Guia Completo: Provider + MVVM (Flutter)

Este documento resume tudo o que foi ensinado neste chat sobre:

- Uso do **package provider**
- MVVM aplicado ao Flutter
- Pastas: `dto/`, `model/`, `repository/`, `provider/`, `ui/`
- Estrutura recomendada
- Exemplo completo
- Boas práticas

1. O que é o Provider?

Provider é um gerenciador de estado baseado em `InheritedWidget`, usado para:

- Compartilhar estado entre widgets
- Criar e descartar objetos automaticamente
- Fazer injeção de dependência
- Reagir a mudanças via `ChangeNotifier`

Principais métodos de leitura:

```
context.watch<T>(); // Rebuilda o widget
context.read<T>();  // Apenas lê, não rebuilda
context.select<T,R>((value) => value.x); // Escuta parte de um objeto
```

2. MVVM no Flutter

A arquitetura MVVM separa:

- **Model** → Entidades de negócio (puro Dart)
- **View** → UI (widgets)
- **ViewModel** → Estado da tela + lógica de apresentação
  No Flutter, implementado com `ChangeNotifier` dentro da pasta **provider/**

Visão macro:

```
View (UI)
   ↕
ViewModel (provider)
   ↕
Repository
   ↕
DTO ↔ API/BD
   ↕
Model
```

3. Estrutura de Pastas Recomendada

Para estudos e projetos simples:

```
lib/
  dto/
  model/
  repository/
  viewmodel/
  ui/
  main.dart
```

4. Função de Cada Pasta

`model/`

Entidades puras da regra de negócio:

```dart
class Juice {
  final String id;
  final String name;
  final int ml;
}
```

`dto/`

Objetos de transporte (JSON):

```dart
class JuiceDto {
  factory JuiceDto.fromJson(Map json);
  Map<String, dynamic> toJson();
  Juice toModel();
}
```

`repository/`

Acesso a dados:

```
abstract class JuiceRepository {
  Future<List<Juice>> getAll();
}
```

Implementação:

```
class JuiceRepositoryImpl implements JuiceRepository {
  @override
  Future<List<Juice>> getAll() async { ... }
}
```

`provider/` (ViewModels)

Gerenciam estado e lógica da UI:

```
class JuiceListViewModel extends ChangeNotifier {
  List<Juice> juices = [];
  bool isLoading = false;
  String? errorMessage;
  Future<void> load() async { ... }
}
```

`ui/`

Telas:

```
final vm = context.watch<JuiceListViewModel>();
```

5. Exemplo Completo

Registro no main.dart

```
MultiProvider(
  providers: [
    Provider(create: (_) => JuiceRepositoryImpl()),
    ChangeNotifierProvider(
      create: (context) => JuiceListViewModel(
        context.read<JuiceRepository>(),
      ),
    ),
  ],
  child: MyApp(),
);
```

ViewModel

```
class JuiceListViewModel extends ChangeNotifier {
  final JuiceRepository _repository;

  JuiceListViewModel(this._repository);

  List<Juice> juices = [];
  bool isLoading = false;

  Future<void> load() async {
    isLoading = true;
    notifyListeners();

    juices = await _repository.getAll();
    isLoading = false;
    notifyListeners();
  }
}
```

Tela

```
class JuiceListPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final vm = context.watch<JuiceListViewModel>();

    if (vm.isLoading) return CircularProgressIndicator();

    return ListView(
      children: vm.juices
          .map((j) => ListTile(title: Text(j.name)))
          .toList(),
    );
  }
}
```

6. Decisão Sobre o Nome da Pasta `provider/`

Apesar de representar ViewModels, manter o nome **provider/**:

- É comum no ecossistema Flutter
- Alinha com o package
- Mantém o código fácil de entender por outros devs

Portanto, a escolha final:

`provider/ ← ViewModels`

7. Resumo Final

- Provider gerencia estado via ChangeNotifier.
- MVVM organiza o projeto com clareza.
- DTO → Repository → Provider(ViewModel) → UI é o fluxo recomendado.
- Estrutura adotada:

`dto/ model/ repository/ provider/ ui/`

- Você domina agora o essencial para criar apps Flutter escaláveis e limpos.

Fim do documento.