Navegação e Proteção de Rotas

Desenvolvimento de Aplicativos Móveis - Flutter

Introdução

A navegação em aplicações com autenticação precisa ser inteligente o suficiente para saber quais telas o usuário pode acessar baseado em seu estado de autenticação. Um usuário não autenticado não deveria conseguir acessar a tela de perfil, e um usuário já autenticado não deveria ser forçado a passar pela tela de login novamente. Além disso, precisamos lidar com deep linking, onde usuários clicam em links de emails de confirmação ou recuperação de senha e são levados diretamente para a tela apropriada dentro do aplicativo.

Nesta seção, vamos construir um sistema de navegação que protege rotas automaticamente baseado no estado de autenticação, implementa transições suaves entre estados, e suporta deep linking para fluxos críticos como confirmação de email e redefinição de senha. O objetivo é criar uma experiência de navegação que parece natural e fluida, onde usuários são guiados para os lugares certos no momento certo sem intervenção manual.

Navegação Baseada em Estado com Auth Wrapper

O padrão fundamental de navegação em aplicações com autenticação é o Auth Wrapper que vimos anteriormente. Este widget serve como o diretor de tráfego da aplicação, decidindo continuamente qual tela deve estar visível baseado no estado atual de autenticação. Vamos revisitar e expandir este conceito para entender completamente como ele funciona e por que é tão poderoso.

O Auth Wrapper não é uma tela em si, mas sim um widget que renderiza outras telas condicionalmente. Ele escuta o estado de autenticação e, baseado nesse estado, decide se deve mostrar a splash screen durante verificação inicial, a tela de login para usuários não autenticados, ou a área principal do aplicativo para usuários autenticados. A beleza deste padrão é que toda a lógica de decisão está centralizada em um único lugar, e as transições entre estados acontecem automaticamente sem código de navegação manual espalhado por múltiplas telas.

Vamos implementar um Auth Wrapper completo que lida com todos os estados possíveis de forma elegante:

```dart
```

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

/// Widget que orquestra a navegação baseada em estado de autenticação
/// Este é o widget raiz da aplicação após MaterialApp
class AuthWrapper extends ConsumerWidget {
  const AuthWrapper({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // Escuta o estado de autenticação
    // Quando muda, este widget reconstrói automaticamente
    final authState = ref.watch(authProvider);

    // Durante estado inicial, mostra splash screen
    // Este estado aparece brevemente ao abrir o app enquanto
    // verifica se há sessão salva
    if (authState is AuthStateInitial) {
      return const SplashScreen();
    }

    // Se usuário está autenticado, mostra área principal do app
    // Passa os dados do usuário para a home page
    if (authState is AuthStateAuthenticated) {
      return MainNavigator(user: authState.user);
    }

    // Para todos os outros estados (Unauthenticated, Loading, Error)
    // mostra o fluxo de autenticação (login/registro)
    // As telas individuais lidam com seus próprios estados de loading e erro
    return const AuthNavigator();
  }
}

/// Navegador para área autenticada do app com bottom navigation
class MainNavigator extends StatefulWidget {
  final User user;

  const MainNavigator({
    Key? key,
    required this.user,
  }) : super(key: key);

  @override
  State<MainNavigator> createState() => _MainNavigatorState();
}
```

```dart
class _MainNavigatorState extends State<MainNavigator> {
  int _currentIndex = 0;

  // Lista de telas da área autenticada
  late final List<Widget> _pages;

  @override
  void initState() {
    super.initState();
    _pages = [
      HomePage(user: widget.user),
      ExplorePage(),
      ProfilePage(user: widget.user),
    ];
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: _pages[_currentIndex],
      bottomNavigationBar: NavigationBar(
        selectedIndex: _currentIndex,
        onDestinationSelected: (index) {
          setState(() {
            _currentIndex = index;
          });
        },
        destinations: const [
          NavigationDestination(
            icon: Icon(Icons.home_outlined),
            selectedIcon: Icon(Icons.home),
            label: 'Início',
          ),
          NavigationDestination(
            icon: Icon(Icons.explore_outlined),
            selectedIcon: Icon(Icons.explore),
            label: 'Explorar',
          ),
          NavigationDestination(
            icon: Icon(Icons.person_outline),
            selectedIcon: Icon(Icons.person),
            label: 'Perfil',
          ),
        ],
      ),
    );
```

```
  }
}

/// Navegador para área não autenticada (login, registro)
class AuthNavigator extends StatelessWidget {
  const AuthNavigator({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    // Usa Navigator nomeado para fluxo de autenticação
    return Navigator(
      initialRoute: '/login',
      onGenerateRoute: (settings) {
        switch (settings.name) {
          case '/login':
            return MaterialPageRoute(
              builder: (_) => const LoginPage(),
            );
          case '/register':
            return MaterialPageRoute(
              builder: (_) => const RegisterPage(),
            );
          case '/forgot-password':
            return MaterialPageRoute(
              builder: (_) => const ForgotPasswordPage(),
            );
          default:
            return MaterialPageRoute(
              builder: (_) => const LoginPage(),
            );
        }
      },
    );
  }
}
```

Esta estrutura separa claramente a área autenticada da não autenticada. O AuthNavigator gerencia as telas de login, registro e recuperação de senha usando seu próprio Navigator interno. O MainNavigator gerencia a área principal do app com bottom navigation. O AuthWrapper simplesmente escolhe qual exibir baseado no estado. Quando o usuário faz login com sucesso e o estado muda para AuthStateAuthenticated, o Flutter automaticamente transiciona de AuthNavigator para MainNavigator. Não há chamadas explícitas de push ou pop, apenas renderização condicional reativa ao estado.

Este padrão resolve elegantemente o problema de pilha de navegação. Quando o usuário está na tela de cadastro e completa o registro, fazendo login automaticamente, não queremos que a tela de cadastro permaneça na pilha de navegação. Com este padrão, toda a árvore de AuthNavigator é simplesmente substituída por MainNavigator.

Não há telas antigas escondidas na pilha. Se o usuário fizer logout mais tarde, o MainNavigator é substituído por AuthNavigator, e começa limpo na tela de login novamente.

Navegação Manual para Fluxos Específicos

Embora o Auth Wrapper lide com transições de alto nível automaticamente, ainda precisamos de navegação manual para fluxos específicos dentro de cada área. Por exemplo, dentro do AuthNavigator, o usuário precisa navegar manualmente de login para registro ou recuperação de senha. Vamos ver como implementar estas navegações de forma limpa e consistente.

A navegação no Flutter pode ser feita de duas formas principais. Navegação imperativa usando Navigator.push e pop, ou navegação declarativa usando rotas nomeadas. Para fluxos de autenticação, recomendo navegação imperativa porque oferece mais controle e é mais direta para fluxos simples. Para aplicações mais complexas com deep linking extensivo, rotas nomeadas podem ser preferíveis.

Vamos ver exemplos de navegação entre telas do fluxo de autenticação:

dart

```dart
// Na tela de login, navegando para registro
TextButton(
  onPressed: () {
    Navigator.of(context).push(
      MaterialPageRoute(
        builder: (context) => const RegisterPage(),
      ),
    );
  },
  child: const Text('Não tem conta? Cadastre-se'),
)

// Na tela de login, navegando para recuperação de senha
TextButton(
  onPressed: () {
    Navigator.of(context).push(
      MaterialPageRoute(
        builder: (context) => const ForgotPasswordPage(),
      ),
    );
  },
  child: const Text('Esqueceu a senha?'),
)

// Na tela de registro, voltando para login após completar
ElevatedButton(
  onPressed: () async {
    // Tenta registrar
    await ref.read(authProvider.notifier).register(
      email: _emailController.text,
      password: _passwordController.text,
    );

    // Se chegou aqui, registro foi bem-sucedido
    // Volta para login mostrando mensagem de sucesso
    if (mounted) {
      Navigator.of(context).pop();
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('Conta criada! Verifique seu email para confirmar.'),
          backgroundColor: Colors.green,
        ),
      );
    }
  },
```

```
  child: const Text('Criar conta'),
)
```

Note como usamos Navigator.of(context).push para avançar e Navigator.of(context).pop para voltar. Este é o padrão básico de navegação em pilha que o Flutter usa. Quando você push uma tela, ela é colocada no topo da pilha. Quando você pop, a tela atual é removida e a anterior reaparece. O botão de voltar nativo do Android automaticamente faz pop, então você não precisa implementar esse comportamento manualmente.

Uma consideração importante é sempre verificar if (mounted) antes de fazer navegação após operações assíncronas. Durante a operação assíncrona, o widget pode ter sido disposed, e tentar navegar causaria erro. Esta verificação previne crashes em cenários de borda como o usuário fechando o app enquanto uma requisição está em andamento.

Transições de Navegação Customizadas

As transições padrão do Flutter funcionam bem, mas você pode customizá-las para criar uma experiência mais polida e consistente com a identidade visual do seu aplicativo. Transições customizadas também podem comunicar semanticamente o tipo de navegação, por exemplo usando slide para navegação lateral entre telas de mesmo nível, e fade para transições entre níveis hierárquicos diferentes.

Vamos implementar algumas transições customizadas comuns:

dart

```dart
/// Transição com fade suave entre telas
class FadePageRoute<T> extends PageRouteBuilder<T> {
  final Widget page;

  FadePageRoute({required this.page})
      : super(
          pageBuilder: (context, animation, secondaryAnimation) => page,
          transitionsBuilder: (context, animation, secondaryAnimation, child) {
            return FadeTransition(
              opacity: animation,
              child: child,
            );
          },
          transitionDuration: const Duration(milliseconds: 300),
        );
}

/// Transição com slide de baixo para cima (padrão iOS)
class SlideUpPageRoute<T> extends PageRouteBuilder<T> {
  final Widget page;

  SlideUpPageRoute({required this.page})
      : super(
          pageBuilder: (context, animation, secondaryAnimation) => page,
          transitionsBuilder: (context, animation, secondaryAnimation, child) {
            const begin = Offset(0.0, 1.0);
            const end = Offset.zero;
            const curve = Curves.easeInOut;

            var tween = Tween(begin: begin, end: end).chain(
              CurveTween(curve: curve),
            );

            return SlideTransition(
              position: animation.drive(tween),
              child: child,
            );
          },
          transitionDuration: const Duration(milliseconds: 350),
        );
}

/// Transição combinando fade e scale (zoom sutil)
class ScalePageRoute<T> extends PageRouteBuilder<T> {
  final Widget page;
```

```
  ScalePageRoute({required this.page})
      : super(
          pageBuilder: (context, animation, secondaryAnimation) => page,
          transitionsBuilder: (context, animation, secondaryAnimation, child) {
            var fadeAnimation = animation;
            var scaleAnimation = Tween<double>(
              begin: 0.95,
              end: 1.0,
            ).animate(
              CurvedAnimation(
                parent: animation,
                curve: Curves.easeInOut,
              ),
            );

            return FadeTransition(
              opacity: fadeAnimation,
              child: ScaleTransition(
                scale: scaleAnimation,
                child: child,
              ),
            );
          },
          transitionDuration: const Duration(milliseconds: 300),
        );
}

// Uso das transições customizadas
Navigator.of(context).push(
  FadePageRoute(page: const RegisterPage()),
);

Navigator.of(context).push(
  SlideUpPageRoute(page: const ProfileEditPage()),
);
```

Estas transições customizadas criam uma sensação mais polida e podem ajudar a estabelecer hierarquia visual. Por exemplo, você pode usar fade para transições entre login e registro que são telas de mesmo nível, e slide up para abrir modais ou telas que representam uma ação secundária como editar perfil. A consistência nas transições ajuda usuários a construir um modelo mental de como o aplicativo está estruturado.

Deep Linking para Fluxos de Email

Deep linking é a capacidade de abrir o aplicativo diretamente em uma tela específica através de um URL. Isto é

essencial para fluxos que começam com emails, como confirmação de conta ou redefinição de senha. O usuário clica em um link no email, e ao invés de abrir no navegador, o link abre o aplicativo diretamente na tela apropriada, já com os parâmetros necessários extraídos do URL.

Configurar deep linking envolve duas partes. Primeiro, configurar o sistema operacional para reconhecer que certos URLs devem abrir seu aplicativo. Segundo, configurar o aplicativo para processar esses URLs e navegar para a tela apropriada.

Configuração no Android

No arquivo android/app/src/main/AndroidManifest.xml, adicione um intent filter dentro da tag activity principal:

```xml
<activity
    android:name=".MainActivity"
    android:launchMode="singleTop"
    android:theme="@style/LaunchTheme"
    android:configChanges="orientation|keyboardHidden|keyboard|screenSize|smallestScreenSize|locale|layoutDirection|fontS
    android:hardwareAccelerated="true"
    android:windowSoftInputMode="adjustResize">

    <!-- Intent filter para deep links -->
    <intent-filter android:autoVerify="true">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <!-- Substitua por seu domínio real -->
        <data
            android:scheme="https"
            android:host="seuapp.com"
            android:pathPrefix="/auth" />
    </intent-filter>

    <!-- Intent filter para esquema customizado (opcional) -->
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data android:scheme="seuapp" />
    </intent-filter>
</activity>
```

Esta configuração diz ao Android que URLs como https://seuapp.com/auth/verify ou seuapp://login devem abrir seu aplicativo ao invés do navegador.

Configuração no iOS

No arquivo ios/Runner/Info.plist, adicione a configuração de URL schemes:

```xml
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleTypeRole</key>
    <string>Editor</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>seuapp</string>
    </array>
  </dict>
</array>

<!-- Para Universal Links (HTTPS) -->
<key>com.apple.developer.associated-domains</key>
<array>
  <string>applinks:seuapp.com</string>
</array>
```

Para Universal Links funcionarem completamente no iOS, você também precisa hospedar um arquivo apple-app-site-association no seu domínio, mas isso está fora do escopo do aplicativo Flutter em si.

Processando Deep Links no Flutter

Agora que o sistema operacional sabe direcionar links para o aplicativo, precisamos processar esses links e navegar apropriadamente. Usaremos o pacote uni_links ou app_links para isto:

```yaml
dependencies:
  app_links: ^3.5.0
```

Então, criamos um service para gerenciar deep links:

```dart
```

```dart
import 'package:app_links/app_links.dart';

class DeepLinkService {
  final AppLinks _appLinks = AppLinks();

  /// Inicializa listeners de deep links
  void initialize(WidgetRef ref) {
    // Processa deep link que abriu o app (quando estava fechado)
    _handleInitialLink(ref);

    // Processa deep links recebidos enquanto app está aberto
    _handleIncomingLinks(ref);
  }

  /// Processa link inicial se app foi aberto por deep link
  Future<void> _handleInitialLink(WidgetRef ref) async {
    try {
      final uri = await _appLinks.getInitialLink();
      if (uri != null) {
        _processDeepLink(uri, ref);
      }
    } catch (e) {
      print('Erro ao processar link inicial: $e');
    }
  }

  /// Escuta deep links recebidos enquanto app está aberto
  void _handleIncomingLinks(WidgetRef ref) {
    _appLinks.uriLinkStream.listen(
      (uri) {
        _processDeepLink(uri, ref);
      },
      onError: (error) {
        print('Erro ao processar deep link: $error');
      },
    );
  }

  /// Processa o URI do deep link e navega apropriadamente
  void _processDeepLink(Uri uri, WidgetRef ref) {
    print('Deep link recebido: $uri');

    // Extrai path e parâmetros
    final path = uri.path;
    final queryParams = uri.queryParameters;
```

```
  // Roteamento baseado no path
  if (path.contains('/auth/verify')) {
    // Link de confirmação de email
    final token = queryParams['token'];
    if (token != null) {
      _handleEmailVerification(token, ref);
    }
  } else if (path.contains('/auth/reset-password')) {
    // Link de redefinição de senha
    final token = queryParams['token'];
    if (token != null) {
      _handlePasswordReset(token, ref);
    }
  } else if (path.contains('/auth/magic-link')) {
    // Magic link login
    final token = queryParams['token'];
    if (token != null) {
      _handleMagicLink(token, ref);
    }
  }
}

/// Processa confirmação de email
Future<void> _handleEmailVerification(String token, WidgetRef ref) async {
  try {
    // Chama endpoint de verificação
    await ref.read(authServiceProvider).verifyEmail(token);

    // Mostra mensagem de sucesso
    // (a navegação acontece automaticamente via AuthWrapper)
  } catch (e) {
    print('Erro ao verificar email: $e');
  }
}

/// Processa redefinição de senha
void _handlePasswordReset(String token, WidgetRef ref) {
  // Navega para tela de redefinir senha com token
  // Esta navegação precisa acontecer no contexto apropriado
  // Você pode armazenar o token e processar no build do AuthWrapper
}

/// Processa magic link
Future<void> _handleMagicLink(String token, WidgetRef ref) async {
  try {
    await ref.read(authServiceProvider).verifyMagicLink(token);
    // Estado de autenticação muda automaticamente
```

```
    } catch (e) {
      print('Erro ao processar magic link: $e');
    }
  }
}
```

Inicializamos este service no main.dart:

dart

```dart
void main() async {
  WidgetsFlutterBinding.ensureInitialized();

  // Inicializa Supabase ou outras dependências

  runApp(
    ProviderScope(
      child: MyApp(),
    ),
  );
}

class MyApp extends ConsumerStatefulWidget {
  @override
  ConsumerState<MyApp> createState() => _MyAppState();
}

class _MyAppState extends ConsumerState<MyApp> {
  final _deepLinkService = DeepLinkService();

  @override
  void initState() {
    super.initState();
    // Inicializa processamento de deep links
    _deepLinkService.initialize(ref);
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'App com Autenticação',
      theme: ThemeData(
        useMaterial3: true,
        colorSchemeSeed: Colors.blue,
      ),
      home: const AuthWrapper(),
    );
  }
}
```

Com esta configuração, quando um usuário clica em um link de confirmação de email, o aplicativo abre automaticamente e processa o token, verificando o email e fazendo login se apropriado. O AuthWrapper detecta a mudança de estado e navega para a área autenticada automaticamente. Do ponto de vista do usuário, ele clica no link e é levado diretamente para dentro do aplicativo já logado, uma experiência muito mais fluida que abrir o navegador e ter que abrir o app manualmente depois.

Proteção de Rotas na Área Autenticada

Dentro da área autenticada, você pode ter telas que requerem permissões específicas além de apenas estar autenticado. Por exemplo, uma tela de administração que só usuários com role de admin podem acessar. Vamos implementar um mecanismo simples de proteção de rotas baseado em roles:

dart

```dart
/// Widget que verifica permissões antes de renderizar conteúdo
class ProtectedRoute extends ConsumerWidget {
  final Widget child;
  final List<String>? requiredRoles;
  final Widget? fallback;

  const ProtectedRoute({
    Key? key,
    required this.child,
    this.requiredRoles,
    this.fallback,
  }) : super(key: key);

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final authState = ref.watch(authProvider);

    // Verifica se está autenticado
    if (authState is! AuthStateAuthenticated) {
      return fallback ?? const Center(
        child: Text('Você precisa estar autenticado'),
      );
    }

    // Se não há requisitos de role, permite acesso
    if (requiredRoles == null || requiredRoles!.isEmpty) {
      return child;
    }

    // Verifica se usuário tem alguma das roles necessárias
    final user = authState.user;
    final hasPermission = requiredRoles!.any(
      (role) => user.roles?.contains(role) ?? false,
    );

    if (!hasPermission) {
      return fallback ?? Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Icon(Icons.lock_outline, size: 64, color: Colors.grey),
            SizedBox(height: 16),
            Text(
              'Você não tem permissão para acessar esta tela',
              style: Theme.of(context).textTheme.titleMedium,
            ),
```

```
          ],
        ),
      );
    }

    return child;
  }
}

// Uso:
class AdminPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ProtectedRoute(
      requiredRoles: ['admin', 'moderator'],
      child: Scaffold(
        appBar: AppBar(title: Text('Painel Admin')),
        body: Center(
          child: Text('Conteúdo restrito para admins'),
        ),
      ),
    );
  }
}
```

Este padrão de ProtectedRoute pode ser expandido para verificações mais complexas de permissões, como verificar se o usuário tem acesso a recursos específicos ou se passou por verificação de dois fatores.

Conclusão da Seção

Nesta seção, construímos um sistema completo de navegação e proteção de rotas para aplicações com autenticação. Aprendemos como o Auth Wrapper orquestra navegação de alto nível baseado em estado, como implementar navegação manual para fluxos específicos com transições customizadas, como configurar e processar deep links para suportar fluxos que começam com emails, e como proteger rotas específicas baseado em permissões do usuário.

O resultado é uma experiência de navegação que parece natural e responsiva. Usuários são automaticamente direcionados para os lugares certos baseado em seu estado de autenticação, links de emails funcionam perfeitamente abrindo o aplicativo na tela apropriada, e conteúdo sensível é protegido contra acesso não autorizado. Toda esta complexidade é abstraída através de padrões claros que mantém o código manutenível e fácil de entender.

Na próxima seção, vamos explorar métodos alternativos de autenticação além do tradicional email e senha, incluindo autenticação passwordless com magic links, login social com Google e GitHub, e autenticação

biométrica usando impressão digital ou reconhecimento facial.

Fim da Seção 9: Navegação e Proteção de Rotas