04

Utilizando o async await

Transcrição

Nesta etapa, nosso foco será **modificar** nosso código para utilizar algumas **técnicas mais sofisticadas de Dart**. Para isso, nosso objetivo é verificar cada passagem e estabelecer formas mais simples e eficazes de implementação.

No arquivo `app_database.dart`, a operação com banco de dados trabalha em torno de `Future`, ou seja, todas as execuções são feitas de maneira assíncrona, o que é bastante comum quando trabalhamos com **IO** em Dart. Por isso, existe uma técnica que simplifica nosso código da mesma forma que `join()`, o qual não é um `Future` e devolve seu valor diretamente por ser uma execução síncrona que não tende a demorar.

Então, teremos a possibilidade de trabalhar com `Future` chamando uma função que devolve o valor esperado com a técnica `async` `await`, parecida com `syntax sugar`. Portanto, ao invés de trabalhar com `.then()`, conseguiremos retornar diretamente para a variável que esperamos.

Para fazer as conversões do código e utilizar `async` `await`, começamos comentando todo o bloco atual de `createDatabase()` para ver a diferença entre as duas aplicações.

O primeiro passo é realizar a execução de `getDatabasePath()` esperando pegar `dbPath` que representa uma `String`. Se fizéssemos de maneira hiperativa, adicionaríamos `final String` indicando `dbPath`; porém, como estamos em `Future`, não conseguimos pegar esta informação, mas com `async` `await` é possível.

Para usar esta técnica, precisamos indicar que o escopo dessa função trabalha com `async` `await` e é executado dentro de um `Future` por meio da chave `async`. Para que o valor esperado seja retornado apenas quando tivermos a resposta deste próprio `Future`, inserimos a chamada de `await` antes de `getDatabasePath()`.

Basicamente estamos executando `.then()` quando adicionamos `await`, mas já devolvemos o valor `dbPath` para nossa variável. Isso só é possível porque o código está sendo executado no escopo de uma `Future`.

```
Future<Database> createDatabase() async {
    final String dbPath = await getDatabasePath();
```

Dessa maneira, nossa sintaxe está bem mais enxuta do que antes, sem necessidade de lidar com *callbacks* de `.then()`. Agora, podemos replicar os demais código como `join()` com o mesmo valor esperado seguido do nome do banco de dados. Se preferir, use o bloco comentado para fazer as modificações e agilizar o trabalho, ou ainda mantenha-os comentados no texto para comparação se preferir.

Também temos a `String` `path` para retornar diretamente o `openDatabase()` na sequência, visto que estamos no escopo de `Future`. Isso significa que quando fazemos o retorno, é a instância de um `Database` envolvida em um `Future`. Se tentarmos usar `return` para uma `String`, o sistema nos alerta que não funciona desta forma; mas se retornarmos nosso `Database`, o programa consegue devolver `Future<Database>`, pois todo seu escopo é executado de maneira assíncrona e nos permite fazer as configurações.

```
Future<Database> createDatabase() async {
    final String dbPath = await getDatabasePath();
    final String path = join(dbPath, 'bytebank.db');
    return openDatabase(path, onCreate: (db, version) {
```

```
        db.execute('CREATE TABLE contacts('
            'id INTEGER PRIMARY KEY, '
            'name TEXT, '
            'account_number INTEGER) ');
        }, version: 1,
//            onDowngrade: onDatabaseDowngradeDelete,
        );

//bloco anterior comentado
}
```

Uma outra técnica que podemos usar é indicar com `await` que queremos mandar `getDatabasePath()` para `join()`, ao invés de devolver `dbPath` para a variável. Desta forma, `join()` só é executado quando `await` for finalizado, o qual segura a execução de `async`.

Isso nos permite fazer com que `await` entre como argumento, simplificando o código e otimizando a conversão para os demais blocos.

```
Future<Database> createDatabase() async {
    final String path = join(await getDatabasePath(), 'bytebank.db');
    return openDatabase(path, onCreate: (db, version) {
        db.execute('CREATE TABLE contacts('
            'id INTEGER PRIMARY KEY, '
            'name TEXT, '
            'account_number INTEGER)');
        }, version: 1,
//            onDowngrade: onDatabaseDowngradeDelete,
        );

//bloco anterior comentado
}
```

Aplicamos a mesma metodologia de `async` em `save()`; no caso de `createDatabase()`, substituímos por `getDatabase()` para ficar mais consistente em relação ao que precisamos, que é pegar o banco de dados.

Para deixarmos de trabalhar com `.then()` neste caso também, começamos com `getDatabase()` indicando que temos acesso ao `Database` com `await`. Em seguida, copiamos o bloco de `Map` para retornar `db.insert()` e resolver o código.

```
Future<int> save(Contact contact) async {
    final Database db = await getDatabase();
    final Map<String, dynamic> contactMap = Map();
    contactMap['name'] = contact.name;
    contactMap['account_number'] = contact.accountNumber;
    return db.insert('contacts', contactMap);

// bloco anterior comentado
}
```

Agora podemos partir para o código de `findAll()` seguindo a mesma técnica. Adicionamos `async` para executar tudo na `Future`, pegamos `getDatabase()`, retornamos para uma variável `Database db` e inserimos o `await` da mesma forma já feita.

Feito isso, podemos trabalhar com `.query()` em nossa tabela de contatos. Já que estamos lidando com `async` e `await`, devemos rever o valor esperado neste caso, pois não colocamos o valor `maps` da referência de `map`; como vimos com `.then()` de `.query()`, o programa espera uma lista com um mapa de `String` de tipo dinâmico.

Portando, escrevemos `final` para representar o resultado chamado `result` de forma mais literal. Com o valor retornado, podemos trabalhar em cima; usamos o mesmo código de `for()` comentado com retorno de `contacts`.

Em seguida, pegamos a lista e a introduzimos após a linha de `await` para termos todos os passos. Trabalhando com `result`, pegamos cada um dos mapas e renomeamos todas as ocorrências de `map` para um nome `row` mais específico que tenha maior relação com as tabelas do banco de dados. Para facilitar, selecionamos a opção "*Rename all occurrences*".

Usando basicamente o mesmo código de antes, pegamos nosso resultado, criamos uma nova lista operando em cada uma das linhas encontradas, criamos um contato e imputamos na lista. Por fim, retornamos a lista de contatos.

```
Future<List<Contact>> findAll() async {
    final Database db = await getDatabase();
    final List<Map<String, dynamic>> result = await db.query('contacts');
        final List<Contact> contacts = List();
        for (Map<String, dynamic> row in result) {
        final Contact contact = Contact(
            row['id'],
            row['name'],
            row['account_number'],
        );
        contacts.add(contact);
    }
    return contacts;

//bloco anterior comentado
}
```

Desta forma, o código fica mais simples em relação ao que estava antes com muitos callbacks e trabalha com apenas o retorno exato que queremos.

Ainda, poderíamos pegar a informação de `result` e substituir em `for()` se quiséssemos, da mesma forma feita com `getDatabasePath()`, sendo uma variável a menos para lidar.

Fizemos todas estas modificações e precisamos testar o código novo no emulador passando por todas as telas e ações da aplicação para avaliar o comportamento.

Com tudo funcionando como esperado, continuamos trabalhando com `Future`, mas a diferença é a maneira de trabalhar com `syntax sugar` para não precisarmos lidar com diversos callbacks, pois é bastante comum em Dart; conforme avançamos nos projetos e temos operações assíncronas com base em `Future`, consideramos `async` `await` para deixar o código mais simples e eficaz.

Aprender esta técnica e adotá-la pode trazer benefícios para projetos em Dart.

A seguir, **refatoraremos** mais o código.