Documento de Arquitetura: Fluxo de Autenticação Reativa em Flutter

Este documento detalha a arquitetura de um fluxo de login e navegação reativa, utilizando as bibliotecas `provider` para gerenciamento de estado e `go_router` para navegação.

1. Visão Geral da Arquitetura

A arquitetura implementada segue o padrão **MVVM (Model-View-ViewModel)**, combinado com princípios de **Injeção de Dependência** e **Navegação Centralizada**.

As principais camadas são:

- **`main.dart`**: O ponto de entrada. Responsável por configurar e prover todas as dependências do aplicativo (Serviços, Repositórios, ViewModels) usando `MultiProvider`.
- **`presentation` (UI)**: A camada de interface do usuário, dividida por *features* (funcionalidades como `login`, `menu`).
  - **View (`_screen.dart`)**: A tela que o usuário vê. É "burra" e apenas exibe o estado e encaminha as ações do usuário para o ViewModel.
  - **ViewModel (`_viewmodel.dart`)**: O cérebro da View. Contém a lógica de estado da tela e responde às ações do usuário, delegando tarefas de negócio para a camada de `data`.
- **`data`**: A camada de lógica de negócio e acesso a dados.
  - **Repository**: Abstrai a origem dos dados. A View/ViewModel não sabe se os dados vêm de uma API, banco de dados local ou cache.
  - **Service**: A implementação concreta do acesso aos dados (ex: `HttpApiService` que faz as chamadas de API).
- **`core`**: Contém código compartilhado por todo o aplicativo, como o `SessionViewModel` (que gerencia o estado de autenticação global) e temas.
- **`navigation`**: Contém a configuração centralizada de navegação, usando o `GoRouter`.

2. O Fluxo de Login: A Cadeia de Eventos

A principal característica desta arquitetura é que a navegação não é feita manualmente. Em vez disso, ela **reage** a uma mudança no estado de autenticação do aplicativo.

Aqui está o passo a passo do que acontece quando o usuário clica no botão "Login":

Passo 1: A View (Tela de Login)

- **Arquivo**: `lib/presentation/features/login/view/login_screen.dart`
- **Ação**: O usuário clica no `LoginButton`. O `onPressed` do botão chama o método `performLogin` do `LoginViewModel`.
- **Código Chave**:

```
// Dentro do onPressed do botão de login
viewModel.performLogin(
  context: context, // Passa o BuildContext
```

```
  username: _userController.text,
  password: _passwordController.text,
);
```

- **Responsabilidade**: Apenas capturar a intenção do usuário e delegá-la ao `LoginViewModel`, fornecendo os dados necessários (usuário, senha) e o `BuildContext` (essencial para o `Provider` funcionar).

Passo 2: O ViewModel da Tela (LoginViewModel)

- **Arquivo**: `lib/presentation/features/login/viewmodel/login_viewmodel.dart`
- **Ação**: O método `performLogin` é executado. Ele chama o `AuthRepository` para validar as credenciais com a API. Se a API retornar sucesso, a mágica começa.
- **Código Chave**:

```
// Dentro do método performLogin, após o sucesso da API...
final user = AppUser(username: username, ...);

// Ação mais importante:
Provider.of<SessionViewModel>(context, listen: false).loginSuccess(user);
```

- **Responsabilidade**: Orquestrar a lógica da tela de login. Ele **não sabe para qual tela navegar**. Sua única responsabilidade após um login bem-sucedido é encontrar o `SessionViewModel` (usando o `context` e o `Provider`) e notificá-lo sobre o sucesso, entregando os dados do usuário.

Passo 3: O ViewModel de Sessão (Estado Global)

- **Arquivo**: `lib/core/viewmodels/session_viewmodel.dart`
- **Ação**: O método `loginSuccess` é chamado pelo `LoginViewModel`. Ele atualiza seu estado interno e notifica todo o aplicativo sobre essa mudança.
- **Código Chave**:

```
void loginSuccess(AppUser user) {
  _currentUser = user;

  // Grita para todos os "ouvintes" que o estado da sessão mudou.
  notifyListeners();
}
```

- **Responsabilidade**: Ser a **única fonte da verdade** sobre o estado de autenticação do aplicativo. Ele mantém os dados do usuário logado e notifica qualquer parte interessada quando esse estado muda (login ou logout).

Passo 4: O Roteador (O Cérebro da Navegação)

- **Arquivo**: `lib/navigation/app_router.dart`
- **Ação**: O `GoRouter` foi configurado para "ouvir" o `SessionViewModel` através do `refreshListenable`. Quando o `notifyListeners()` é chamado, o `GoRouter` automaticamente reavalia sua lógica de `redirect`.

- **Código Chave**:

```
late final router = GoRouter(
  // 1. O GoRouter está ouvindo o SessionViewModel
  refreshListenable: _sessionViewModel,

  // 2. A lógica de redirect é re-executada
  redirect: (context, state) {
    final bool isLoggedIn = _sessionViewModel.isLoggedIn; // Agora é `true`
    final String currentLocation = state.matchedLocation;   // É '/login'

    // 3. Esta condição se torna verdadeira
    if (isLoggedIn && currentLocation == '/login') {
      // 4. A regra retorna a string do caminho de destino
      return '/menu';
    }
    return null;
  },
);
```

- **Responsabilidade**: Agir como o "guarda de trânsito" do aplicativo. Com base no estado de autenticação (`isLoggedIn`) e na localização atual do usuário, ele decide se permite o acesso, bloqueia ou redireciona. Ao receber a string `'/menu'` , ele realiza a navegação para a tela de menu.

3. Vantagens Desta Arquitetura

1. **Desacoplamento**: O `LoginViewModel` não sabe nada sobre a `MenuScreen` . A `MenuScreen` não sabe nada sobre o `LoginViewModel` . A navegação é controlada por um sistema centralizado que apenas observa o estado.
2. **Centralização de Regras**: Toda a lógica de "quem pode ir para onde" está em um único lugar (`app_router.dart`), tornando-a fácil de entender e modificar.
3. **Testabilidade**: É mais fácil testar cada parte isoladamente. Você pode testar o `LoginViewModel` sem precisar de um `BuildContext` ou de um sistema de navegação real.
4. **Robustez**: Protege contra acesso indevido a rotas. Como o `GoRouter` guarda todas as rotas, é impossível um usuário não logado acessar uma tela protegida.
5. **Deep Linking**: Esta arquitetura está pronta para *deep linking* (abrir o app em uma tela específica a partir de um link externo) sem esforço adicional.