Arquitetura e Organização do Código

Desenvolvimento de Aplicativos Móveis - Flutter

Introdução

Uma aplicação bem estruturada facilita manutenção, testes e colaboração em equipe. Nesta seção, vamos estabelecer uma arquitetura clara para o módulo de autenticação que pode escalar conforme o projeto cresce. Vamos aprender a organizar nosso código de forma que ele seja fácil de entender, manter e expandir no futuro.

A forma como organizamos nosso código hoje determina a facilidade ou dificuldade que teremos para adicionar novas funcionalidades amanhã. Uma boa arquitetura é um investimento que se paga rapidamente à medida que o projeto evolui.

Por Que Organização Importa

Antes de mergulharmos na estrutura específica, precisamos entender por que dedicar tempo à organização é fundamental, especialmente em um módulo crítico como autenticação.

Quando começamos um projeto, é tentador colocar todo o código em poucos arquivos grandes. Inicialmente isso funciona, o código é simples e conseguimos encontrar tudo facilmente. Mas à medida que o projeto cresce, essa abordagem se torna um pesadelo. Encontrar onde está a lógica de login entre milhares de linhas de código, entender quais arquivos dependem de quais, e fazer mudanças sem quebrar outras partes do sistema se torna cada vez mais difícil.

Uma arquitetura bem pensada resolve esses problemas antes que eles apareçam. Ela cria uma estrutura previsível onde cada desenvolvedor sabe exatamente onde encontrar ou adicionar código. Ela separa responsabilidades de forma que mudanças em uma parte do sistema não causam efeitos colaterais inesperados em outras partes. E ela facilita testes, permitindo que você teste cada componente isoladamente.

Estrutura de Pastas por Features

A organização por features agrupa todo código relacionado a uma funcionalidade específica em um único diretório. Esta abordagem é mais intuitiva que separar por tipo de arquivo, onde todos os models ficam em uma pasta, todos os services em outra, e assim por diante.

Vamos considerar a diferença prática. Na organização por tipo de arquivo, se você precisa trabalhar em autenticação, seus arquivos estão espalhados por toda a aplicação. O model do usuário está em uma pasta models no topo do projeto, o service de autenticação está em outra pasta services, as telas estão em pages, e

assim por diante. Você precisa navegar por múltiplos diretórios apenas para entender como autenticação funciona.

Na organização por features, tudo relacionado a autenticação está junto em um diretório auth. Abra essa pasta e você vê imediatamente todos os componentes que fazem autenticação funcionar. Isso não apenas facilita navegação, mas também torna o código mais modular, pois cada feature é relativamente autocontida.

Vou propor uma estrutura específica que equilibra organização clara com flexibilidade para crescimento. Esta estrutura foi testada em projetos reais e se mostrou eficaz tanto para projetos pequenos quanto para aplicações complexas.

```
lib/
├── core/
│   ├── constants/
│   │   └── app_constants.dart
│   ├── theme/
│   │   └── app_theme.dart
│   └── utils/
│       └── validators.dart
├── features/
│   └── auth/
│       ├── data/
│       │   ├── models/
│       │   │   └── user_model.dart
│       │   ├── repositories/
│       │   │   └── auth_repository.dart
│       │   └── services/
│       │       └── auth_service.dart
│       ├── presentation/
│       │   ├── pages/
│       │   │   ├── login_page.dart
│       │   │   ├── register_page.dart
│       │   │   ├── forgot_password_page.dart
│       │   │   └── auth_wrapper.dart
│       │   ├── widgets/
│       │   │   ├── custom_text_field.dart
│       │   │   ├── auth_button.dart
│       │   │   └── social_login_button.dart
│       │   └── providers/
│       │       └── auth_provider.dart
│       └── domain/
│           ├── entities/
│           │   └── user.dart
│           └── usecases/
│               ├── login_usecase.dart
│               ├── register_usecase.dart
│               └── logout_usecase.dart
└── main.dart
```

Vamos entender cada parte desta estrutura e por que ela está organizada desta forma.

O Diretório Core

O diretório core contém código compartilhado que pode ser usado por múltiplas features da aplicação. É importante que este diretório contenha apenas código verdadeiramente compartilhado, não código específico de

uma feature.

Dentro de core, temos constants onde ficam valores constantes usados em toda aplicação, como URLs de API, chaves de armazenamento, duração de timeouts, e outros valores que você quer definir uma vez e usar em vários lugares. Centralizar constantes aqui facilita mudanças, pois você altera em um único lugar ao invés de procurar valores duplicados por todo o código.

O subdiretório theme contém a definição do tema visual da aplicação. Aqui você define cores, tipografia, estilos de botões, campos de texto, e todos os outros aspectos visuais que devem ser consistentes por toda aplicação. Com Material Design 3, isso se torna ainda mais importante, pois você pode definir um sistema de cores base e o Flutter gera automaticamente variações adequadas.

O diretório utils contém funções utilitárias reutilizáveis. Para autenticação especificamente, aqui é onde colocamos validators, que são funções para validar email, senha, e outros campos. Essas validações serão usadas tanto nas telas de login quanto de cadastro, então faz sentido centralizá-las em um único lugar.

O Diretório Features

Cada funcionalidade principal da aplicação tem seu próprio diretório dentro de features. Para autenticação, temos features/auth que contém tudo relacionado a essa funcionalidade. Se no futuro você adicionar outras features como profile, settings, ou notifications, cada uma terá seu próprio diretório no mesmo nível de auth.

Esta organização torna muito fácil adicionar, remover ou modificar features sem afetar o resto da aplicação. Se você decidir refatorar completamente como autenticação funciona, todos os arquivos que precisa modificar estão claramente identificados dentro de features/auth.

Separação em Camadas Dentro de Features

Dentro de cada feature, organizamos o código em três camadas inspiradas em Clean Architecture, que é um padrão arquitetural focado em separação de responsabilidades e independência de frameworks.

A Camada de Presentation

A camada presentation contém tudo relacionado à interface do usuário. Aqui ficam as pages, que são as telas completas da aplicação como login_page, register_page e forgot_password_page. Cada tela é responsável por exibir informação ao usuário e capturar suas interações.

Os widgets são componentes reutilizáveis de interface que podem ser usados em múltiplas telas. Por exemplo, custom_text_field é um campo de texto estilizado seguindo o padrão visual da sua aplicação. Ao invés de duplicar todo o código de estilização em cada tela, você cria um widget reutilizável e usa em todos os lugares. Isso mantém consistência visual e facilita mudanças futuras, pois você modifica o widget em um lugar e a mudança se reflete em todas as telas que o usam.

Os providers são responsáveis pelo gerenciamento de estado específico da interface. Usando Riverpod, por exemplo, aqui você definiria o auth_provider que gerencia o estado de autenticação e expõe métodos para login, logout e outras operações. As telas escutam este provider e reagem automaticamente quando o estado muda, reconstruindo a interface conforme necessário.

Um aspecto importante da camada de presentation é que ela não deve conhecer detalhes de implementação de como os dados são obtidos. A tela de login não precisa saber se você está usando Supabase, Firebase ou uma API customizada. Ela apenas chama métodos do provider, que por sua vez orquestra as operações necessárias usando as outras camadas.

A Camada de Domain

A camada domain contém a lógica de negócio pura da aplicação. Este é o coração do seu sistema, onde as regras do que sua aplicação faz estão definidas, independentemente de detalhes técnicos de implementação.

As entities são representações puras dos conceitos do seu domínio. A entidade User, por exemplo, representa o conceito de um usuário no contexto da sua aplicação. Ela contém apenas os atributos essenciais e comportamentos, sem nenhuma dependência de frameworks, bibliotecas de networking ou bancos de dados. É Dart puro.

Por que essa pureza importa? Porque torna esta camada extremamente testável e portável. Você pode testar suas entidades e regras de negócio sem precisar de nenhuma infraestrutura externa. E se no futuro decidir migrar de Flutter para outra tecnologia, esta camada pode ser reutilizada quase sem modificações.

Os usecases representam as ações que usuários podem realizar na aplicação. Login_usecase encapsula toda a lógica de o que significa fazer login: validar credenciais, obter token, salvar sessão. Ao encapsular cada ação em seu próprio usecase, você torna o código mais legível e testável. Cada usecase tem uma responsabilidade clara e bem definida.

A Camada de Data

A camada data lida com a origem e destino dos dados. Ela implementa os detalhes técnicos de como obter e armazenar informações, isolando essas complexidades das outras camadas.

Os models são representações dos dados como eles existem externamente, seja vindo de uma API JSON, de um banco de dados local, ou de outra fonte. O user_model, por exemplo, sabe como converter JSON recebido de uma API em um objeto Dart, e vice-versa. Ele contém lógica de serialização e deserialização específica para o formato usado pela sua fonte de dados.

Note a diferença sutil mas importante entre models e entities. Uma entity User representa o conceito puro de usuário no domínio da aplicação. Um model UserModel representa como os dados de usuário são estruturados na API ou banco de dados. Você converte models em entities ao trazer dados para dentro da aplicação, e entities em models ao enviar dados para fora. Esta conversão acontece nos repositories.

Os services contém o código que faz chamadas específicas para APIs, SDKs ou outras fontes de dados externas. O auth_service, por exemplo, faz as chamadas HTTP para endpoints de login e cadastro do Supabase. Ele lida com detalhes técnicos como construção de URLs, serialização de bodies de requisição, e parsing de respostas.

Os repositories implementam interfaces definidas no domain e servem como ponte entre as camadas de domain e data. Um repository decide de onde buscar os dados, pode combinar múltiplas fontes se necessário, e converte models em entities antes de retornar para a camada de domain. Esta abstração permite que você mude implementações de data sem afetar o resto da aplicação. Por exemplo, você poderia trocar Supabase por Firebase modificando apenas o repository e service, sem tocar em nenhuma lógica de negócio ou interface.

Fluxo de Dados Entre Camadas

Entender como dados fluem entre essas camadas é crucial para trabalhar efetivamente com esta arquitetura. Vamos traçar o caminho completo de uma operação de login para ver todas as peças trabalhando juntas.

O usuário digita email e senha na tela de login e toca no botão "Entrar". A login_page captura esta interação e chama um método do auth_provider passando as credenciais. O provider então invoca o login_usecase, que contém a lógica de negócio de autenticação. O usecase chama o auth_repository solicitando que faça login com as credenciais fornecidas.

O repository, por sua vez, chama o auth_service que faz a requisição HTTP real para a API do Supabase. A API responde com dados JSON. O service converte esse JSON em um UserModel usando os métodos de parsing do model. O repository pega esse UserModel e o converte em uma entity User pura. Esta entity User sobe pela pilha, sendo retornada para o usecase, depois para o provider, e finalmente a tela atualiza a interface com os dados do usuário autenticado.

Se em qualquer ponto deste fluxo ocorrer um erro, por exemplo, erro de rede no service ou credenciais inválidas retornadas pela API, a exceção propaga pela pilha sendo transformada e enriquecida em cada camada até chegar na tela, que pode então mostrar uma mensagem apropriada ao usuário.

Este fluxo pode parecer complexo inicialmente, especialmente comparado a simplesmente fazer uma chamada HTTP direto de uma tela. Mas os benefícios se tornam evidentes conforme o projeto cresce. Cada camada tem uma responsabilidade clara, mudanças são localizadas, testes são mais fáceis, e você pode modificar implementações sem efeitos em cascata por toda aplicação.

Quando Simplificar a Arquitetura

É importante reconhecer que a arquitetura completa que apresentei pode ser excessiva para projetos muito simples ou protótipos rápidos. Se você está fazendo um app de estudo pessoal ou um MVP rápido para validar uma ideia, pode ser pragmático começar com uma estrutura mais simples.

Para projetos pequenos, você pode fundir as camadas domain e data, eliminando a conversão entre models e entities. Você pode ter os services chamando diretamente de dentro dos providers, sem a camada intermediária de usecases. A chave é entender os princípios por trás da separação, para que quando o projeto crescer e a complexidade justificar, você saiba como refatorar para a arquitetura completa.

O perigo é cair no extremo oposto e não ter nenhuma organização. Mesmo em projetos pequenos, pelo menos separe apresentação de lógica de negócio. Não coloque chamadas de API diretamente em widgets. Não misture lógica de validação com código de construção de interface. Esses princípios básicos valem sempre, independentemente do tamanho do projeto.

Gerenciamento de Estado para Autenticação

O estado de autenticação é especial porque precisa ser acessível por toda aplicação. Múltiplas telas precisam saber se o usuário está logado, quem é o usuário atual, e reagir quando ele faz login ou logout. Vamos entender como implementar isso efetivamente.

Existem várias soluções de gerenciamento de estado no Flutter, incluindo Provider, BLoC, MobX, GetX e Riverpod. Para autenticação, vou recomendar Riverpod por sua simplicidade, poder e excelente integração com Flutter moderno. Mas os conceitos que vou apresentar se aplicam a outras soluções também.

O estado de autenticação tipicamente contém algumas informações essenciais. Primeiro, um indicador de se o usuário está autenticado ou não. Segundo, os dados do usuário atual quando autenticado, como id, email e nome. Terceiro, um indicador de loading para operações em andamento, como durante um login. E quarto, possíveis mensagens de erro que precisam ser mostradas ao usuário.

Podemos modelar esses diferentes estados usando classes. Ter estados explícitos ao invés de múltiplas variáveis booleanas torna o código mais fácil de entender e menos propenso a bugs. Você nunca fica em um estado impossível como "autenticado mas sem dados de usuário" porque cada estado representa uma situação válida e completa.

Aqui está como estruturamos esses estados usando classes Dart:

dart

```dart
// Representa o usuário autenticado
class User {
  final String id;
  final String email;
  final String? name;
  final String? photoUrl;

  const User({
    required this.id,
    required this.email,
    this.name,
    this.photoUrl,
  });
}

// Estados possíveis da autenticação
abstract class AuthState {
  const AuthState();
}

// Estado inicial ao abrir o app, ainda verificando se há sessão salva
class AuthStateInitial extends AuthState {
  const AuthStateInitial();
}

// Durante operações de login, logout, etc
class AuthStateLoading extends AuthState {
  const AuthStateLoading();
}

// Usuário está autenticado com sucesso
class AuthStateAuthenticated extends AuthState {
  final User user;

  const AuthStateAuthenticated(this.user);
}

// Usuário não está autenticado
class AuthStateUnauthenticated extends AuthState {
  const AuthStateUnauthenticated();
}

// Erro durante operação de autenticação
class AuthStateError extends AuthState {
  final String message;
```

```
  const AuthStateError(this.message);
}
```

Com esses estados definidos, criamos um StateNotifier que gerencia transições entre estados e expõe métodos para operações de autenticação. Este notifier vive durante toda vida útil da aplicação e widgets podem escutá-lo para reagir a mudanças.

A beleza desta abordagem é que widgets automaticamente recebem notificações quando o estado muda. Se o usuário faz login, o estado muda para AuthStateAuthenticated, e qualquer widget escutando esse estado reconstrói sua interface automaticamente. Não há necessidade de gerenciamento manual de listeners ou callbacks complexos.

Nomenclatura e Convenções

Uma parte importante de código bem organizado é nomenclatura consistente e clara. Vamos estabelecer convenções que tornam o código mais legível.

Para arquivos, use snake_case com nomes descritivos. O arquivo auth_service.dart contém AuthService, login_page.dart contém LoginPage. A correspondência entre nome de arquivo e classe principal facilita navegação.

Para classes, use PascalCase. AuthService, UserModel, LoginPage. Para variáveis e métodos, use camelCase começando com minúscula. emailController, validatePassword, handleLogin. Para constantes, use SCREAMING_SNAKE_CASE. API_BASE_URL, MAX_LOGIN_ATTEMPTS.

Nomes devem ser descritivos o suficiente para entender o propósito sem precisar ler a implementação, mas não tão longos que fiquem inconvenientes. Prefira emailController a e ou emailTextEditingControllerForLoginForm. O equilíbrio é clareza com concisão.

Para métodos, use verbos que descrevem a ação. login, validateEmail, saveToken. Para variáveis booleanas, prefira is ou has seguido de um adjetivo. isLoading, hasError, isAuthenticated. Isso torna condicionais mais legíveis, como if (isLoading) ao invés de if (loading).

Estrutura de Arquivos Específicos

Vamos ver exemplos de como estruturar arquivos específicos dentro desta arquitetura, começando com componentes simples e construindo até estruturas mais complexas.

Um arquivo de widget reutilizável deve ser autocontido e focado. Aqui está a estrutura de custom_text_field.dart:

```
dart
```

```dart
import 'package:flutter/material.dart';

/// Widget customizado de campo de texto seguindo padrão visual da aplicação.
/// Encapsula estilização e comportamentos comuns como toggle de senha.
class CustomTextField extends StatefulWidget {
  final String label;
  final String? hint;
  final IconData? prefixIcon;
  final bool isPassword;
  final TextEditingController? controller;
  final String? Function(String?)? validator;
  final TextInputType? keyboardType;
  final TextInputAction? textInputAction;
  final void Function(String)? onFieldSubmitted;

  const CustomTextField({
    Key? key,
    required this.label,
    this.hint,
    this.prefixIcon,
    this.isPassword = false,
    this.controller,
    this.validator,
    this.keyboardType,
    this.textInputAction,
    this.onFieldSubmitted,
  }) : super(key: key);

  @override
  State<CustomTextField> createState() => _CustomTextFieldState();
}

class _CustomTextFieldState extends State<CustomTextField> {
  // Estado local para controlar visibilidade de senha
  bool _obscureText = true;

  @override
  void initState() {
    super.initState();
    // Inicializa obscureText baseado em isPassword
    _obscureText = widget.isPassword;
  }

  @override
  Widget build(BuildContext context) {
    return TextFormField(
```

```
controller: widget.controller,
obscureText: widget.isPassword && _obscureText,
keyboardType: widget.keyboardType,
textInputAction: widget.textInputAction,
validator: widget.validator,
onFieldSubmitted: widget.onFieldSubmitted,
decoration: InputDecoration(
 labelText: widget.label,
 hintText: widget.hint,
 prefixIcon: widget.prefixIcon != null
    ? Icon(widget.prefixIcon)
    : null,
 suffixIcon: widget.isPassword
    ? IconButton(
        icon: Icon(
          _obscureText ? Icons.visibility : Icons.visibility_off,
        ),
        onPressed: () {
          setState(() {
            _obscureText = !_obscureText;
          });
        },
      )
    : null,
 filled: true,
 fillColor: Theme.of(context)
    .colorScheme
    .surfaceVariant
    .withOpacity(0.5),
 border: OutlineInputBorder(
  borderRadius: BorderRadius.circular(12),
  borderSide: BorderSide.none,
 ),
 enabledBorder: OutlineInputBorder(
  borderRadius: BorderRadius.circular(12),
  borderSide: BorderSide.none,
 ),
 focusedBorder: OutlineInputBorder(
  borderRadius: BorderRadius.circular(12),
  borderSide: BorderSide(
    color: Theme.of(context).colorScheme.primary,
    width: 2,
  ),
 ),
 errorBorder: OutlineInputBorder(
  borderRadius: BorderRadius.circular(12),
  borderSide: BorderSide(
```

```
            color: Theme.of(context).colorScheme.error,
            width: 2,
          ),
        ),
      ),
    );
  }
}
```

Este arquivo mostra boas práticas. Começa com um comentário de documentação explicando o propósito do widget. Os parâmetros são bem nomeados e tipados. Estado local obscureText é privado e gerenciado internamente. A implementação encapsula toda complexidade de estilização, expondo uma interface simples para quem usa.

Conclusão da Seção

Nesta seção, estabelecemos a fundação arquitetural para nosso módulo de autenticação. Aprendemos por que organização importa, como estruturar pastas por features, como separar código em camadas com responsabilidades claras, e como essas camadas trabalham juntas em um fluxo de dados coerente.

Entender e aplicar esta arquitetura desde o início economiza tempo e frustração conforme o projeto cresce. O investimento inicial em organização se paga rapidamente através de código mais fácil de entender, modificar e testar.

Na próxima seção, vamos aplicar esta arquitetura na prática, detalhando cada tela do fluxo de autenticação e como elas se conectam dentro desta estrutura organizada.

Fim da Seção 2: Arquitetura e Organização do Código