Exemplo Prático: Simulador de Status Online

Vamos construir um aplicativo simples que mostra uma lista de usuários e seu status online. O desafio aqui é garantir que a tela seja atualizada automaticamente sempre que um usuário muda de status, sem a necessidade de reconstruir a tela inteira.

Nós vamos refatorar o mesmo código, passo a passo, mostrando a evolução de setState() para Provider e, finalmente, para Riverpod.

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

// Definindo o modelo de dados para o nosso usuário.
class User {
  final String id;
  final String name;
  bool isOnline;

  User({required this.id, required this.name, this.isOnline = false});
}

//
============================================================
===========
// PARTE 1: GERENCIAMENTO DE ESTADO COM PROVIDER
//
============================================================
===========

// Esta classe é a nossa ViewModel (ou State Manager) para o Provider.
// Ela estende ChangeNotifier, um "emissor" de notificação de mudanças.
class OnlineUsersNotifier extends ChangeNotifier {
  // Lista privada de usuários. O underscore `_` indica que é privada.
  final List<User> _users = [
    User(id: '1', name: 'Alice', isOnline: true),
    User(id: '2', name: 'Bob'),
    User(id: '3', name: 'Charlie', isOnline: true),
    User(id: '4', name: 'Diana'),
  ];

  // Getter para expor a lista de forma segura (sem permitir modificações diretas).
  List<User> get users => _users;

  // Método para alternar o status online de um usuário.
  void toggleStatus(String userId) {
    // Procura o usuário na lista pelo ID.
    final user = _users.firstWhere((user) => user.id == userId);
```

```
    // Inverte o status de online.
    user.isOnline = !user.isOnline;
    // Notifica todos os "ouvintes" (widgets) que o estado mudou.
    notifyListeners();
  }
}

// O provider que irá fornecer a nossa ViewModel (OnlineUsersNotifier) para a árvore de
widgets.
// Ele é colocado no topo da árvore.
final onlineUsersProvider = ChangeNotifierProvider((ref) => OnlineUsersNotifier());

//
==========================================================================
===========
// PARTE 2: GERENCIAMENTO DE ESTADO COM RIVERPOD
//
==========================================================================
===========

// Para o Riverpod, usamos StateNotifier e StateNotifierProvider.
// StateNotifier é uma versão mais robusta e testável do ChangeNotifier.
class OnlineUsersStateNotifier extends StateNotifier<List<User>> {
  // O construtor inicializa o estado com a lista de usuários.
  OnlineUsersStateNotifier() : super([
    User(id: '1', name: 'Alice', isOnline: true),
    User(id: '2', name: 'Bob'),
    User(id: '3', name: 'Charlie', isOnline: true),
    User(id: '4', name: 'Diana'),
  ]);

  // Método para alternar o status.
  void toggleStatus(String userId) {
    // O estado no StateNotifier é imutável.
    // Primeiro, criamos uma nova lista a partir do estado atual.
    final newUsers = [...state];
    // Encontramos o índice do usuário a ser modificado.
    final userIndex = newUsers.indexWhere((user) => user.id == userId);
    // Invertemos o status do usuário na nova lista.
    newUsers[userIndex].isOnline = !newUsers[userIndex].isOnline;
    // Atribuímos a nova lista ao estado. Isso dispara a atualização do estado.
    state = newUsers;
  }
}

// O provider do Riverpod. Ele irá fornecer a instância do nosso StateNotifier.
// O tipo `List<User>` é a forma de indicar o tipo do estado que será compartilhado.
final onlineUsersNotifierProvider = StateNotifierProvider<OnlineUsersStateNotifier,
List<User>>(
  (ref) => OnlineUsersStateNotifier(),
```

```
);

//
================================================================
==========
// UI DO APLICATIVO (WIDGETS)
//
================================================================
==========

// Widget principal que contém toda a nossa aplicação.
void main() {
  runApp(
    // Envolve a aplicação com ProviderScope para usar Riverpod.
    // É uma boa prática usar ProviderScope mesmo que você use Providers.
    const ProviderScope(
      child: MyApp(),
    ),
  );
}

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Status Online',
      theme: ThemeData(primarySwatch: Colors.indigo),
      home: const HomeScreen(),
    );
  }
}

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Simulador de Status Online'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: const [
            Text(
              'Usando Provider e Riverpod',
              style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
```

```
            ),
            SizedBox(height: 20),
            // Widget que usa o Provider.
            Expanded(child: ProviderView()),
            SizedBox(height: 20),
            // Widget que usa o Riverpod.
            Expanded(child: RiverpodView()),
          ],
        ),
      ),
    );
  }
}

//
// ============================================================================
// WIDGETS DE VISUALIZAÇÃO
//
// ============================================================================

// Widget que consome o estado usando o pacote `provider`.
class ProviderView extends StatelessWidget {
  const ProviderView({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    // Usamos o `Consumer` para "escutar" o Provider.
    // Apenas este widget e seus filhos serão reconstruídos em caso de mudança.
    return Consumer<OnlineUsersNotifier>(
      builder: (context, onlineUsers, child) {
        // Exibe uma lista de usuários.
        return Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            const Text(
              'Lista de Usuários (Provider)',
              style: TextStyle(fontSize: 16, fontWeight: FontWeight.bold),
            ),
            const SizedBox(height: 10),
            Expanded(
              child: ListView.builder(
                itemCount: onlineUsers.users.length,
                itemBuilder: (context, index) {
                  final user = onlineUsers.users[index];
                  return UserTile(
                    user: user,
                    // Ao tocar no tile, chamamos o método do nosso Notifier.
                    onTap: () => onlineUsers.toggleStatus(user.id),
```

```
              );
            },
          ),
        ),
      ],
    );
  },
 );
 }
}

// Widget que consome o estado usando o pacote `flutter_riverpod`.
class RiverpodView extends ConsumerWidget {
 const RiverpodView({Key? key}) : super(key: key);

 // O Riverpod usa `ref` para interagir com os providers.
 @override
 Widget build(BuildContext context, WidgetRef ref) {
  // Usamos `ref.watch` para "escutar" o provider.
  // Isso fará com que o widget seja reconstruído quando o estado mudar.
  final users = ref.watch(onlineUsersNotifierProvider);

  return Column(
   crossAxisAlignment: CrossAxisAlignment.start,
   children: [
    const Text(
     'Lista de Usuários (Riverpod)',
     style: TextStyle(fontSize: 16, fontWeight: FontWeight.bold),
    ),
    const SizedBox(height: 10),
    Expanded(
     child: ListView.builder(
      itemCount: users.length,
      itemBuilder: (context, index) {
       final user = users[index];
       return UserTile(
        user: user,
        // Chamamos o método do notifier através de `ref.read`.
        // Usamos `read` aqui pois não precisamos que o widget seja reconstruído
        // apenas por causa do método.
        onTap:                                    ()                                    =>
ref.read(onlineUsersNotifierProvider.notifier).toggleStatus(user.id),
       );
      },
     ),
    ),
   ],
  );
 }
}
```

```
// Widget reutilizável para exibir cada usuário na lista.
class UserTile extends StatelessWidget {
  final User user;
  final VoidCallback onTap;

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

  @override
  Widget build(BuildContext context) {
    return Card(
      child: ListTile(
        title: Text(user.name),
        subtitle: Text(user.isOnline ? 'Online' : 'Offline'),
        leading: CircleAvatar(
          backgroundColor: user.isOnline ? Colors.green : Colors.grey,
        ),
        onTap: onTap,
      ),
    );
  }
}
```

Desafio da Metodologia Ativa

Estudo de Caso: Adicionando uma Busca de Usuários

Um dos requisitos da equipe de design é que o aplicativo permita que o usuário filtre a lista de online para encontrar uma pessoa específica. A equipe de backend nos informou que o filtro será feito no front-end para evitar requisições desnecessárias.

Sua missão, se a aceitar:

1. **Análise:** Onde a lógica de busca (filtro) deveria ser implementada? Você precisa de um novo Notifier ou um novo Provider para o termo de busca?
2. **Refatore:** Modifique a camada do **Riverpod** para incluir um TextField no topo da lista. O usuário deve digitar um nome e a lista de usuários exibida deve ser atualizada em tempo real para mostrar apenas os usuários cujo nome contém o texto digitado.
3. **Adicione um Teste:** Crie um teste unitário para o OnlineUsersStateNotifier que simule a busca e verifique se a lista filtrada retorna os resultados corretos.

Estrutura de Pastas

O padrão MVVM é ideal para projetos que precisam de uma separação clara entre a interface do usuário (View) e a lógica de negócio. Veja como ele se aplica ao nosso projeto:

- **lib/views/**: Esta pasta contém os **Widgets (Views)** que são responsáveis apenas por construir a interface do usuário (UI). Eles recebem os dados do ViewModel e não contêm nenhuma lógica de negócio.
- **lib/viewmodels/**: Esta é a camada do **ViewModel** (que no Riverpod é nosso StateNotifier ou Notifier). Ele gerencia o estado e a lógica de negócio. A View "assiste" a essa camada para saber quando precisa ser reconstruída.
- **lib/models/**: Aqui estão os **Models**, que são classes de dados simples que representam as entidades da nossa aplicação, como User. Eles não contêm lógica de negócio.
- **lib/repositories/**: Esta pasta é uma camada adicional que adiciona o padrão **Repositório**. Ela abstrai a origem dos dados (seja uma API, um banco de dados local, etc.). O ViewModel interage com o Repository, e não diretamente com a fonte de dados, o que torna o código mais flexível e fácil de testar.

- **lib/main.dart**: O ponto de entrada da aplicação, onde os providers e a configuração inicial são definidos.

Essa organização deixa claro que a **lógica** fica no viewmodels e a **aparência** fica nas views. É uma forma poderosa de manter o código limpo, testável e fácil de manter em projetos maiores.