Persistência de Objetos

Neste módulo, introduziremos o conceito de persistência de dados em aplicativos móveis usando Flutter. A persistência de dados é essencial para manter informações entre sessões de uso do aplicativo. Abordaremos os conceitos básicos de armazenamento local e mostraremos como salvar e recuperar dados simples usando o pacote shared_preferences. Veremos um exemplo de código passo a passo que ilustra como salvar e recuperar uma string.

Introdução à Persistência de Dados

Conceito de Persistência de Dados

Persistência de dados refere-se ao processo de armazenamento de informações de forma que elas possam ser recuperadas e usadas novamente em sessões futuras do aplicativo. Isso é fundamental para manter configurações, preferências do usuário e outros dados essenciais.

Usando shared_preferences para Persistência Simples

O pacote shared_preferences permite salvar dados simples em pares chave-valor. Isso é útil para armazenar pequenas quantidades de informações, como preferências de usuário.

Passo a Passo com Exemplo de Código

1. Adicionar dependência ao pubspec.yaml

dependencies:

```
  flutter:
    sdk: flutter
  shared_preferences: ^2.0.6
```

2. Importar o pacote no arquivo Dart

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

3. Salvar um dado (String)

```
Future<void> _saveData(String value) async {
  final prefs = await SharedPreferences.getInstance();
  await prefs.setString('my_key', value);
}
```

4. Recuperar um dado (String)

```
Future<String?> _getData() async {
  final prefs = await SharedPreferences.getInstance();
  return prefs.getString('my_key');
}
```

5. Exemplo Completo

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

```
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomeScreen(),
    );
  }
}
class HomeScreen extends StatefulWidget {
  @override
  _HomeScreenState createState() => _HomeScreenState();
}
class _HomeScreenState extends State<HomeScreen> {
  TextEditingController _controller = TextEditingController();
  String _savedData = 'Nenhum dado salvo';
  @override
```

```
  void initState() {
    super.initState();
    _loadData();
  }

  Future<void> _saveData(String value) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('my_key', value);
    setState(() {
      _savedData = value;
    });
  }

  Future<void> _loadData() async {
    final prefs = await SharedPreferences.getInstance();
    setState(() {
      _savedData = prefs.getString('my_key') ?? 'Nenhum dado salvo';
    });
  }

  @override
  Widget build(BuildContext context) {
```

```
return Scaffold(
 appBar: AppBar(
  title: Text('Persistência de Dados'),
 ),
 body: Padding(
  padding: const EdgeInsets.all(16.0),
  child: Column(
   children: <Widget>[
    TextField(
     controller: _controller,
     decoration: InputDecoration(labelText: 'Digite um texto'),
    ),
    ElevatedButton(
     onPressed: () {
      _saveData(_controller.text);
     },
     child: Text('Salvar Dados'),
    ),
    SizedBox(height: 20),
```

```
            Text('Dado salvo: $_savedData'),
          ],
        ),
      ),
    );
  }
}
```

Explicação: Este exemplo cria um aplicativo Flutter simples que permite ao usuário digitar um texto e salvá-lo usando shared_preferences. O texto salvo é recuperado e exibido na tela quando o aplicativo é iniciado.

Nesta aula, exploraremos o uso do SQLite para persistência de dados mais complexa em Flutter. O SQLite é um banco de dados leve que pode ser embutido em aplicativos móveis. Veremos como configurar e usar o SQLite com o pacote sqflite, criando um banco de dados, tabelas e realizando operações básicas.

Usando SQLite com Flutter

Conceito de SQLite

SQLite é um banco de dados relacional leve, ideal para aplicações móveis por ser autocontido e fácil de usar. Ele permite armazenar dados estruturados localmente no dispositivo.

Configurando SQLite com sqflite

1. Adicionar dependência ao pubspec.yaml

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: ^2.0.0+4
  path: ^1.8.0
```

2. Importar os pacotes necessários

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';
```

3. Inicializar o banco de dados

```
Future<Database> initializeDB() async {
  String path = await getDatabasesPath();
  return openDatabase(
    join(path, 'example.db'),
    onCreate: (database, version) async {
      await database.execute(
        'CREATE TABLE items(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT)',
      );
    },
```

```
    version: 1,
  );
}
```

4. Inserir um item no banco de dados

```
Future<void> insertItem(String name) async {
  final db = await initializeDB();
  await db.insert(
    'items',
    {'name': name},
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}
```

5. Recuperar itens do banco de dados

```
Future<List<Map<String, dynamic>>> retrieveItems() async {
  final db = await initializeDB();
  return db.query('items');
}
```

6. Exemplo Completo

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

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

import 'package:path/path.dart';

void main() {

  runApp(MyApp());

}

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      home: HomeScreen(),

    );

  }

}

class HomeScreen extends StatefulWidget {

  @override

  _HomeScreenState createState() => _HomeScreenState();

}

class _HomeScreenState extends State<HomeScreen> {

  TextEditingController _controller = TextEditingController();
```

```
List<Map<String, dynamic>> _items = [];

@override

void initState() {

 super.initState();

 _loadItems();

}

Future<Database> initializeDB() async {

 String path = await getDatabasesPath();

 return openDatabase(

  join(path, 'example.db'),

  onCreate: (database, version) async {

   await database.execute(

     'CREATE TABLE items(id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT)',

    );

  },

  version: 1,

 );

}
```

```
Future<void> insertItem(String name) async {
  final db = await initializeDB();
  await db.insert(
    'items',
    {'name': name},
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
  _loadItems();
}
Future<void> _loadItems() async {
  final db = await initializeDB();
  final data = await db.query('items');
  setState(() {
    _items = data;
  });
}
@override
Widget build(BuildContext context) {
```

```
return Scaffold(
 appBar: AppBar(
  title: Text('SQLite com Flutter'),
 ),
 body: Padding(
  padding: const EdgeInsets.all(16.0),
  child: Column(
   children: <Widget>[
    TextField(
     controller: _controller,
     decoration: InputDecoration(labelText: 'Digite um nome'),
    ),
    ElevatedButton(
     onPressed: () {
      insertItem(_controller.text);
     },
     child: Text('Inserir no Banco de Dados'),
    ),
    SizedBox(height: 20),
```

```
        Expanded(
          child: ListView.builder(
            itemCount: _items.length,
            itemBuilder: (context, index) {
              return ListTile(
                title: Text(_items[index]['name']),
              );
            },
          ),
        ),
      ],
    ),
  ),
);
}
}
```

Explicação: Este exemplo mostra como configurar o SQLite com sqflite, criar uma tabela, inserir itens e recuperar itens para exibição em uma lista. O aplicativo permite que o usuário insira nomes que são salvos no banco de dados e exibidos em uma lista.

Nesta aula, exploraremos as operações CRUD (Create, Read, Update, Delete) usando SQLite no Flutter. Veremos como realizar cada uma dessas operações com exemplos de código detalhados, permitindo que você gerencie dados de maneira eficiente em seu aplicativo Flutter.

Operações CRUD com SQLite

Operação de Criação (Create)

1. Inserir um novo item

```
Future<void> insertItem(String name) async {
  final db = await initializeDB();
  await db.insert(
    'items',
    {'name': name},
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}
```

Operação de Leitura (Read)

2. Recuperar todos os itens

```
Future<List<Map<String, dynamic>>> retrieveItems() async {
  final db = await initializeDB();
  return db.query('items');
```

```
}
```

Operação de Atualização (Update)

3. Atualizar um item existente

```
Future<void> updateItem(int id, String newName) async {
  final db = await initializeDB();
  await db.update(
    'items',
    {'name': newName},
    where: 'id = ?',
    whereArgs: [id],
  );
}
```

Operação de Exclusão (Delete)

1. Deletar um item existente

```
Future<void> deleteItem(int id) async {
  final db = await initializeDB();
  await db.delete(
    'items',
    where: 'id = ?',
```

```
    whereArgs: [id],
  );
}
```

2. Exemplo Completo

```
import 'package:flutter/material.dart';
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomeScreen(),
    );
  }
}
class HomeScreen extends StatefulWidget {
```

```
  @override
  _HomeScreenState createState() => _HomeScreenState();
}
class _HomeScreenState extends State<HomeScreen> {
  TextEditingController _controller = TextEditingController();
  List<Map<String, dynamic>> _items = [];
  @override
  void initState() {
    super.initState();
    _loadItems();
  }
  Future<Database> initializeDB() async {
    String path = await getDatabasesPath();
    return openDatabase(
      join(path, 'example.db'),
      onCreate: (database, version) async {
        await database.execute(
          'CREATE TABLE items(id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT)',
```

```
      );
    },
    version: 1,
  );
}
Future<void> insertItem(String name) async {
  final db = await initializeDB();
  await db.insert(
    'items',
    {'name': name},
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
  _loadItems();
}
Future<void> updateItem(int id, String newName) async {
  final db = await initializeDB();
  await db.update(
    'items',
    {'name': newName},
```

```
    where: 'id = ?',
    whereArgs: [id],
  );
  _loadItems();
 }
 Future<void> deleteItem(int id) async {
  final db = await initializeDB();
  await db.delete(
    'items',
    where: 'id = ?',
    whereArgs: [id],
  );
  _loadItems();
 }
 Future<void> _loadItems() async {
  final db = await initializeDB();
  final data = await db.query('items');
  setState(() {
    _items = data;
```

```
  });
}
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Operações CRUD com SQLite'),
    ),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          TextField(
            controller: _controller,
            decoration: InputDecoration(labelText: 'Digite um nome'),
          ),
          ElevatedButton(
            onPressed: () {
              insertItem(_controller.text);
```

```
  },
  child: Text('Inserir no Banco de Dados'),
),
SizedBox(height: 20),
Expanded(
  child: ListView.builder(
    itemCount: _items.length,
    itemBuilder: (context, index) {
      return ListTile(
        title: Text(_items[index]['name']),
        trailing: Row(
          mainAxisSize: MainAxisSize.min,
          children: [
            IconButton(
              icon: Icon(Icons.edit),
              onPressed: () {
                updateItem(_items[index]['id'], 'Novo Nome');
              },
            ),
```

```
                IconButton(
                  icon: Icon(Icons.delete),
                  onPressed: () {
                    deleteItem(_items[index]['id']);
                  },
                ),
              ],
            ),
          );
        },
      ),
    ),
  ],
),
),
);
}
}
```

Explicação: Este exemplo demonstra todas as operações CRUD usando SQLite. O usuário pode inserir, atualizar, visualizar e deletar itens no banco de dados através de uma interface simples.

Nesta aula, abordaremos as melhores práticas para persistência de dados em Flutter. Discutiremos como estruturar e organizar seu código para garantir eficiência e manutenibilidade, além de práticas recomendadas para segurança e performance. Veremos como aplicar essas práticas em um exemplo real.

Melhores Práticas em Persistência de Dados

Estrutura e Organização do Código

1. Separação de Responsabilidades

Mantenha a lógica de persistência separada da lógica de apresentação. Use classes de serviço ou repositório para interagir com o banco de dados.

2. Uso de Models

Crie modelos (models) para representar seus dados de forma estruturada.

```
class Item {
  final int? id;
  final String name;
  Item({this.id, required this.name});
  Map<String, dynamic> toMap() {
    return {'id': id, 'name': name};
  }
```

}

3. Gerenciamento de Estado

Utilize pacotes de gerenciamento de estado como provider ou bloc para manter o estado da aplicação consistente.

Segurança e Performance

1. Evite o Uso de Strings Literais para Consultas

Use parâmetros nomeados para evitar injeção de SQL.

await db.query('items', where: 'id = ?', whereArgs: [id]);

2. Compactar o Banco de Dados

Periodicamente, compacte o banco de dados para otimizar o espaço.

await db.execute('VACUUM');

3. Criptografia

Para dados sensíveis, considere usar criptografia para proteger informações armazenadas.

4. Fechar Conexões

Certifique-se de fechar as conexões com o banco de dados para liberar recursos.

await db.close();

5. Exemplo Completo

import 'package:flutter/material.dart';

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

import 'package:path/path.dart';

void main() {

  runApp(MyApp());

}

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      home: HomeScreen(),

    );

  }

}



class Item {

  final int? id;

  final String name;

  Item({this.id, required this.name});

  Map<String, dynamic> toMap() {
```

```
    return {'id': id, 'name': name};
  }
}
class DatabaseService {
  Future<Database> initializeDB() async {
    String path = await getDatabasesPath();
    return openDatabase(
      join(path, 'example.db'),
      onCreate: (database, version) async {
        await database.execute(
          'CREATE TABLE items(id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT)',
        );
      },
      version: 1,
    );
  }
  Future<void> insertItem(Item item) async {
    final db = await initializeDB();
```

```
  await db.insert(
    'items',
    item.toMap(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}
Future<List<Item>> retrieveItems() async {
  final db = await initializeDB();
  final List<Map<String, dynamic>> maps = await db.query('items');
  return List.generate(maps.length, (i) {
    return Item(
      id: maps[i]['id'],
      name: maps[i]['name'],
    );
  });
}
Future<void> updateItem(Item item) async {
  final db = await initializeDB();
  await db.update(
```

```
      'items',
      item.toMap(),
      where: 'id = ?',
      whereArgs: [item.id],
    );
  }
  Future<void> deleteItem(int id) async {
    final db = await initializeDB();
    await db.delete(
      'items',
      where: 'id = ?',
      whereArgs: [id],
    );
  }
  Future<void> closeDB() async {
    final db = await initializeDB();
    await db.close();
  }
}
```

```
class HomeScreen extends StatefulWidget {
  @override
  _HomeScreenState createState() => _HomeScreenState();
}
class _HomeScreenState extends State<HomeScreen> {
  final DatabaseService dbService = DatabaseService();
  List<Item> _items = [];
  @override
  void initState() {
    super.initState();
    _loadItems();
  }
  Future<void> _loadItems() async {
    final items = await dbService.retrieveItems();
    setState(() {
      _items = items;
    });
  }
  @override
```

```
Widget build(BuildContext context) {
 return Scaffold(
  appBar: AppBar(
   title: Text('Melhores Práticas em Persistência'),
  ),
  body: Padding(
   padding: const EdgeInsets.all(16.0),
   child: Column(
    children: <Widget>[
     Expanded(
      child: ListView.builder(
       itemCount: _items.length,
       itemBuilder: (context, index) {
        return ListTile(
         title: Text(_items[index].name),
         trailing: Row(
          mainAxisSize: MainAxisSize.min,
          children: [
           IconButton(
```

```
            icon: Icon(Icons.edit),
            onPressed: () {
              dbService.updateItem(
                Item(id: _items[index].id, name: 'Novo Nome'),
              );
              _loadItems();
            },
          ),
          IconButton(
            icon: Icon(Icons.delete),
            onPressed: () {
              dbService.deleteItem(_items[index].id!);
              _loadItems();
            },
          ),
        ],
      ),
    );
  },
```

```
            ),
          ),
        ],
      ),
    ),
  );
 }
 @override
 void dispose() {
  dbService.closeDB();
  super.dispose();
 }
}
```

Explicação: Este exemplo aplica melhores práticas de persistência de dados, incluindo a separação de responsabilidades com a classe DatabaseService, uso de modelos, e fechamento adequado da conexão com o banco de dados.

Esses exemplos e explicações fornecem uma base sólida para iniciantes em Flutter entenderem e implementarem persistência de dados em seus aplicativos. Para mais detalhes, consulte a documentação oficial do Flutter: Flutter Documentation.

Materiais Extras

Você pode realizar o download do arquivo contendo os materiais extras utilizados ao longo das aulas por meio do seguinte link: https://drive.google.com/file/d/1mg7lqMl8Pt2zl0rHIsFS0Qew00YN-sEX/view?usp=sharing.

Conteúdo Bônus

Para auxiliar seu entendimento na aplicação do SQLite no Flutter, sugiro o vídeo "Tutorial SQLite no Flutter - Parte 1", apresentado pelo Prof. Diego Antunes, disponível no YouTube. No vídeo, o professor também foca na integração do SQLite ao Flutter, o que pode te auxiliar a consolidar seu preparo para o desenvolvimento de aplicativos com armazenamento local eficiente.

Referências Bibliográficas

BOYLESTAD, R. L.; NASHELSKY, L. Dispositivos Eletrônicos e Teoria de Circuitos. 11. ed. Pearson, 2013.

DEITEL, P. J.; DEITEL, H. M. Ajax, Rich Internet Applications e Desenvolvimento Web para Programadores. Pearson, 2008.

DUARTE, W. Delphi para Android e iOS: Desenvolvendo Aplicativos Móveis. Brasport, 2015.

FELIX, R.; SILVA, E. L. da. Arquitetura para Computação Móvel. 2. ed. Pearson, 2019.

LEE, V.; SCHNEIDER, H.; SCHELL, R. Aplicações Móveis: Arquitetura, Projeto e Desenvolvimento. Pearson, 2005.

MARINHO, A. L.; CRUZ, J. L. da. Desenvolvimento de Aplicações para Internet. 2. ed. Pearson, 2019.

MOLETTA, A. Você na Tela: Criação Audiovisual para a Internet. Summus, 2019.

SILVA, D. (Org.) Desenvolvimento para dispositivos móveis. Pearson, 2017.

Ir para exercício