

Gerenciamento de Estado para Autenticação

Desenvolvimento de Aplicativos Móveis - Flutter

Introdução

O estado de autenticação é único porque precisa ser acessível por toda a aplicação. Múltiplas telas precisam saber se o usuário está logado, quem é o usuário atual, e reagir automaticamente quando ele faz login ou logout. Este estado não pertence a uma tela específica, mas é compartilhado globalmente. Gerenciar este estado de forma eficaz é crucial para criar uma experiência fluida onde o aplicativo responde apropriadamente a mudanças de autenticação sem código duplicado ou lógica espalhada por múltiplos lugares.

Nesta seção, vamos construir um sistema completo de gerenciamento de estado para autenticação usando Riverpod. Escolhemos Riverpod porque oferece um excelente equilíbrio entre simplicidade e poder, tem integração natural com Flutter, e escala bem conforme a aplicação cresce. Porém, os conceitos que vamos aprender se aplicam a outras soluções de gerenciamento de estado como BLoC, MobX ou GetX.

O objetivo é criar um sistema onde o estado de autenticação vive em um único lugar, todas as telas podem escutá-lo e reagir automaticamente às mudanças, e operações como login e logout são métodos limpos que atualizam este estado centralizado. Quando implementado corretamente, isto resulta em uma aplicação onde mudanças de autenticação fluem naturalmente por toda interface sem necessidade de gerenciamento manual de callbacks ou atualizações.

Conceitos Fundamentais de Estado de Autenticação

Antes de mergulharmos no código, precisamos entender o que significa estado de autenticação e quais informações ele precisa conter. O estado de autenticação é essencialmente uma resposta para a pergunta: qual é a situação atual do usuário em relação ao login neste aplicativo?

Esta pergunta aparentemente simples tem várias respostas possíveis, e cada uma representa um estado distinto que nossa interface precisa saber lidar. O usuário pode estar em processo de autenticação, com uma requisição de login sendo processada no servidor. Pode estar autenticado com sucesso, com dados do usuário disponíveis. Pode não estar autenticado, exigindo que mostremos a tela de login. Pode ter ocorrido um erro durante tentativa de autenticação, exigindo que mostremos mensagem apropriada. E há também o estado inicial quando o aplicativo está verificando se existe uma sessão salva.

Cada um destes estados tem implicações sobre o que mostramos ao usuário. Se estamos verificando sessão inicial, mostramos uma splash screen. Se o usuário está autenticado, navegamos para a home do aplicativo. Se não está autenticado, mostramos login. Se está carregando, mostramos indicador de progresso. Se houve erro, mostramos mensagem de erro. A chave para gerenciar isto efetivamente é modelar estes estados explicitamente ao invés de usar múltiplas variáveis booleanas que podem criar estados impossíveis ou inconsistentes.

Modelando Estados com Classes

A melhor forma de modelar estados distintos em Dart é usando hierarquia de classes. Criamos uma classe base abstrata que representa o conceito geral de estado de autenticação, e então classes concretas para cada estado específico possível. Esta abordagem torna impossível criar estados inválidos e torna o código que consome o estado mais claro através de pattern matching.

Vamos começar definindo as classes que representam nosso domínio de autenticação:

```
dart

/// Representa um usuário autenticado no sistema
/// Esta é uma entidade pura do domínio, sem dependências de frameworks

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,
    });

    /// Construtor conveniente para criar cópia com modificações
    User copyWith({
        String? id,
        String? email,
        String? name,
        String? photoUrl,
    }) {
        return User(
            id: id ?? this.id,
            email: email ?? this.email,
            name: name ?? this.name,
            photoUrl: photoUrl ?? this.photoUrl,
        );
    }
}
```

A classe User representa o conceito de um usuário no domínio da aplicação. Note que ela é imutável, todas as propriedades são final e o construtor é const. Imutabilidade simplifica raciocínio sobre estado porque você sabe

que uma vez criado, um objeto User nunca muda. Se precisar de uma versão modificada, você cria uma nova instância usando copyWith.

Agora vamos modelar os diferentes estados que a autenticação pode estar:

dart

```
/// Classe base abstrata para todos os estados de autenticação
/// Usar classe abstrata permite polimorfismo e pattern matching
abstract class AuthState {
  const AuthState();
}

/// Estado inicial quando o app está verificando se há sessão salva
/// Este estado aparece apenas brevemente ao abrir o aplicativo
class AuthStateInitial extends AuthState {
  const AuthStateInitial();
}

/// Estado quando uma operação de autenticação está em andamento
/// Por exemplo, durante login, logout, ou registro
class AuthStateLoading extends AuthState {
  const AuthStateLoading();
}

/// Estado quando o usuário está autenticado com sucesso
/// Contém os dados do usuário atual
class AuthStateAuthenticated extends AuthState {
  final User user;

  const AuthStateAuthenticated(this.user);

  /// Operador de igualdade para comparações
  @override
  bool operator ==(Object other) {
    if (identical(this, other)) return true;
    return other is AuthStateAuthenticated && other.user.id == user.id;
  }

  /// Interface deve mostrar telas de login
  @override
  int hashCode => user.id.hashCode;
}

/// Estado quando o usuário não está autenticado
class AuthStateUnauthenticated extends AuthState {
  const AuthStateUnauthenticated();
}

/// Estado quando ocorreu um erro durante operação de autenticação
/// Contém mensagem de erro para exibir ao usuário
class AuthStateError extends AuthState {
  final String message;
```

```
const AuthStateError(this.message);

@Override
bool operator ==(Object other) {
  if (identical(this, other)) return true;
  return other is AuthStateError && other.message == message;
}

@Override
int get hashCode => message.hashCode;
}
```

Esta hierarquia de classes captura completamente todos os estados possíveis de autenticação. Cada classe é uma descrição semântica clara de uma situação específica. Quando você vê `AuthStateAuthenticated` no código, sabe imediatamente que o usuário está logado e tem acesso aos dados dele. Quando vê `AuthStateLoading`, sabe que uma operação está em andamento e deve mostrar indicador de progresso.

Implementamos operadores de igualdade em algumas classes para permitir comparações efetivas. Isto é especialmente útil quando queremos detectar se o estado realmente mudou ou se é o mesmo estado com os mesmos valores. Frameworks de gerenciamento de estado usam estas comparações para decidir se devem reconstruir widgets.

Criando o `AuthNotifier`

Agora que temos nossos estados modelados, precisamos de uma classe que gerencia transições entre estes estados e expõe métodos para operações de autenticação. No Riverpod, esta classe é um `StateNotifier`, que mantém um estado interno e notifica listeners quando ele muda.

Vamos construir nosso `AuthNotifier` passo a passo, começando com a estrutura básica:

```
dart
```

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

/// Gerencia o estado de autenticação e expõe métodos para login, logout, etc
/// StateNotifier<AuthState> significa que esta classe mantém um estado do tipo AuthState
class AuthNotifier extends StateNotifier<AuthState> {
    // Dependências que o notifier precisa para funcionar
    final AuthService _authService;
    final SecureStorageService _storageService;

    // Construtor inicializa com estado inicial e verifica sessão existente
    AuthNotifier({
        required AuthService authService,
        required SecureStorageService storageService,
    }) : _authService = authService,
        _storageService = storageService,
        super(const AuthStateInitial());
    // Assim que o notifier é criado, verifica se há sessão salva
    _checkAuthStatus();
}

/// Verifica se há uma sessão válida salva localmente
/// Chamado automaticamente quando o notifier é criado
Future<void> _checkAuthStatus() async {
    try {
        // Tenta obter tokens salvos
        final hasTokens = await _storageService.hasTokens();

        if (!hasTokens) {
            // Não há tokens, usuário não está autenticado
            state = const AuthStateUnauthenticated();
            return;
        }

        // Há tokens, tenta obter dados do usuário atual
        // Isto também valida que os tokens ainda são válidos
        final user = await _authService.getCurrentUser();

        if (user != null) {
            // Tokens são válidos e temos dados do usuário
            state = AuthStateAuthenticated(user);
        } else {
            // Tokens inválidos ou expirados, limpa storage
            await _storageService.clearAll();
            state = const AuthStateUnauthenticated();
        }
    } catch (e) {

```

```
// Erro ao verificar autenticação, assume não autenticado
print('Erro ao verificar status de autenticação: $e');
state = const AuthStateUnauthenticated();
}
}
}
```

Esta estrutura básica estabelece a fundação do nosso gerenciador de estado. O notifier recebe suas dependências através do construtor, seguindo o princípio de injeção de dependências que facilita testes. O super constructor recebe o estado inicial, que é AuthStateInitial enquanto estamos verificando se há sessão salva. O método checkAuthStatus é chamado automaticamente na criação e determina se devemos ir para estado autenticado ou não autenticado baseado na presença de tokens válidos.

Agora vamos adicionar os métodos para as principais operações de autenticação:

dart

```
/// Método para fazer login com email e senha
// Atualiza o estado para loading, faz a requisição, e atualiza para
// authenticated ou error baseado no resultado

Future<void> login({
    required String email,
    required String password,
}) async {
    // Muda estado para loading imediatamente para UI possa reagir
    state = const AuthStateLoading();

    try {
        // Chama o service que faz a requisição real ao servidor
        final user = await _authService.login(
            email: email,
            password: password,
        );

        // Login bem-sucedido, atualiza estado para authenticated
        state = AuthStateAuthenticated(user);

    } catch (e) {
        // Login falhou, atualiza estado para error com mensagem
        state = AuthStateError(_getErrorMessage(e));

        // Após mostrar erro por alguns segundos, volta para unauthenticated
        // permitindo que usuário tente novamente
        await Future.delayed(const Duration(seconds: 3));
        if (mounted) {
            state = const AuthStateUnauthenticated();
        }
    }
}

/// Método para registrar novo usuário
Future<void> register({
    required String email,
    required String password,
    String? name,
}) async {
    state = const AuthStateLoading();

    try {
        // Cria a conta
        await _authService.register(
            email: email,
            password: password,
        );
    } catch (e) {
        state = AuthStateError(_getErrorMessage(e));
    }
}
```

```
name: name,  
);  
  
// Após criar conta, dependendo da configuração, pode:  
// 1. Logar automaticamente o usuário  
// 2. Pedir confirmação de email antes de logar  
  
// Aqui vamos assumir que precisa confirmar email  
state = const AuthStateUnauthenticated();  
  
// Se quisesse logar automaticamente após registro:  
// await login(email: email, password: password);  
  
} catch (e) {  
  state = AuthStateError(_getErrorMessage(e));  
  await Future.delayed(const Duration(seconds: 3));  
  if (mounted) {  
    state = const AuthStateUnauthenticated();  
  }  
}  
}  
  
/// Método para fazer logout  
/// Limpa tokens locais e atualiza estado  
Future<void> logout() async {  
  try {  
    // Tenta revogar refresh token no servidor  
    await _authService.logout();  
  } catch (e) {  
    // Mesmo se revogação falhar, continua com logout local  
    print('Erro ao revogar token no servidor: $e');  
  }  
  
  // Limpa todos os dados de autenticação salvos localmente  
  await _storageService.clearAll();  
  
  // Atualiza estado para não autenticado  
  state = const AuthStateUnauthenticated();  
}  
  
/// Método para recuperação de senha  
Future<void> requestPasswordReset(String email) async {  
  state = const AuthStateLoading();  
  
  try {  
    await _authService.resetPassword(email);  
  } catch (e) {  
    state = AuthStateError(_getErrorMessage(e));  
  }  
}  
}
```

```

// Volta para unauthenticated após enviar email
// A UI pode mostrar mensagem de confirmação separadamente
state = const AuthStateUnauthenticated();

} catch (e) {
  state = AuthStateError(_getErrorMessage(e));
  await Future.delayed(const Duration(seconds: 3));
  if (mounted) {
    state = const AuthStateUnauthenticated();
  }
}

/// Converte exceções em mensagens amigáveis ao usuário
String _getErrorMessage(dynamic error) {
  if (error is AuthException) {
    return error.message;
  }

  // Para outros tipos de erro, retorna mensagem genérica
  return 'Ocorreu um erro. Por favor, tente novamente.';
}

```

Cada método segue o mesmo padrão: atualizar estado para loading, executar a operação, atualizar estado para authenticated ou error baseado no resultado. Este padrão consistente torna o código previsível e fácil de entender. Note como usamos o setter state do StateNotifier para atualizar o estado. Cada vez que atribuímos um novo valor a state, todos os listeners são notificados automaticamente e podem reconstruir suas interfaces.

O padrão de transição de erro para unauthenticated após alguns segundos é importante. Queremos mostrar a mensagem de erro tempo suficiente para o usuário ler, mas depois voltar ao estado que permite nova tentativa. Usamos mounted para verificar se o notifier ainda está ativo antes de mudar o estado, prevenindo erros se o notifier for disposed durante o delay.

Configurando o Provider

Para que nosso AuthNotifier seja acessível por toda aplicação, precisamos criar um provider do Riverpod. Providers são essencialmente fábricas que criam e gerenciam instâncias dos nossos objetos, permitindo que qualquer widget os acesse facilmente.

dart

```

import 'package:flutter_riverpod/flutter_riverpod.dart';

// Providers para as dependências do AuthNotifier
// Em aplicação real, estes viriam de outros arquivos
final authServiceProvider = Provider<AuthService>((ref) {
  return AuthService();
});

final storageServiceProvider = Provider<SecureStorageService>((ref) {
  return SecureStorageService();
});

/// Provider principal que expõe o AuthNotifier
/// StateNotifierProvider gerencia o ciclo de vida do notifier
/// e disponibiliza tanto o notifier quanto seu estado
final authProvider = StateNotifierProvider<AuthNotifier, AuthState>((ref) {
  // Obtém as dependências de outros providers
  final authService = ref.read(authServiceProvider);
  final storageService = ref.read(storageServiceProvider);

  // Cria e retorna instância do AuthNotifier
  return AuthNotifier(
    authService: authService,
    storageService: storageService,
  );
});

```

O StateNotifierProvider é especial porque expõe duas coisas. Primeiro, ele expõe o estado atual através de `ref.watch(authProvider)`, que retorna o `AuthState` atual. Segundo, ele expõe o notifier em si através de `ref.read(authProvider.notifier)`, que dá acesso aos métodos como `login` e `logout`. Esta separação é intencional: você usa `watch` para escutar mudanças de estado e reconstruir widgets, e usa `read` para chamar métodos sem causar reconstruções desnecessárias.

Para que providers funcionem, precisamos envolver nossa aplicação com `ProviderScope` no `main.dart`:

dart

```

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

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

  // Inicializar Supabase ou outras dependências aqui

  // ProviderScope torna todos os providers disponíveis na árvore de widgets
  runApp(
    const ProviderScope(
      child: MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

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

```

O ProviderScope na raiz da aplicação é essencial. Sem ele, nenhum provider funcionaria e você receberia erros ao tentar acessá-los. Pense no ProviderScope como o contentor que mantém todas as instâncias criadas pelos providers.

Consumindo o Estado nas Telas

Agora que temos nosso estado gerenciado centralmente, vamos aprender as diferentes formas de consumi-lo nas telas. O Riverpod oferece três métodos principais para acessar providers: watch, read e listen. Cada um tem propósito específico e usar o correto é importante para performance e correção.

Vamos começar com o padrão mais comum, usando ConsumerWidget para criar telas que reagem ao estado:

dart

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

/// Tela de login que reage automaticamente ao estado de autenticação
/// ConsumerWidget é como StatelessWidget mas com acesso a ref
class LoginPage extends ConsumerWidget {
  // Controllers para os campos de texto
  final _emailController = TextEditingController();
  final _passwordController = TextEditingController();

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // ref.watch escuta mudanças no estado e reconstrói quando muda
    final authState = ref.watch(authProvider);

    return Scaffold(
      appBar: AppBar(title: const Text('Login')),
      body: Padding(
        padding: const EdgeInsets.all(24),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            TextField(
              controller: _emailController,
              decoration: const InputDecoration(
                labelText: 'Email',
                prefixIcon: Icon(Icons.email_outlined),
              ),
              keyboardType: TextInputType.emailAddress,
            ),
            const SizedBox(height: 16),
            TextField(
              controller: _passwordController,
              decoration: const InputDecoration(
                labelText: 'Senha',
                prefixIcon: Icon(Icons.lock_outlined),
              ),
              obscureText: true,
            ),
            const SizedBox(height: 24),
            ElevatedButton(
              // Desabilita botão se está carregando
              onPressed: authState is AuthStateLoading
                ? null
                : () {
                    // ref.read acessa o notifier sem causar rebuild

```

```
ref.read(authProvider.notifier).login(  
    email: _emailController.text,  
    password: _passwordController.text,  
);  
,  
},  
child: authState is AuthStateLoading  
? const CircularProgressIndicator()  
: const Text('Entrar'),  
,  
],  
,  
,  
),  
);  
}  
}
```

Este exemplo mostra o uso básico de watch e read. Usamos ref.watch(authProvider) para obter o estado atual. Isto cria uma subscrição: quando o estado muda, o widget reconstrói automaticamente. No onPressed do botão, usamos ref.read(authProvider.notifier) para acessar os métodos do notifier. Usamos read aqui porque não queremos reconstruir quando o estado muda, apenas queremos chamar um método.

Checkamos o tipo do estado usando is para decidir o comportamento. Se é AuthStateLoading, mostramos indicador de progresso e desabilitamos o botão. Este pattern matching baseado em tipo é claro e seguro, o compilador garante que tratamos todos os casos.

Agora vamos ver um padrão mais avançado usando ref.listen para efeitos colaterais como navegação:

dart

```
class LoginPage extends ConsumerStatefulWidget {
  const LoginPage({Key? key}) : super(key: key);

  @override
  ConsumerState<LoginPage> createState() => _LoginPageState();
}

class _LoginPageState extends ConsumerState<LoginPage> {
  final _emailController = TextEditingController();
  final _passwordController = TextEditingController();

  @override
  void dispose() {
    _emailController.dispose();
    _passwordController.dispose();
    super.dispose();
  }

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

    // ref.listen executa efeitos colaterais quando estado muda
    // Não causa rebuild, apenas executa o callback
    ref.listen<AuthState>(authProvider, (previous, next) {
      // Quando muda para authenticated, navega para home
      if (next is AuthStateAuthenticated) {
        Navigator.of(context).pushReplacementNamed('/home');
      }

      // Quando muda para error, mostra SnackBar
      if (next is AuthStateError) {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: Text(next.message),
            backgroundColor: Colors.red,
          ),
        );
      }
    });

    return Scaffold(
      appBar: AppBar(title: const Text('Login')),
      body: Padding(
        padding: const EdgeInsets.all(24),
        child: Column(

```

```
mainAxisAlignment: MainAxisAlignment.center,  
children: [  
  TextField(  
    controller: _emailController,  
    decoration: const InputDecoration(  
      labelText: 'Email',  
      prefixIcon: Icon(Icons.email_outlined),  
    ),  
    keyboardType: TextInputType.emailAddress,  
    enabled: authState is! AuthStateLoading,  
  ),  
  const SizedBox(height: 16),  
  TextField(  
    controller: _passwordController,  
    decoration: const InputDecoration(  
      labelText: 'Senha',  
      prefixIcon: Icon(Icons.lock_outlined),  
    ),  
    obscureText: true,  
    enabled: authState is! AuthStateLoading,  
  ),  
  const SizedBox(height: 24),  
  SizedBox(  
    width: double.infinity,  
    height: 48,  
    child: ElevatedButton(  
      onPressed: authState is AuthStateLoading  
        ? null  
        : () => _handleLogin(),  
      child: authState is AuthStateLoading  
        ? const SizedBox(  
            height: 20,  
            width: 20,  
            child: CircularProgressIndicator(  
              strokeWidth: 2,  
              valueColor: AlwaysStoppedAnimation<Color>(  
                Colors.white,  
              ),  
            ),  
        )  
        : const Text('Entrar'),  
    ),  
  ),  
  const SizedBox(height: 16),  
  TextButton(  
    onPressed: () {  
      Navigator.of(context).pushNamed('/register');  
    },
```

```

    },
    child: const Text('Não tem conta? Cadastre-se'),
),
],
),
),
);
}
}

void _handleLogin() {
// Validação básica antes de tentar login
if (_emailController.text.trim().isEmpty) {
ScaffoldMessenger.of(context).showSnackBar(
const SnackBar(content: Text('Por favor, informe seu email')),
);
return;
}

if (_passwordController.text.isEmpty) {
ScaffoldMessenger.of(context).showSnackBar(
const SnackBar(content: Text('Por favor, informe sua senha')),
);
return;
}

// Chama método de login do notifier
ref.read(authProvider.notifier).login(
  email: _emailController.text.trim(),
  password: _passwordController.text,
);
}
}
}

```

O ref.listen é perfeito para efeitos colaterais como navegação, mostrar diálogos ou SnackBars. Ele não causa rebuild do widget, apenas executa o callback quando o estado muda. Recebemos tanto o estado anterior quanto o novo, permitindo reagir especificamente a transições. Por exemplo, podemos fazer algo apenas quando mudamos de loading para authenticated, não toda vez que estamos authenticated.

O Auth Wrapper: Navegação Baseada em Estado

Um padrão essencial em aplicações com autenticação é o auth wrapper, um widget que decide qual tela mostrar baseado no estado de autenticação atual. Isto centraliza a lógica de navegação condicional:

```

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

/// Widget que decide qual tela mostrar baseado no estado de autenticação
/// Este é tipicamente o widget raiz da aplicação
class AuthWrapper extends ConsumerWidget {
  const AuthWrapper({Key? key}) : super(key: key);

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

    // Pattern matching no estado para decidir o que mostrar
    return authState.when(
      // Durante verificação inicial, mostra splash
      initial: () => const SplashScreen(),
      // Durante operações de loading, mantém tela atual
      // (cada tela individual mostra seu próprio loading)
      loading: () => const CircularProgressIndicator(),
      // Quando autenticado, mostra área principal do app
      authenticated: (user) => const HomePage(),
      // Quando não autenticado, mostra tela de login
      unauthenticated: () => const LoginPage(),
      // Quando há erro, mostra na tela de login
      // (o próprio estado de erro já mostrou SnackBar)
      error: (message) => const LoginPage(),
    );
  }
}

```

Espere, o código acima não compilaria porque AuthState não tem método when. Esse é um padrão que você pode adicionar com freezed ou implementar manualmente. Vamos ver a versão sem when, usando pattern matching tradicional:

dart

```

class AuthWrapper extends ConsumerWidget {
  const AuthWrapper({Key? key}) : super(key: key);

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

    // Pattern matching tradicional com if/else
    if (authState is AuthStateInitial) {
      return const SplashScreen();
    }

    if (authState is AuthStateAuthenticated) {
      return HomePage(user: authState.user);
    }

    // Para todos os outros estados (Unauthenticated, Loading, Error)
    // mostra a tela de login, que lida com seus próprios estados
    return const LoginPage();
  }
}

```

Este wrapper é colocado como widget inicial da aplicação. Ele automaticamente navega entre splash, login e home baseado no estado. Não há lógica de navegação manual com push ou pop, apenas renderização condicional que muda conforme o estado. Isto é extremamente poderoso porque garante que a tela correta está sempre sendo mostrada para o estado atual, sem possibilidade de inconsistência.

Persistência de Estado Entre Sessões

Um aspecto crucial de autenticação mobile é que usuários esperam permanecer logados entre usos do aplicativo. Já implementamos o armazenamento seguro de tokens e a verificação de sessão no método checkAuthStatus do AuthNotifier. Vamos revisar como isto funciona em conjunto:

Quando o AuthNotifier é criado, ele automaticamente chama checkAuthStatus. Este método verifica se há tokens salvos no secure storage. Se há, tenta obter dados do usuário atual, o que implicitamente valida os tokens. Se bem-sucedido, muda estado para AuthStateAuthenticated. Se os tokens são inválidos ou expirados, limpa o storage e muda para AuthStateUnauthenticated.

Este fluxo acontece rapidamente, geralmente em menos de um segundo. Durante esta verificação, o AuthWrapper mostra a splash screen. Assim que o estado muda para authenticated ou unauthenticated, o wrapper automaticamente mostra a tela apropriada. O usuário vê a splash brevemente e depois é levado diretamente para home se estava logado, ou para login se não estava.

A renovação automática de tokens também integra naturalmente com este sistema. Se implementamos um interceptor que detecta quando access token expira e automaticamente renova usando refresh token, isto é completamente transparente para o gerenciamento de estado. O AuthNotifier mantém estado authenticated durante toda a renovação, a operação continua normalmente, e o usuário nunca percebe que tokens foram renovados nos bastidores.

Listeners Globais para Logging e Analytics

Uma vantagem de gerenciamento de estado centralizado é que podemos observar todas as mudanças de estado em um único lugar, útil para logging, analytics ou debugging. O Riverpod permite criar observers globais:

dart

```

import 'package:flutter_riverpod/flutter_riverpod.dart';

/// Observer global que loga todas as mudanças de estado
class AuthStateLogger extends ProviderObserver {
  @override
  void didUpdateProvider(
    ProviderBase provider,
    Object? previousValue,
    Object? newValue,
    ProviderContainer container,
  ) {
    // Só loga mudanças no authProvider
    if (provider == authProvider) {
      print('Auth state changed: $previousValue -> $newValue');

      // Aqui você poderia enviar eventos para analytics
      // if (newValue is AuthStateAuthenticated) {
      //   analytics.logLogin();
      // } else if (previousValue is AuthStateAuthenticated) {
      //   analytics.logLogout();
      // }
    }
  }
}

// No main.dart, registra o observer
void main() {
  runApp(
    ProviderScope(
      observers: [AuthStateLogger()],
      child: const MyApp(),
    ),
  );
}

```

Este padrão permite monitorar todas as transições de estado sem poluir a lógica de negócio do AuthNotifier. Você pode ter múltiplos observers, cada um com responsabilidade diferente como logging, analytics, ou crash reporting.

Conclusão da Seção

Nesta seção, construímos um sistema completo de gerenciamento de estado para autenticação usando Riverpod. Aprendemos a modelar estados explicitamente com hierarquia de classes, criar um StateNotifier que gerencia

transições entre estados e expõe métodos para operações de autenticação, configurar providers que tornam tudo acessível globalmente, e consumir estado nas telas usando watch, read e listen apropriadamente.

O resultado é uma arquitetura onde o estado de autenticação vive em um único lugar, todas as telas reagem automaticamente às mudanças, e a lógica de autenticação está centralizada e testável. Quando o usuário faz login, o estado muda e toda a interface se atualiza automaticamente. Quando faz logout, o mesmo acontece de forma coordenada sem necessidade de gerenciamento manual.

Na próxima seção, vamos focar na integração com APIs, implementando os services que o AuthNotifier usa para comunicação com o backend, incluindo tratamento robusto de erros de rede e renovação automática de tokens.

Fim da Seção 7: Gerenciamento de Estado para Autenticação