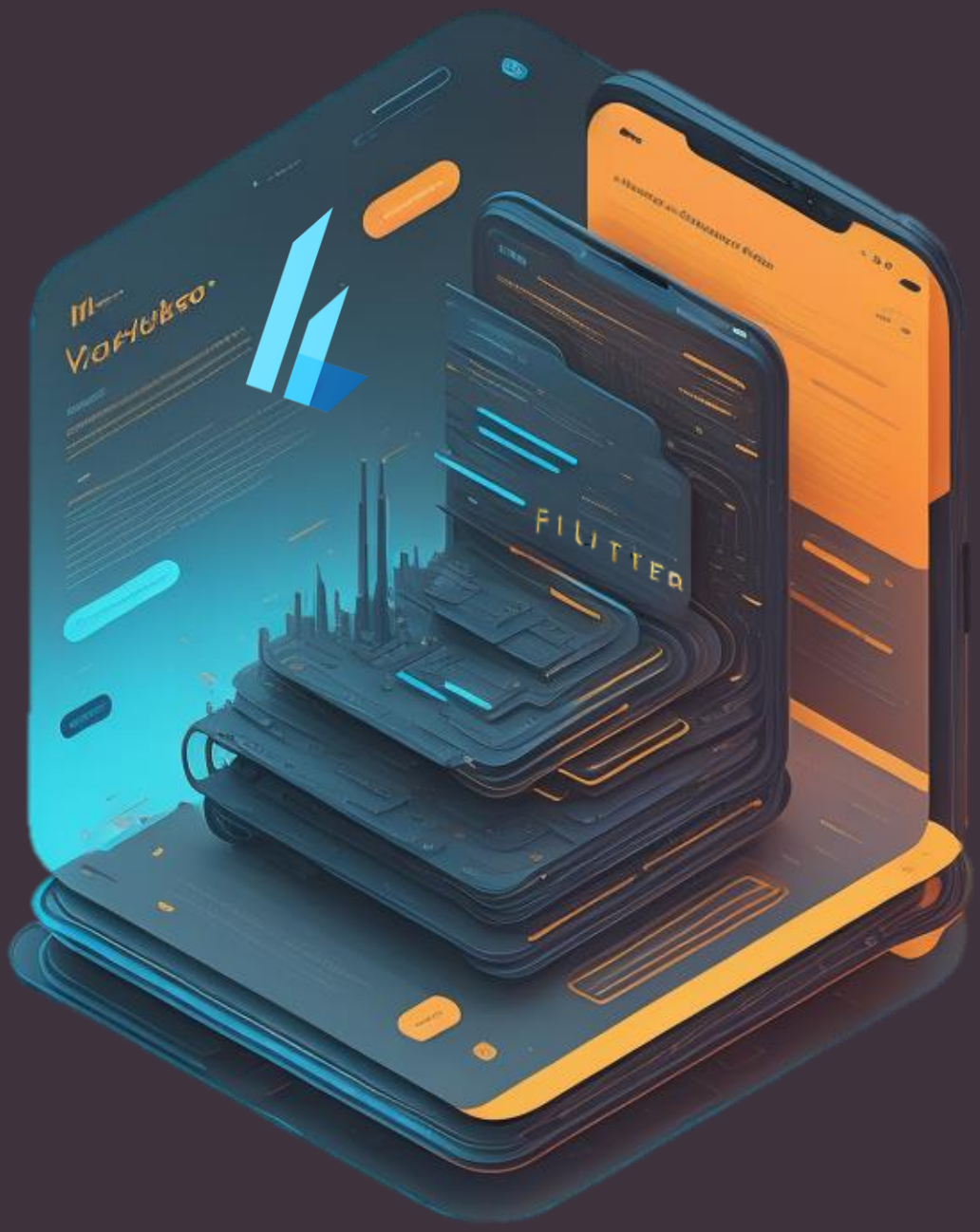
Desbravando o Flutter

Uma Jornada rumo à eficiência com Cubit



Eduardo Comparin

1 Introdução ao Flutter e Cubit

1.1 Uma Visão Geral

Flutter é um framework open-source desenvolvido pela Google para criar interfaces de usuário nativas de alta qualidade, utilizando uma única base de código. Sua linguagem principal é o Dart.

Com o Flutter, é possível criar aplicativos para diversas plataformas, como iOS, Android, Web e Desktop, proporcionando uma experiência consistente. Além disso, a biblioteca de widgets rica e a performance nativa são pontos fortes.

1.2 Gerenciamento de Estado em Flutter

À medida que aplicativos crescem em complexidade, surge a necessidade crucial de gerenciar o estado de forma eficiente para evitar bugs e garantir uma experiência fluida ao usuário.

Abordagens tradicionais, como o uso de variáveis globais, podem levar a códigos difíceis de manter. A abordagem reativa, por outro lado, oferece soluções mais estruturadas usando streams e blocos.

1.3 Cubit: Gerenciamento de Estado

Cubit é uma biblioteca para gerenciamento de estado no Flutter, originada do O Bloc (Business Logic Component) proporcionando uma maneira fácil e eficaz de lidar com a complexidade do estado da aplicação. Ele segue o princípio de reatividade, permitindo atualizações automáticas da interface com base nas mudanças de estado.

No Cubit, o estado da aplicação é representado por uma classe imutável.

A comunicação entre componentes ocorre por meio de emissão de eventos e escuta de alterações de estado. Isso promove uma arquitetura limpa e modular

2 Fundamentos do Cubit

O Cubit é uma poderosa biblioteca de gerenciamento de estado para aplicações Flutter, simplificando o controle de estados complexos. Neste capítulo, exploraremos os fundamentos essenciais do Cubit para que você possa começar a utilizá-lo em seus projetos.

2.1 Instalação e Configuração do Cubit

Antes de mergulhar no mundo do Cubit, é crucial configurar e instalar a biblioteca corretamente. Aprenda a adicionar a dependência Cubit ao seu projeto Flutter e realizar as configurações iniciais necessárias para garantir uma integração suave.

```dart
// Criando uma parta cubit com um arquivo chamado counter_cubit.dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);
}
```

Após adicionar a dependência, é crucial realizar a configuração inicial do Cubit. Este processo envolve a configuração de parâmetros fundamentais para garantir a correta inicialização da biblioteca no seu projeto.

2.2 Criando o Primeiro Cubit

Agora que seu ambiente está configurado, é hora de criar seu primeiro Cubit. Entenda a estrutura básica do Cubit e como ele organiza o estado da sua aplicação.

Um Cubit consiste em três partes principais: Estado, Cubit e Evento. Vejamos um exemplo simples:

```dart
// Criando o arquivo do state
class CounterState extends Equatable{
  const CounterState({
      this.count = 0
    });

  final int count;

  CounterState copyWith({
    int count,
  }){
    return CounterState(
      count: count ?? this.count,
      );
  }
}

// Criando o arquivo do Cubit
class CounterCubit extends Cubit<CounterState> {

  CounterCubit() : super(const CounterState());

  //Evento
  Future<void> IncrementEvent() async{
    emit(state.copyWith(count: state.count + 1))
  }

}
```

A classe CounterState representa o estado inicial do Cubit. Aqui, definimos um estado inicial com um contador igual a zero. O CounterCubit é a parte central, gerenciando o estado, enquanto o IncrementEvent é um evento que altera esse estado.

Agora, ao chamar increment(), o Cubit emite um novo estado com o contador incrementado. Isso demonstra como lidamos eficientemente com transições de estado, mantendo nossa aplicação reativa e fácil de gerenciar.

Exploramos aqui os fundamentos do Cubit, desde a instalação até a criação de um Cubit funcional. Compreender esses conceitos é crucial para aproveitar ao máximo o poder do Cubit no desenvolvimento de aplicações Flutter robustas e escaláveis.

3 Utilizando Cubit na Prática

O Cubit é uma ferramenta poderosa para o gerenciamento de estados em aplicações Flutter. Neste capítulo, vamos mergulhar na prática, explorando como integrar o BlocC Cubit em seus widgets e aprimorar a reatividade da sua interface.

3.1 Conectando Cubit aos Widgets

Agora, conectamos os Cubits aos nossos widgets para gerenciar o estado de forma eficiente. Veja como integrar um Cubit em um widget:

```dart
class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final cubit = context.read<MyCubit>();
    return BlocBuilder<MyCubit, MyState>(
      builder: (context, state) {
        return Text("Olá Mundo");
        // Construa seu widget com base no estado do Cubit
      },
    );
  }
}
```

O BlocBuilder é usado para reagir a mudanças de estado e reconstruir a UI automaticamente. Quando um dos botões é pressionado, um evento correspondente é despachado para o Cubit, que então emite um novo estado, acionando a reconstrução da UI pelo BlocBuilder. Explicando o código:

MyWidget extends StatelessWidget: Esta classe MyWidget é uma widget do Flutter que extende StatelessWidget, o que significa que não tem estado interno mutável.

build(BuildContext context): Este é o método obrigatório em qualquer widget Flutter. Ele é chamado para construir a representação visual do widget.

final cubit = context.read<MyCubit>(): Está sendo usado o context.read para obter uma instância de MyCubit. O context é uma referência ao contexto do widget no Flutter, que fornece acesso a vários recursos, como o tema, o roteador, e, neste caso, o Cubit.

return BlocBuilder<MyCubit, MyState>(builder: (context, state) {},): Este é o ponto principal do código. BlocBuilder é um widget fornecido pelo Bloc que reconstrói automaticamente seu filho (o widget dentro do callback builder) quando o estado do MyCubit muda.

MyCubit*:* é o cubit que está sendo observado. **MyState** é o tipo de estado que o MyCubit gerencia.

Builder: é uma função que será chamada sempre que o estado do MyCubit for atualizado. Ela recebe o context atual e o novo state como argumentos.

Dentro do builder, você deve construir o seu widget com base no estado atual do Cubit. Isso geralmente envolve a renderização condicional de diferentes partes do UI dependendo do estado.

3.2 Otimizando a Reatividade

Para otimizar a reatividade no código fornecido, você pode considerar algumas práticas e ajustes. Aqui estão algumas sugestões:

Recriações Desnecessárias de Widgets e Cubit

Dentro do método build, evite criar instâncias de objetos ou realizar cálculos complexos que não dependem do estado do Cubit. Isso pode levar a recriações desnecessárias de widgets.

Se o Cubit for imutável, você pode extrair a instância fora do método build para evitar recriações desnecessárias a cada reconstrução do widget.

```dart
@override
Widget build(BuildContext context) {
  final cubit = context.read<MyCubit>();
  // Evite cálculos complexos ou criações de objetos desnecessárias aqui
  return BlocBuilder<MyCubit, MyState>(
    builder: (context, state) {
      // Construa seu widget com base no estado do Cubit
    },
  );
}
```

Use const Quando Possível

Utilize o modificador const sempre que possível para criar widgets que não dependem de variáveis mutáveis. Isso pode ajudar a otimizar a performance, reduzindo a necessidade de reconstrução do widget.

Lembre-se de que, ao otimizar a reatividade, é importante equilibrar a simplicidade do código com a performance.

```dart
@override
Widget build(BuildContext context) {
  final cubit = context.read<MyCubit>();
  // Use const para widgets que não dependem de variáveis mutáveis
  return BlocBuilder<MyCubit, MyState>(
    builder: (context, state) {
      return const MyImmutableWidget();
    },
  );
}
```

3.3 Tratando Erros e Exceções

No seu BlocBuilder, você pode usar um BlocListener para lidar especificamente com estados de erro. Isso permite que você reaja de maneira específica a diferentes tipos de erros e forneça feedback ao usuário.

BlocListener para Tratar Estados de Erro

No seu BlocBuilder, você pode usar um BlocListener para lidar especificamente com estados de erro. Isso permite que você reaja de maneira específica a diferentes tipos de erros e forneça feedback ao usuário.

```
return BlocListener<MyCubit, MyState>(
  listener: (context, state) {
    if (state is MyErrorState) {
      // Lidar com o erro, exibir uma mensagem ao usuário, fazer log, etc.
      Scaffold.of(context).showSnackBar(SnackBar(
        content: Text('Ocorreu um erro: ${state.errorMessage}'),
      ));
    }
  },
  child: BlocBuilder<MyCubit, MyState>(
    builder: (context, state) {
      // Construa seu widget com base no estado do Cubit
    },
  ),
);
```

Tratamento Global de Erros

Utilize um tratamento global de erros para capturar exceções não tratadas. Você pode fazer isso usando FlutterError.onError.

```
void main() {
  FlutterError.onError = (FlutterErrorDetails details) {
    // Lidar com o erro global, por exemplo, enviar logs para o servidor.
    print('Erro global: ${details.exception}');
  };
  runApp(MyApp());
}
```

Exibição de Mensagens de Erro ao Usuário

Sempre forneça feedback claro ao usuário quando ocorrer um erro. Isso pode incluir o uso de SnackBars, Dialogs ou outras formas de apresentar mensagens de erro.

```
Scaffold.of(context).showSnackBar(SnackBar(
  content: Text('Ocorreu um erro: ${errorMessage}'),
));
```

Tratamento de Erros Assíncronos:

Ao lidar com chamadas assíncronas, use try-catch para capturar erros dentro das funções assíncronas.

```
Future<void> fetchData() async {
  try {
    // Código que pode lançar exceções
    await api.fetchData();
  } catch (e) {
    print('Erro ao buscar dados: $e');
  }
}
```

Lembre-se de ajustar essas estratégias de acordo com as necessidades específicas do seu aplicativo e o tipo de erro que você está tratando. É fundamental testar seu código em cenários de erro durante o desenvolvimento.

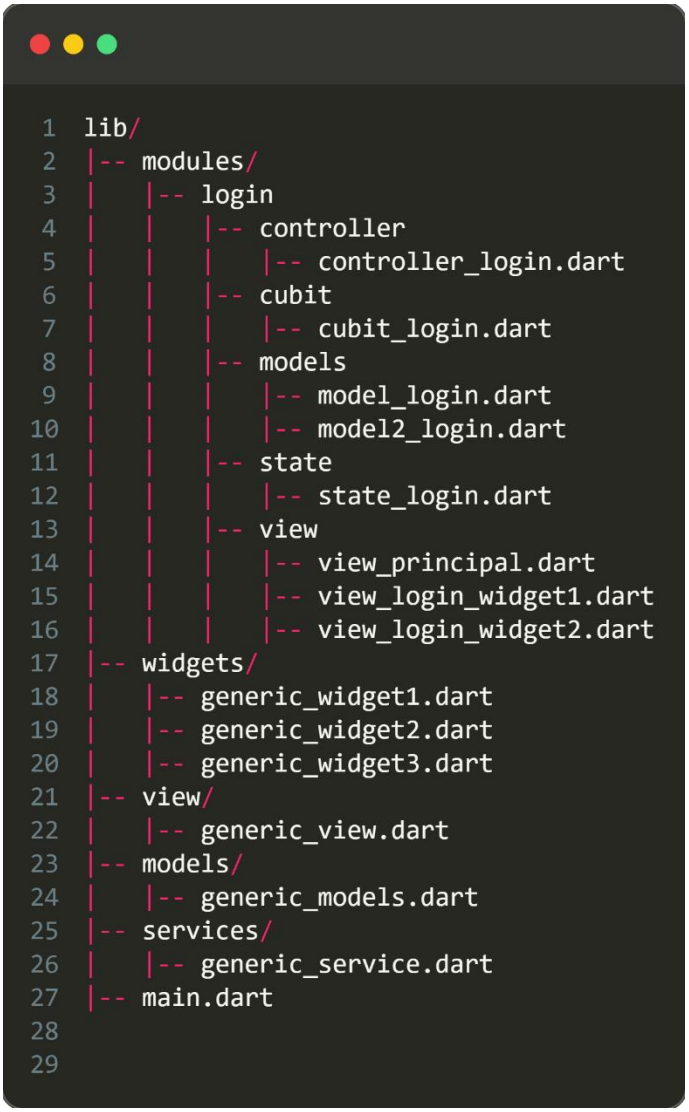
Ao seguir esses passos práticos, você estará preparado para utilizar o BlocC de forma eficiente, integrando o Cubit em seus widgets e proporcionando uma experiência de usuário reativa e robusta em suas aplicações Flutter.

4 Boas Práticas e Dicas

4.1 Estrutura e Organização de Código

Ao desenvolver em Flutter, a estrutura de diretórios bem organizada facilita a manutenção do código. Separe seus arquivos por funcionalidade, como View, models, Controler, Cubit e State.

Existem varias formas de organizar o projeto. Uma boa maneira de fazer essa organização é em módulos onde irá concentrar todos arquivos de terminada parte do seu projeto. Por exemplo “login”:



```
lib/
|-- modules/
|   |-- login
|   |   |-- controller
|   |   |   |-- controller_login.dart
|   |   |-- cubit
|   |   |   |-- cubit_login.dart
|   |   |-- models
|   |   |   |-- model_login.dart
|   |   |   |-- model2_login.dart
|   |   |-- state
|   |   |   |-- state_login.dart
|   |   |-- view
|   |   |   |-- view_principal.dart
|   |   |   |-- view_login_widget1.dart
|   |   |   |-- view_login_widget2.dart
|-- widgets/
|   |-- generic_widget1.dart
|   |-- generic_widget2.dart
|   |-- generic_widget3.dart
|-- view/
|   |-- generic_view.dart
|-- models/
|   |-- generic_models.dart
|-- services/
|   |-- generic_service.dart
|-- main.dart
```

A estrutura segue um padrão modular, organizando os diferentes aspectos da sua aplicação em diretórios específicos.

Modules: Cada módulo é responsável por uma parte separada da funcionalidade do aplicativo. No exemplo, há um módulo chamado "login“, este é um módulo específico, pela funcionalidade de login.

Controller: contém a lógica de controle para a funcionalidade de login. O controller e concentrado as chamas das APIs com suas logicas para transformar e tratar o retorno

Cubit: Contém o Cubit relacionado à funcionalidade de login, concentrar a logica e manipulação dos dados sem a necessidade de varias requisições a API. (Se a regra de negocio permitir) .

Models: Aqui estão os modelos relacionados à funcionalidade de login.

State: Contém classes de estado relacionadas ao módulo de login. Onde controla a atualização dos dados e das view.

View: Contém arquivos relacionados à interface do usuário para a funcionalidade de login. view_principal.dart pode ser a tela principal do módulo de login, e view_login_widget1.dart e view_login_widget2.dart podem ser widgets específicos para a tela de login criando uma separação nas próprias view com a reutilização dos widgets genéricos.

Widgets: Aqui contém widgets reutilizáveis que podem ser utilizados em diferentes partes da aplicação. chamo em varias telas conseguindo nos dar uma estrutura fácil e pratica para manutenção do código.

Essa estrutura modular é uma prática comum em desenvolvimento Flutter para facilitar a escalabilidade, manutenção e reutilização de código. Certifique-se de adaptar essa estrutura de acordo com as necessidades específicas do seu projeto.

4.2 Práticas para Organização de Cubits

Quando trabalhamos com Cubits no Flutter, agrupar Cubits relacionados em diretórios específicos torna o código mais organizado e fácil de manter. Manter a lógica de negócios (Cubit) separada da interface gráfica (Widget) facilita a compreensão e a reutilização do código.

BlocC com Cubit é uma escolha eficaz para gerenciamento de estado global no Flutter. Centralize o estado do seu aplicativo para uma gestão mais eficiente.

Utilizar um Cubit dedicado para o estado global do aplicativo simplifica o gerenciamento do estado em toda a aplicação.

```dart
class AppCubit extends Cubit<AppState> {
  // Implementação do Cubit para gerenciar o estado global do aplicativo
}

void main() {
  runApp(
    BlocProvider(
      create: (context) => AppCubit(),
      child: MyApp(),
    ),
  );
}
```

Cubit é uma biblioteca para gerenciamento de estado no Flutter, originada do O Bloc (Business Logic Component) proporcionando uma maneira fácil e eficaz de lidar com a complexidade do estado da aplicação. Ele segue o princípio de reatividade, permitindo atualizações automáticas da interface com base nas mudanças de estado.

4.3 Evitando Anti-padrões

Evite anti-padrões, como o uso excessivo de estado global. Utilize Cubits com parcimônia para evitar compartilhar mais estado do que o necessário.

```dart
// Evite isso: Uso excessivo de estado global
class GlobalState {
  static String userName = '';
}

// Prefira isso: Gerenciamento de estado com Cubit
class UserCubit extends Cubit<String> {
  UserCubit() : super('');

  void setUserName(String name) => emit(name);
}
```

4.4 Teste

A escrita de testes unitários é essencial para garantir que seus Cubits funcionem corretamente. Teste de forma isolada e, em seguida, integre com widgets para uma garantia completa de funcionamento.

```dart
// Teste Unitário
void main() {
  test('Incrementa o contador', () {
    final cubit = CounterCubit();
    cubit.increment();
    expect(cubit.state, 1);
  });
}
```

Ao usar Flutter, especialmente com a abordagem de gerenciamento de estado BlocC usando Cubits. Organização estruturada, lógica clara e testes robustos são fundamentais para o desenvolvimento eficiente de aplicativos Flutter.

5 Avançando com Cubit

5.1 Integração com API e Dados Assíncronos

Vamos mergulhar na integração de Cubit com APIs e dados assíncronos. Gerencie requisições assíncronas de forma eficaz para criar aplicativos dinâmicos. Requisição Assíncrona:

```dart
// Exemplo de Cubit para gerenciar dados assíncronos
class DataCubit extends Cubit<List<Data>> {
  DataCubit() : super([]);

  Future<void> fetchData() async {
    final data = await ApiService.getData();
    emit(data);
  }
}

```

No exemplo acima a chamada esta sendo feita diretamente no cubit, ao seguir nosso exemplo de estrutura já visto anteriormente, esta chamada seria feita dentro do controller. O cubit chamaria o controller e o controller chamaria a Api.

Aprenda a gerenciar requisições assíncronas com Cubit, proporcionando uma experiência de usuário suave enquanto os dados são carregados. Gerenciamento Assíncrono:

```dart
// Exemplo de Cubit gerenciando o estado durante uma requisição assíncrona
BlocBuilder<DataCubit, List<Data>>(
  builder: (context, state) {
    if (state.isEmpty) {
      return CircularProgressIndicator();
    } else {
      return ListView.builder(
        itemCount: state.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(state[index].name),
          );
        },
      );
    }
  },
);
```

Descubra como lidar com atualizações dinâmicas nos dados, mantendo a interface gráfica sincronizada com as mudanças no estado do Cubit. Atualizações Dinâmicas:

```dart
// Exemplo de Cubit emitindo atualizações dinâmicas
class RealTimeDataCubit extends Cubit<List<Data>> {
  RealTimeDataCubit() : super([]);

  void updateData(Data newData) {
    final updatedData = List.from(state)..add(newData);
    emit(updatedData);
  }
}
```

5.3 Extensões e Plugins Úteis

Explore extensões e plugins úteis para otimizar o desenvolvimento Flutter. Descubra recursos adicionais que facilitam a construção de aplicativos poderosos. Utilizando Extensões:

```dart
// Exemplo de extensão para formatar datas
extension DateFormatting on DateTime {
  String formattedDate() {
    return '${this.day}/${this.month}/${this.year}';
  }
}
// Uso da extensão
final currentDate = DateTime.now();
print(currentDate.formattedDate());  // Saída: '10/01/2024'
```

Aprofunde-se em recursos adicionais que o Flutter oferece. Explore extensões e plugins para aprimorar a experiência de desenvolvimento.

Descubra as contribuições valiosas da comunidade Flutter. Explore plugins e extensões desenvolvidos por outros desenvolvedores, enriquecendo o ecossistema Flutter.

```dart
// Exemplo de uso de um plugin para navegação avançada
import 'package:flutter_bloc/flutter_bloc.dart';

class AdvancedNavigator {
  static void navigateToNextScreen(BuildContext context) {
    Navigator.of(context).pushNamed('/next_screen');
  }
}
```

Este capítulo leva você além do básico com Cubit no Flutter, explorando integração assíncrona, atualizações dinâmicas, extensões úteis e contribuições da comunidade. Desenvolva aplicativos robustos e amplie suas habilidades com as ferramentas certas.

A comunidade do flutter esta em constante evolução com uma infinidade de plugins. Acesse o https://pub.dev/ e de uma olhada, o flutter também faz apresentações de alguns plugins todo mês/semana.

6 Conclusão e Recursos Adicionais

6.1 Recapitulando Principais Conceitos

Vamos recapitular os principais conceitos abordados nesta jornada. Cubit, com seu gerenciamento de estado eficiente, proporciona uma estrutura sólida para o desenvolvimento Flutter. Benefícios do Uso do Cubit:

Simplicidade: Cubit simplifica o gerenciamento de estado, tornando o código mais claro.

Reatividade: Responde eficientemente às mudanças de estado, proporcionando uma experiência de usuário fluida.

Comparação com Outras Alternativas: Ao comparar com outras alternativas de gerenciamento de estado, Cubit destaca-se pela sua abordagem direta e fácil integração com o Flutter.

6.2 Recursos Adicionais e Leituras

Para aprimorar ainda mais suas habilidades no Flutter, explore os seguintes recursos adicionais e leituras recomendadas.

A documentação oficial do Flutter e tutoriais específicos sobre Cubit oferecem insights detalhados e exemplos práticos.

Participe de comunidades online e fóruns de suporte para interagir com outros desenvolvedores Flutter. Troque experiências, obtenha ajuda e fique atualizado com as práticas mais recentes.

```
1  https://flutter.dev/docs/get-started/install
2  https://pub.dev/packages/bloc
3  https://pub.dev/packages/flutter_bloc
4  https://flutter.dev/community
5  https://stackoverflow.com/questions/tagged/flutter
```

Links Uteis

https://flutter.dev/docs/get-started/install
https://pub.dev/packages/bloc
https://pub.dev/packages/flutter_bloc
https://stackoverflow.com/questions/tagged/flutter
https://flutter.dev/community

Linkedin:
https://www.linkedin.com/in/eduardo-c-798ab1236/