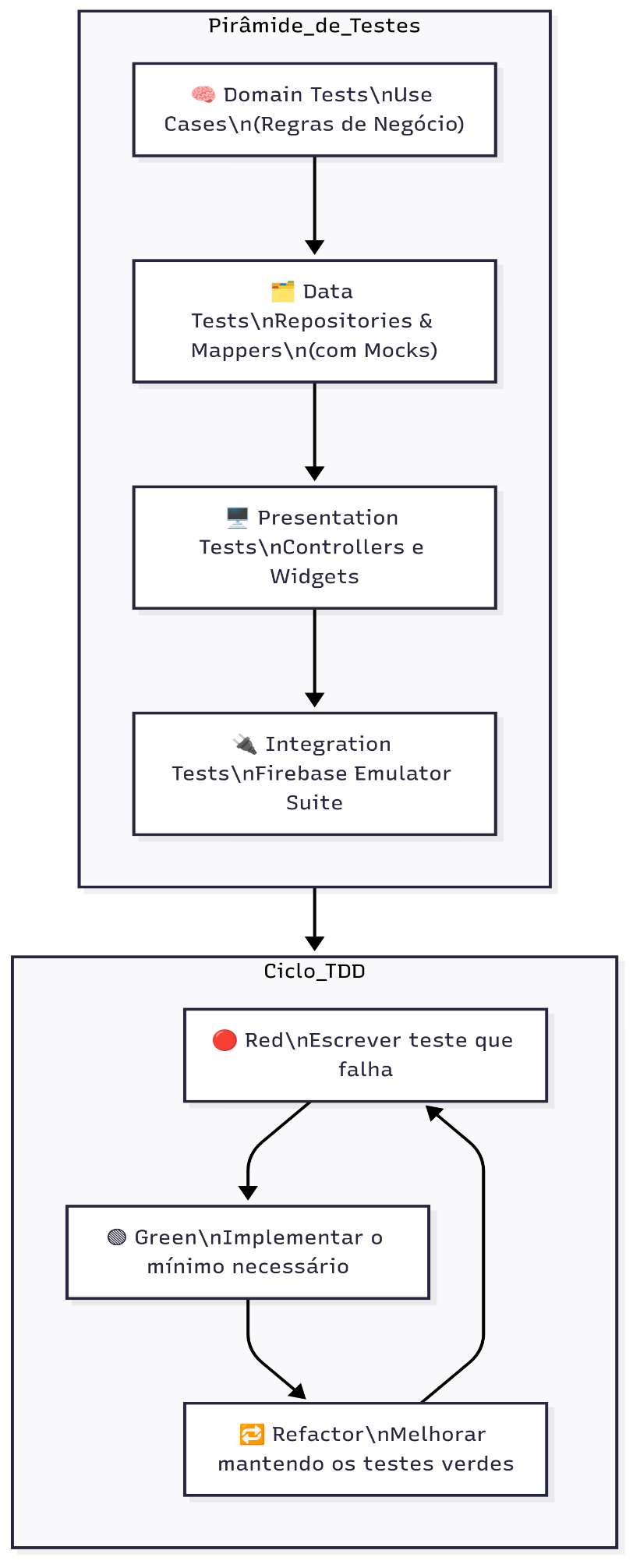
### Pirâmide de Testes

* **Domain (Base da pirâmide)**
  + Testes unitários de **casos de uso (Use Cases)**.
  + Validam as **regras de negócio puras**, sem dependência de frameworks.
  + Exemplo: CalcularOrcamentoTotalTest, ValidarConvidadoPresencaTest.
* **Data (Camada intermediária)**
  + Testes de **repositórios, mapeamentos e fontes de dados (DataSources)**.
  + Uso de **mocks/stubs** para simular Firestore, Drift/SQLite ou APIs externas.
  + Exemplo: UsuarioRepositoryMockTest, EventoMapperTest.
* **Presentation (Topo da pirâmide)**
  + Testes de **Controllers e Widgets** no Flutter.
  + Garantem que as mudanças no estado do Controller reflitam corretamente na UI.
  + Exemplo: EventoControllerWidgetTest.
* **Integração (Cobertura cruzada)**
  + Executados com **Firebase Emulator Suite** (Firestore, Auth, Functions).
  + Validam fluxos completos: cadastro de usuário, criação de evento, convite e orçamento.

### Ciclo TDD aplicado

1. **Red** → escrever um teste que inicialmente falha.
   * Ex.: Criar teste para validar que Evento.nome não pode ser vazio.
2. **Green** → implementar o mínimo necessário para passar no teste.
   * Ex.: Adicionar regra simples de validação no Evento.
3. **Refactor** → melhorar o código mantendo os testes verdes.
   * Ex.: Extrair lógica de validação para um Validator reutilizável.

Imagem 10:



## 11. Benefícios da Arquitetura

A arquitetura do Faça a Festa foi planejada para equilibrar **qualidade técnica, escalabilidade e experiência do usuário**, garantindo robustez e flexibilidade para evolução contínua.

### Principais Benefícios

* **Testabilidade**
  + Cada camada (Domain, Data, Presentation) pode ser testada isoladamente.
  + O uso de TDD garante que funcionalidades sejam validadas antes mesmo da implementação completa.
* **Escalabilidade**
  + Estrutura modular que permite adicionar novas features sem comprometer código existente.
  + Exemplo: inclusão de um novo meio de pagamento ou integração com outro serviço de convite digital sem alterar fluxos centrais.
* **Offline-first**
  + Usuário continua utilizando o app sem conexão.
  + Sincronização transparente com Firestore quando a internet volta, evitando perda de dados.
* **Organização e Clareza**
  + A divisão clara entre **camadas** (Domain, Data, Presentation) facilita entendimento e manutenção.
  + Novos desenvolvedores conseguem compreender rapidamente o fluxo da aplicação.
* **Aderência a Boas Práticas**
  + Uso de **Clean Architecture, SOLID e TDD** garante código limpo, reutilizável e sustentável a longo prazo.
  + Padrões de projeto (Repository, Strategy, Adapter, etc.) resolvem problemas recorrentes com consistência.

Imagem 11:

