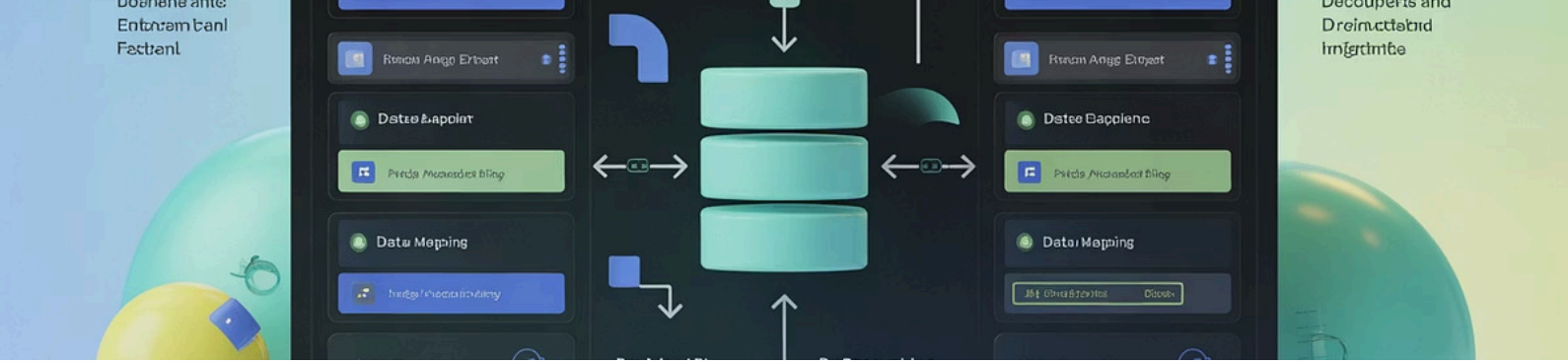

ProvidersRepository: Do Conceito à Implementação

No capítulo anterior, estabelecemos as bases da arquitetura orientada a domínios e o papel das abstrações na separação clara entre UI, regras de negócio e infraestrutura. Exploramos como programar “contra contratos” nos permite evoluir fontes de dados, estratégias de cache e integrações externas sem fricção para a camada de apresentação. Agora, avançamos do “porquê” arquitetural para o “como” concreto: formalizamos o repositório como a porta de entrada estável do domínio para acesso a fornecedores.

Neste capítulo, apresentamos o contrato do `ProvidersRepository`, responsável por prover uma experiência fluida à aplicação — do carregamento inicial via cache local à listagem consolidada e filtragens específicas — e detalhamos a fronteira remota via `ProvidersRemoteApi`, que lida com a sincronização incremental e paginação de dados. Mantemos, assim, o mapeamento rigoroso entre DTOs e Entities e preservamos o desacoplamento entre domínio e infraestrutura.

`ProvidersRepository`

A porta de entrada estável do domínio, garantindo uma interface fluida para carregamento, listagem e filtragem de dados de fornecedores, abstraindo a origem.

`ProvidersRemoteApi`

Gerencia a interação com a fonte de dados remota, implementando a sincronização incremental e a paginação de dados para eficiência e escalabilidade.

Mapeamento DTOs & Entities

Assegura a conversão precisa entre os formatos de dados da rede (DTOs) e as representações do domínio (Entities), mantendo a integridade do modelo de negócio.

Desacoplamento

A separação explícita entre domínio e infraestrutura, promovendo um sistema robusto, testável e flexível a futuras mudanças nas fontes de dados ou tecnologias.

Em outras palavras, saímos da visão conceitual para codificar os pontos de extensão que tornam o sistema robusto, testável e pronto para escalar, garantindo que a aplicação seja resiliente e performática.

`ProvidersRepository`: O Contrato de Domínio Essencial

Chegamos ao coração da nossa arquitetura baseada em abstrações: o contrato `ProvidersRepository`. Esta `abstract class` serve como o ponto de entrada principal e mais estável para a camada de apresentação (UI) e a lógica de negócio do FoodSafe interagirem com os dados dos fornecedores. Sua existência garante que a UI nunca precise se preocupar com a complexidade de onde os dados vêm (remoto ou local), como são armazenados ou como são convertidos. Ela promete sempre entregar **Entities** puras, validadas e prontas para uso.

```
import '../entities/provider.dart';

abstract class ProvidersRepository {
  /// Render inicial rápido a partir do cache local.
  Future<List<Provider>> loadFromCache();

  /// Sincronização incremental (>= lastSync).
  /// Retorna quantos registros mudaram.
  Future<int> syncFromServer();

  /// Listagem completa (normalmente do cache após sync).
  Future<List<Provider>> listAll();

  /// Destaques (filtrados do cache por `featured`).
  Future<List<Provider>> listFeatured();

  /// Opcional: busca direta por ID no cache.
  Future<Provider?> getById(int id);
}
```

Ao implementar este contrato, estamos programando contra uma abstração. Isso significa que podemos trocar a implementação subjacente (por exemplo, mudar a fonte de dados ou a estratégia de cache) sem que uma única linha de código da UI precise ser alterada. Os métodos definidos aqui são os pilares dessa interação limpa e desacoplada:

`loadFromCache()`

Proporciona uma renderização inicial rápida do aplicativo, carregando dados diretamente do cache local. Isso garante uma experiência de usuário ágil, especialmente em cenários offline ou com conexões instáveis.

`syncFromServer()`

Responsável pela sincronização incremental dos dados do servidor. Ele busca apenas as alterações mais recentes a partir de um determinado ponto no tempo (`lastSync`), minimizando o tráfego de rede e otimizando o consumo de recursos. Retorna a quantidade de registros que foram alterados.

`listAll()`

Retorna a listagem completa de todos os fornecedores, geralmente após uma sincronização bem-sucedida, garantindo que a UI sempre trabalhe com o conjunto de dados mais atualizado e consolidado disponível no cache local.

`listFeatured()`

Filtra e retorna apenas os fornecedores marcados como "destaque" (`featured`) do cache local. Ideal para seções de UI que precisam exibir provedores com algum tipo de promoção ou visibilidade especial.

`getById(int id)`

Permite a busca direta e eficiente de um fornecedor específico por seu identificador único (ID) no cache local. Isso é útil para exibir detalhes de um único provedor sem a necessidade de carregar toda a lista.

A clareza e a simplicidade deste contrato são fundamentais para construir um aplicativo robusto, testável e fácil de manter, onde as preocupações de domínio são isoladas das complexidades da infraestrutura.

`ProvidersRemoteApi` — O Contrato de Acesso Remoto ao Supabase

A `ProvidersRemoteApi` representa a fronteira entre nossa aplicação e o mundo exterior, especificamente com o backend Supabase. Sua finalidade é ser um contrato estrito para a comunicação remota, lidando exclusivamente com os detalhes da base de dados (como nomes de colunas, formatação de datas e metadados) e retornando **DTOs** (Data Transfer Objects) que espelham essa estrutura.

Um aspecto crucial da sincronização de dados é a estratégia `updated_at DESC` + `>= since`. Esta abordagem garante uma sincronização incremental eficiente, buscando apenas registros que foram alterados ou criados a partir de um determinado ponto no tempo. Ordenar por `updated_at DESC` e usar um filtro inclusivo `>= since` simplifica a lógica e, mais importante, evita "buracos" na sincronização que poderiam ocorrer se timestamps emparelhados resultassem em registros perdidos em iterações subsequentes.

```dart
import '../dtos/provider_dto.dart';

class PageCursor {
  final String? value; // p.ex., next offset/last id token
  const PageCursor(this.value);
}

class RemotePage<T> {
  final List<T> items;
  final PageCursor? next;
  const RemotePage({required this.items, this.next});
}

abstract class ProvidersRemoteApi {
  /// Busca página de providers ordenada por updated_at DESC.
  /// since => filtro inclusivo (>= since).
  Future<RemotePage<ProviderDto>> fetchProviders({
    DateTime? since,
    int limit = 200,
    PageCursor? cursor,
  });

  // Futuro: escrita/edição no remoto, se o app vier a suportar:
  // Future<void> upsertProviders(List<ProviderDto> dtos);
}
```

Este contrato define uma única operação principal para a leitura de dados:

`fetchProviders()`

Este método é responsável por buscar uma página de provedores do Supabase. Ele permite a filtragem por `since` para sincronizações incrementais, a definição de um `limit` para controlar a quantidade de dados por requisição e um `cursor` para navegação entre páginas. Ao retornar um `RemotePage<ProviderDto>`, a API assegura que a aplicação possa lidar com grandes volumes de dados de forma escalável e eficiente.

As escolhas de design para a `ProvidersRemoteApi` são fundamentadas em princípios de robustez e escalabilidade:

Retorno Paginado

A utilização de um tipo de retorno paginado (`RemotePage`) é fundamental para preparar a aplicação para lidar com bases de dados que podem crescer exponencialmente. Ao invés de tentar "puxar tudo" de uma vez, o que consumiria recursos desnecessariamente e levaria a falhas em larga escala, a paginação garante que apenas um subconjunto gerenciável de dados seja transferido em cada requisição, otimizando o uso de rede e memória.

DTO-only

A decisão de que a `ProvidersRemoteApi` deve operar exclusivamente com **DTOs** é uma medida de segurança e desacoplamento. Ela evita que detalhes específicos da implementação do backend (como nomes de colunas ou particularidades de um ORM) "vazem" para as camadas de domínio ou apresentação da aplicação. A conversão de **DTOs** para **Entities** e vice-versa é responsabilidade exclusiva do **Mapper** dentro do **Repository**, mantendo assim uma clara separação de responsabilidades e uma arquitetura mais limpa e flexível.