**Aula XXIV e XXV - DevAppII**

**Trabalhando com DataBase**

**SQLite**

**DataBase Relacional**

**Flutter e SQLite**

**Programação Assíncrona**

**Criando o Banco de Dados**

**Cadastrando e listando dados**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Fontes:***

[Flutter - Beautiful native apps in record time](https://flutter.dev/)

<https://flutter.dev/docs/cookbook/persistence/sqlite>

<https://pub.dev/packages/sqflite>

<https://rockcontent.com/br/blog/sqlite/>

<https://www.devmedia.com.br/bancos-de-dados-relacionais/20401>

<https://firebase.google.com/docs/flutter/setup?hl=pt-br&platform=Android>

[Building chat app with Flutter and Firebase | by Quang Duy Tran | Flutter Community](https://medium.com/flutter-community/building-a-chat-app-with-flutter-and-firebase-from-scratch-9eaa7f41782e)

[FlutterFire](https://firebase.flutter.dev/)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQLite**

O SQLite é uma base de dados relacional de código aberto e que dispensa o uso de um servidor na sua atuação. Armazenando seus arquivos dentro de sua própria estrutura, ele é capaz de funcionar muito bem em aplicações diversas, principalmente, websites de tráfego médio e sistemas mobile. Essa independência acontece porque ele consegue colocar os seus arquivos dentro de si próprio. Essa base de dados é de código aberto e gratuita, sendo muito utilizada em aplicações mobile, **com foco no sistema Android**.



Uma das principais características é que o SQLite dispensa um servidor, e também não demanda nenhum tipo de configuração. Assim, seu uso como base de dados para diferentes tipos de aplicações se torna, além de fácil, mais fluído, dinâmico e leve.

Outro grande diferencial que compõem a lista de qualidades do SQLite é o fato de que ele suporta um grande número de linguagens, algo em torno de 30. Essa versatilidade faz toda diferença para que seu uso tenha sido cada vez mais recorrente.

**Configurações gerais e algumas diferenças com o MySQL**

Começando pela arquitetura, enquanto o SQLite não precisa de um servidor, o MySQL é totalmente dependente nesse sentido, sendo projetado de maneira distinta. **Quanto ao domínio, o MySQL, apesar de código aberto, pertence à Oracle, enquanto o SQLite é público.**

Outra diferença principal é a compatibilidade de dados, muito mais ampla no MySQL, o que ajuda a entender a razão de ele ser mais popular. **O SQLite trabalha de forma limitada, apenas com Integer, Real, Text e Blob.**

O armazenamento é também uma importante questão que marca as diferenças entre essas duas bases de dados. MySQL trabalha com um servidor de 600MB e SQLite disponibiliza 250KB em sua biblioteca. São diferenças marcantes, mas que também indicam por que SQLite costuma ser usado para aplicações mobile.

**Segurança**

Apesar de ter mecanismos de segurança reduzidos, a configuração do SQLite é muito mais fácil e simplificada. Já o MyQSL demanda maior conhecimento e esforços mais extensos nesse sentido, ainda que tenha maior capacidade de proteção, especialmente, com a autenticação que oferece aos usuários.

*Fonte:* [*https://www.devmedia.com.br/sqlite-o-pequeno-notavel/7249*](https://www.devmedia.com.br/sqlite-o-pequeno-notavel/7249)

**Para relembrar:**

**O que é um Banco de Dados Relacional?**

Um banco de dados relacional é um mecanismo de armazenamento que permite a persistência de dados e opcionalmente implementar funcionalidades. **Um Sistema Gerenciador de Banco de Dados Relacional (SGBDR) é um software que controla o armazenamento, recuperação, exclusão, segurança e integridade dos dados em um banco de dados.** **Um banco de dados relacional armazena dados em tabelas.**

**Tabelas são organizadas em colunas, e cada coluna armazena um tipo de dados (inteiro, números reais, strings de caracteres, data, etc.).** Os dados de uma simples “instância” de uma tabela são armazenados como uma linha. Por exemplo, a tabela Cliente teria colunas como numeroCliente, primeiroNome e sobrenome, e uma linha na tabela teria algo como {123, “Pafúncio”, “Silva”}.

Cliente

| Id (integer) | Nome (text) | Idade (integer) |
| --- | --- | --- |
| 001 | Pafúncio | 24 |
| 002 | Adriana | 25 |

Compras

| Código(Integer) | Produto(varchar(50)) | Valor(real) | Id (Integer) |
| --- | --- | --- | --- |
| 4563 | Monitor | R$ 1289,00 | 002 |
|  |  |  |  |

**Flutter e SQLite**

A partir de agora iremos trabalhar com uma aplicação mobile e SQLite. Para começarmos vamos usar um codebase para podermos trabalhar.

Repo: <https://github.com/cristijung/Pets_sqlite.git>

Faça a clonagem dele e abra na IDE que costuma trabalhar.

**Vamos observar a organização dos componentes.**

|  | A organização dos componentes está arranjada de forma padronizada, a cada diretório temos o nosso arquivo .dart.  Esta padronização nos permite termos códigos mais enxutos e dispostos de forma organizada fazendo assim com que o desenvolvedor possa localizar e/ou editar seus componentes de forma mais rápida. |
| --- | --- |

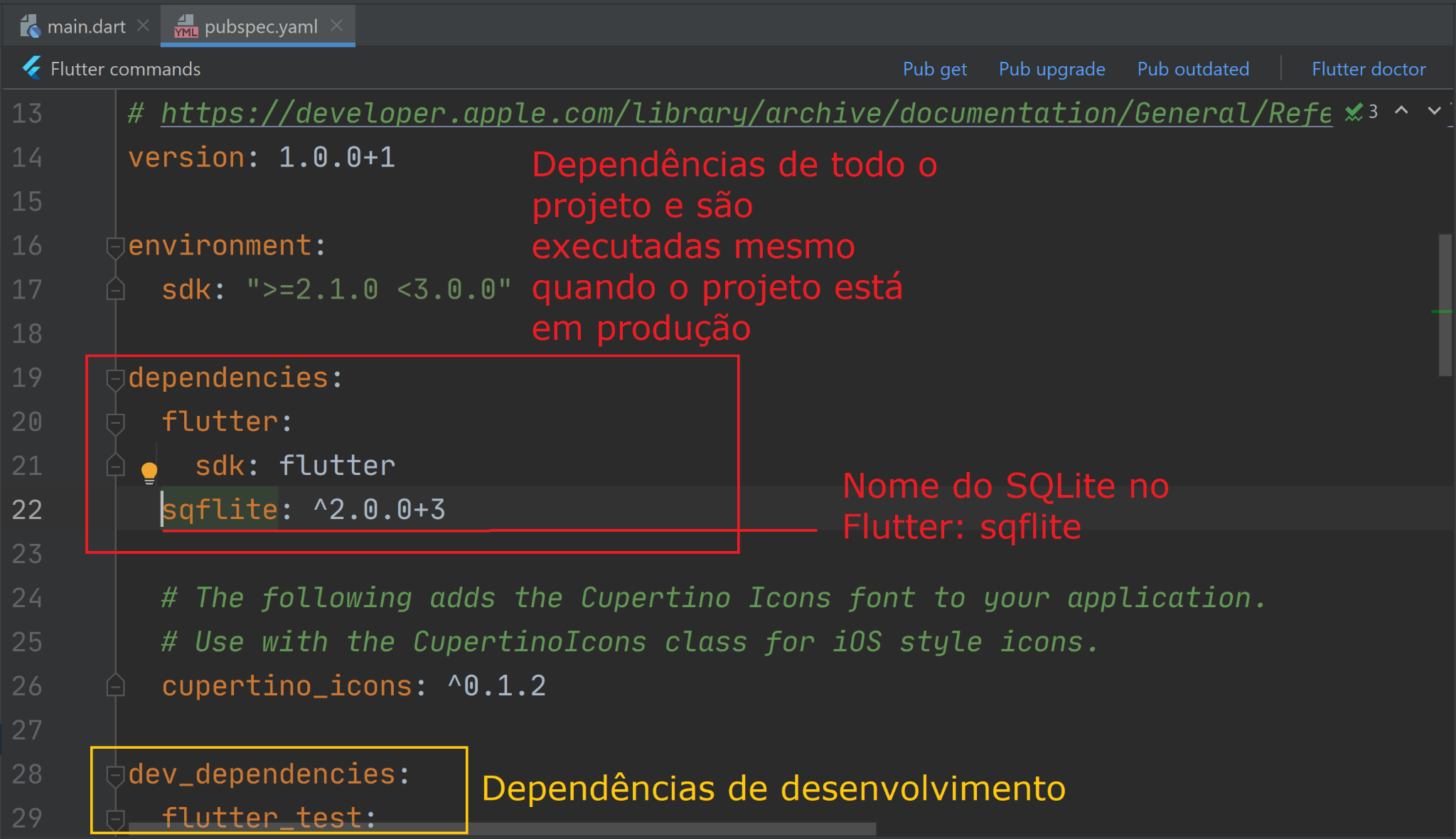
Agora iremos configurar as dependências do SQLite no nosso projeto.

Vamos abrir o arquivo **pubsec.yaml**

Precisamos ter cuidado onde iremos configurar as dependências, o arquivo pubsec é o responsável pelo setamento das dependências de todo o nosso projeto e temos mais de um tipo de dependência que podemos configurar.

**Em dependencies, configuramos as dependências para TODO o projeto, portanto, o SQLite entrará aqui e não em dev\_dependencies!**

**Em dependencies, mesmo quando o projeto está em produção ele será instalado!**



A versão que estamos colocando do SQLite é a última verificada, sempre podemos conferir neste link: <https://pub.dev/packages/sqflite>

Vamos salvar as alterações do arquivo pubsec.yaml, podemos digitar o comando **flutter pub get** no terminal, ou se você está no VsCode, também pode salvar com o arquivo ativo ou no Android Studio clicar sobre o link **pub get.**

**Programação Assíncrona**

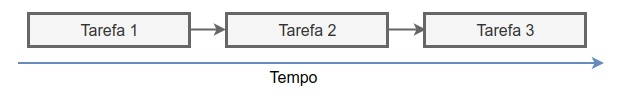
Quando estamos desenvolvendo uma **aplicação Flutter,** é comum precisarmos acessar recursos externos, seja um arquivo, buscar dados de um banco de dados local ou até acessar algum recurso de uma API externa. Porém, todo recurso leva um tempo para retornar algo e precisamos liberar o app para que as outras tarefas continuem executando, “destravando” a execução e permitindo que a interface da tela, por exemplo, seja renderizada. Para isso, utilizamos o conceito de programação assíncrona no Flutter!

**Como funciona a programação assíncrona**

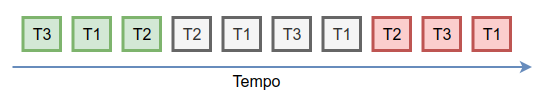
É comum que uma aplicação acesse recursos externos, o que pode levar um certo tempo até que o processo seja finalizado, porém, neste meio tempo, a aplicação precisa ser “liberada” para executar suas outras tarefas, como a renderização da interface da tela.

**Para isso, precisamos trabalhar com um modelo de programação um pouco diferente do “habitual”, a programação assíncrona.**

Basicamente, na programação assíncrona, uma tarefa pode ser executada enquanto espera o processamento de uma outra finalizar. Este modelo é diferente da programação a que estamos mais habituados a trabalhar, chamada de programação síncrona. **Na programação síncrona, uma tarefa só é executada quando outra é finalizada, como podemos ver na imagem abaixo:**



Já na programação assíncrona, diversas tarefas podem ser executadas em “paralelo”, dividindo o tempo com a execução de diferentes tarefas:



**Programação assíncrona no Flutter**

Para utilizar o modelo assíncrono no Flutter, precisamos utilizar três palavras-chave: **async, await e Future.**

* **O async** determina que um método será assíncrono, ou seja, não irá retornar algo imediatamente, então o aplicativo pode continuar a execução de outras tarefas enquanto o processamento não é finalizado.
* **O await** serve para determinar que o aplicativo deve esperar uma resposta de uma função antes de continuar a execução. Isso é muito importante pois há casos em que uma função depende do retorno de outra.
* **Já o Future** determina que uma função irá retornar algo no “futuro”, ou seja, é uma função que levará um tempo até ser finalizada.

**Implementando o modelo assíncrono**

Imagine, então, um método que retornará todos os pets de um determinado banco de dados (não se preocupe com a forma que estes dados serão pesquisados). Como é algo que pode demorar um certo tempo até ser finalizado (temos que buscar os dados do banco de dados, processar as informações para, só então, retornar estes dados), precisamos criar uma função assíncrona, como podemos ver abaixo:

fonte: <https://www.treinaweb.com.br/blog/como-funciona-o-async-await-e-future-no-flutter>

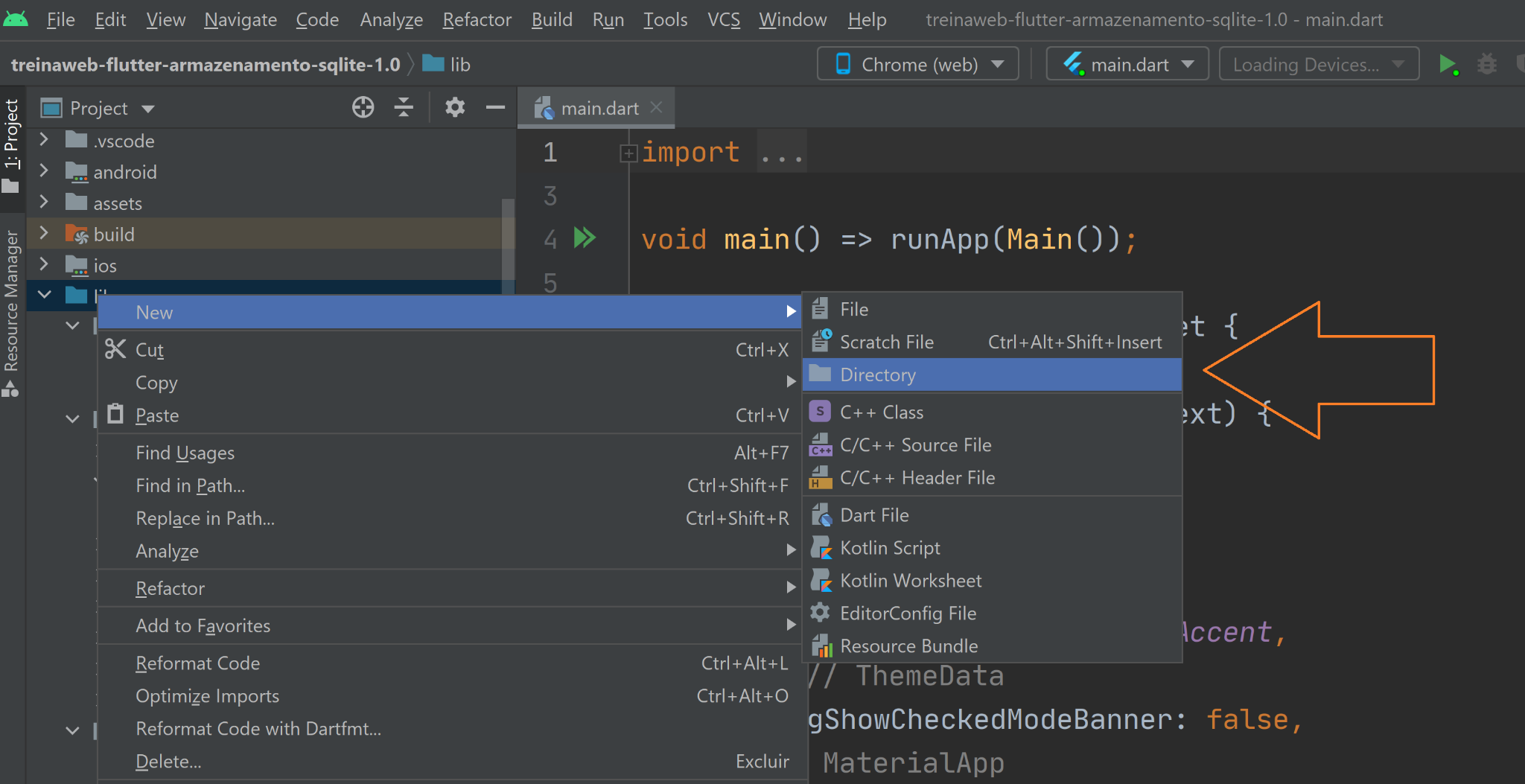
| **Future<List> getAllPets() async {**  **final dataList = await DbUtil.getData('pets');**  **\_petList = dataList.map((pets) => Pet.fromMap(pets)).toList();**  **return \_petList;**  **}** |
| --- |

Depois que vimos uma breve introdução de como o Flutter trabalha com programação assíncrona e síncrona, vamos adiante na criação do nosso banco de dados da aplicação que já temos o código.

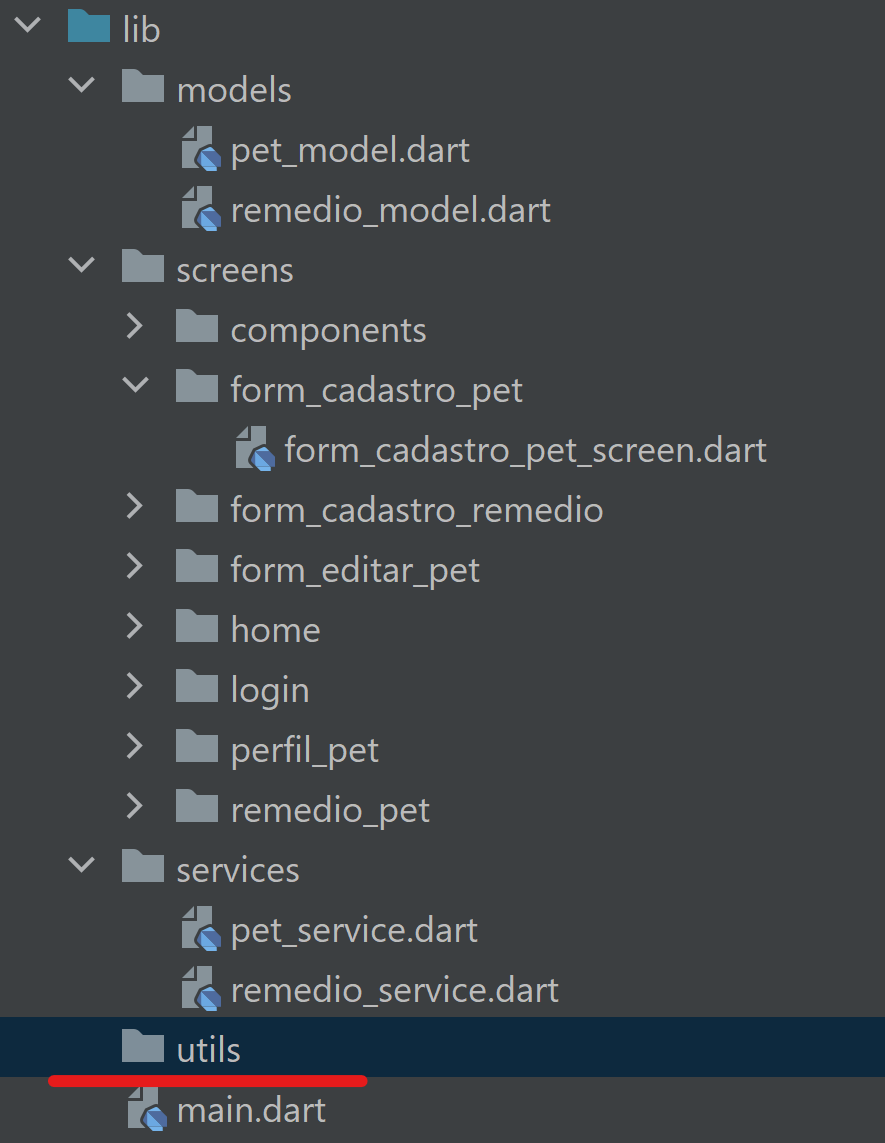
**Criando o Banco de Dados**

Vamos observar que na organização da nossa aplicação já temos o “services” que nada mais é que a definição dos dados que iremos usar para armazenar e editar.

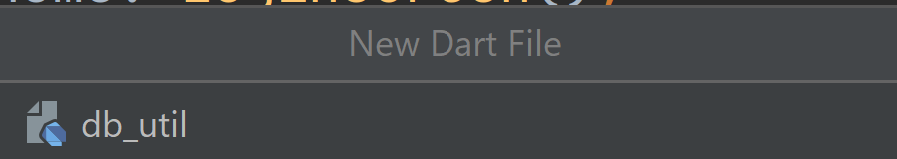
Agora iremos criar outro diretório dentro da raiz do nosso projeto chamado de utils (utilitários):



Nome: utils



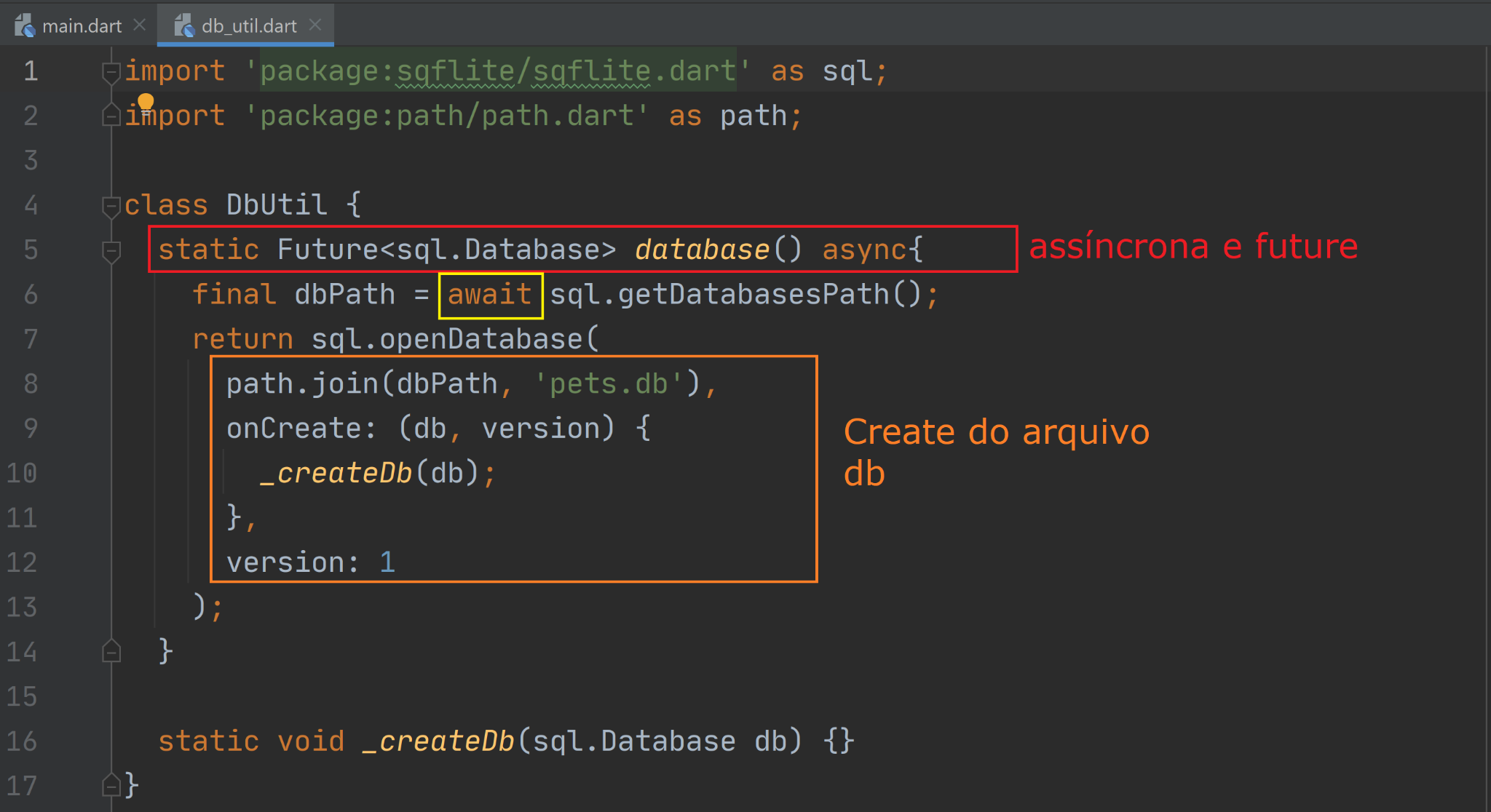
Dentro deste diretório, iremos criar um arquivo .dart com nome de:



Desta forma, deixaremos a camada de services só para organizar os dados e a utils para salvar e editar estes dados.

Nesta camada utils iremos criar métodos para tratar com a tabela de pets e a tabela de remédios.

Neste arquivo db\_util tudo relacionado ao banco de dados será definido aqui, portanto iremos começar com a criação desta classe responsável pelo tratamento dos dados da nossa aplicação.



Pois bem, o método **\_createDb** recebe como parâmetro o sql.Database e a criação do nosso arquivo definido pelo caminho na classe DbUtil.

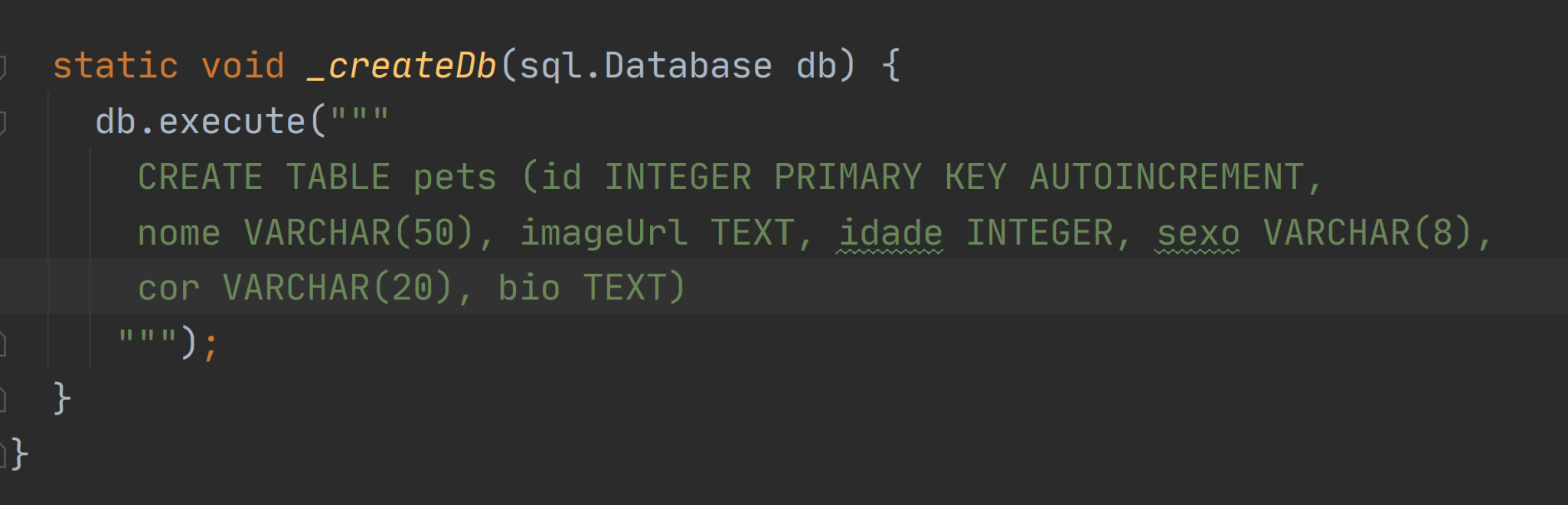
Nesta próxima etapa, iremos criar as tabelas dentro do nosso método, iremos definir dentro do método **db.execute** e iremos passar nossos parâmetros.

Aqui iremos observar um cuidado com o que iremos passar, bem importante definir que:

| **db.execute(“”” “”” )** |
| --- |

Tudo o que passamos por ali, entre estas 3 aspas duplas de cada lado, será reconhecido como um String e não precisamos nos preocupar com fechamento de “”!

No db.execute iremos digitar os códigos SQL para criar a nossa tabela e consequentemente, um banco de dados.



**Cadastrando e listando dados**

A próxima etapa será tratar os dados no formulário de cadastro de pets para definir a persistência destes dados.

Vamos acessar o arquivo form\_cadastro\_pet\_screen e pet\_services.

**…. próxima aula =)**