Banco de dados SQLite na Lista de Compras

Android com Kotlin

Flávio Augusto de Freitas

Banco de dados SQLite na Lista de Compras

Neste projeto, vamos entender como funciona a persistência de dados em um banco de dados SQLite, para isso, vamos aproveitar nosso aplicativo de lista de compras. Nosso aplicativo de lista de compras está bem legal, porém você deve ter percebido que, toda vez que fechamos e abrimos de novo, os itens que estavam inseridos são perdidos. Isso acontece porque não estamos utilizando nenhuma forma de persistência de

dados.

O objetivo deste projeto é armazenar e recuperar dados de um banco de dados SQLite que já está disponível na plataforma do Android. Ao contrário de outros bancos de dados, o SQLite não utiliza um servidor, ele armazena a estrutura de tabelas e dados em um arquivo no disco. Quando criamos um banco de dados para o nosso aplicativo, ele estará acessível em todas as nossas classes, mas não em outro aplicativo. Quando o aplicativo é desinstalado, seus dados também são removidos.

A Biblioteca Anko SQLite

Aproveitarei esta seção para introduzir a biblioteca Anko. O objetivo dessa biblioteca é deixar o desenvolvimento de aplicativos com Kotlin mais rápido e mais fácil. Se o Kotlin por si só já tem essa proposta, o Anko vem para ajudar ainda mais nesse quesito. Abiblioteca Anko também foi desenvolvida pela equipe da JetBrains, a mesma empresa criadora da linguagem Kotlin, e por isso ela tem uma integração incrível com a linguagem.

O Anko pode ser dividido em quatro partes diferentes:

• Anko Commons: uma caixa de ferramentas para diversas funções para as tarefas mais comuns do Android.

• Anko Layouts: criação de layouts dinâmicos de forma rápida e eficiente.

• Anko SQLite: ferramentas que facilitam a criação e manipulação de banco de dados SQLite. • Anko Coroutines: utilitários baseados na biblioteca kotlinx.coroutines.

Por ora, vamos utilizar o Anko SQLite para manipulação do banco de dados. Nos próximos capítulos exploraremos algumas outras ferramentas do Anko, mas vale muito a pena conferir a documentação oficial e todo o poder da biblioteca, em sua página no GitHub: https://github.com/Kotlin/anko.

Usaremos a biblioteca Anko para manipulação de dados com SQLite porque é uma biblioteca fantástica e facilita muito o dia a dia do desenvolvedor, mas não é a única opção para trabalhar com persistência de dados locais no Android. Na hora de decidir qual arquitetura seu aplicativo usará, temos alguns caminhos que podem ser seguidos:

Utilizar a classe SQLiteOpenHelper disponível na plataforma Android

A plataforma Android nos disponibiliza uma classe chamada SQLiteOpenHelper. Com sua implementação, podemos criar um banco de dados, tabelas, gerenciar versões e posteriormente utilizá-la para inserir, deletar, atualizar e buscar dados do banco de dados. A grande vantagem de se implementar diretamente a classe SQLiteOpenHelper é que você terá o controle total do seu banco de dados; a desvantagem é que é uma abordagem muito mais trabalhosa em relação a código.

O seguinte artigo mostra como implementar um banco dedados utilizando a classe SQLiteOpenHelper : https://developer.android.com/training/data-storage/sqlite.html.

Utilizar alguma solução ORM

A sigla ORM vem do inglês *Object Relational Mapper*, que podemos traduzir como "mapeamento de objeto relacional". Na prática, a função do ORM é abstrair ao máximo as interações como banco de dados através de seus objetos relacionais. A ideia é fazer o programador focar seu tempo somente em desenvolver o código e nas lógicas de negócio, e não se preocupar com o banco de dados. A grande vantagem é a facilidade de interação com os dados e a alta produtividade que utilizar um ORM pode trazer ao

2

desenvolvimento do projeto; a desvantagem é que exige uma certa maturidade do programador e perfeito entendimento dos conceitos de Orientação a Objetos para conseguir implementar a solução de forma eficaz.

Durante o evento Google I/O 2017 foi anunciada a biblioteca Room, que é a solução ORM do Google para Android.

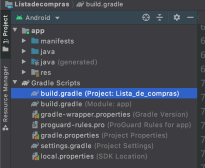
O seguinte artigo mostra como utilizar a biblioteca Room para interações com banco de dados: https://developer.android.com/training/data-storage/room/index.html.

Para este projeto, vamos usar a biblioteca Anko SQLite. Ela não é uma solução de ORM, mas também facilita muito a manipulação de dados em um banco de dados. Na verdade, o Anko SQLite é uma camada de abstração da classe SQLiteOpenHelper feita em Kotlin. A grande diferença é que o Anko utiliza recursos da linguagem Kotlin para facilitar a programação do banco de dados. Ainda teremos tudo sob nosso controle, como se tivéssemos utilizando a classe SQLiteOpenHelper, mas com a vantagem de um código muito mais intuitivo e fácil de implementar.

Adicionando Anko SQLite ao projeto

Para utilizar o Anko, devemos adicionar suas dependências ao nosso projeto nos arquivos de build.gradle.

Primeiro, localize o arquivo build.gradle do projeto, ele estará marcado com a frase (Project: Lista de Compras) ao lado:

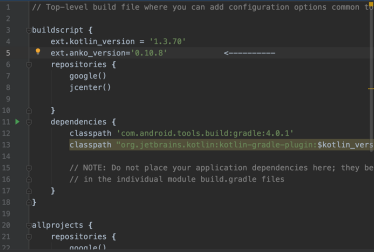


*Figura 1: build.gradle (Project).*

Primeiro, vamos definir uma variável para guardar a versão do Anko. No momento da escrita deste texto, a biblioteca está na versão 0.10.8 (você pode confirmar a versão visitando o GitHub do projeto em https://github.com/Kotlin/anko, rolar a página para baixo e verá a versão do lado direito, logo abaixo de **Releases**.), então faremos uma variável para guardar essa versão e, caso seja atualizada, basta voltar a essa variável e atualizar seu valor.

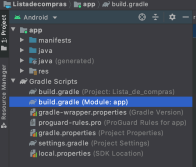
Abra o arquivo build.gradle e adicione, logo abaixo da definição da versão do Kotlin, a seguinte variável: ext.anko\_version='0.10.8'

3

*Figura 2: Anko version.*

A seta indica o local de criação da variável.

Agora devemos incluir as dependências no arquivo build.gradle do módulo. Ele estará logo abaixo do arquivo do projeto. Você localizará visualmente pela frase (Module: app) ao lado do nome do arquivo.



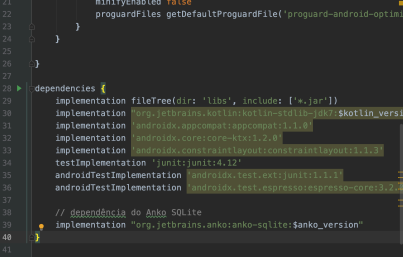
*Figura 3: build.gradle (Module).*

Com o arquivo aberto, navegue até a seção de dependências e adicione a seguinte linha: implementation "org.jetbrains.anko:anko-sqlite:$anko\_version"

*A configuração compile está obsoleta e foi substituída pela configuração implementation junto com o Android Plugin3.0.0: http://d.android.com/r/tools/update-dependency-configurations.html.*

Veja na imagem a dependência do Anko SQLite:

4

*Figura 4: Dependência do Anko SQLite.*

Feito isso, clique no botão Sync Now (sincronizar agora) e o Android baixará a biblioteca e a adicionará ao projeto.

*Dica: a biblioteca Anko possui diversas funções que são úteis ao dia a dia do desenvolvimento Android. Uma delas é o startActivity que vimos no projeto anterior. Lá utilizamos o startActivity para abrir a tela de cadastro com o seguinte código:*

*val intent = Intent(this, CadastroActivity::class.java)*

*startActivity(intent)*

*Com a biblioteca Anko adicionada ao nosso projeto, poderíamos reescrever o mesmo código da seguinte forma:*

*startActivity<CadastroActivity>()*

*Isso mesmo, com uma única linha e com um código muito mais claro! Você precisará do seguinte import para implementação desse código:*

*import org.jetbrains.anko.startActivity*

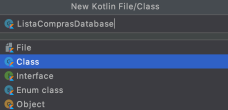
Criando o banco de dados

A primeira coisa a fazer é a criação do banco de dados e das tabelas necessárias. No nosso caso, precisaremos de uma única tabela para guardar as informações de produtos. A biblioteca Anko provê uma classe chamada ManagedSQLiteOpenHelper, que nada mais é do que uma abstração em cima da classe SQLiteOpenHelper. Seu intuito é fazer o gerenciamento do banco de dados, desde a criação até a manipulação de dados. Para criação do nosso banco de dados, faremos uma classe que estende a classe ManagedSQLiteOpenHelper.

5

A classe ManagedSQLiteOpenHelper recebe 4 parâmetros em seu construtor, sendo 3 obrigatórios e 1 opcional. O primeiro é um Context, com o contexto da aplicação (obrigatório); depois uma String com o nome do banco de dados (obrigatório); um objeto do tipo CursorFactory, caso o programador queira implementar uma lógica de cursores personalizada (opcional); e um Integer com a versão do banco de dados (obrigatório).

Então vamos criar uma classe para nosso banco de dados chamada ListaComprasDatabase. Para isso, primeiro crie o arquivo ListaComprasDatabase dentro do pacote br.com.androidkotlin. Clique com o botão direito do mouse em br.com.androidkotlin, que fica dentro de app > java. Escolha New > Kotlin/File Class. Na janela que aparece deixe como na figura abaixo e clique em OK.



*Figura 5: No macOS basta pressionar ENTER.*

Agora vamos implementar o código da classe, que deve ter um construtor que recebe um contexto do tipo Context e deve estender a classe ManagedSQLiteOpenHelper. Observe a implementação desse código.

class ListaComprasDatabase(context: Context) : ManagedSQLiteOpenHelper(ctx = context, name = "listaCompras.db", version = 1) {

}

Adicione também os seguintes imports :

import android.content.Context

import org.jetbrains.anko.db.ManagedSQLiteOpenHelper

Nessa classe devemos implementar 2 métodos: onCreate e onUpgrade.

O método onCreate é responsável pela criação das tabelas no banco de dados. Quando acessamos essa classe, ela automaticamente executa o onCreate e verifica se o banco dedados já está criado ou não. Caso o banco de dados não esteja criado, ele executa o código do onCreate. Por isso, devemos colocar aqui todo o código referente à criação das tabelas, com a seguinte estrutura: fun onCreate(db: SQLiteDatabase), sendo que a variável db é o banco de dados em si, pronto para manipulação.

O outro método que precisamos implementar é o onUpgrade.Diferente do anterior, o onUpgrade é executado somente quando há uma alteração na versão do banco de dados. Ele possui a seguinte estrutura: onUpgrade(db: SQLiteDatabase,oldVersion: Int, newVersion: Int), sendo que a variável db é o banco de dados em si para manipulação; e as variáveis oldVersion e newVersion são números inteiros que guardam

6

aversão do banco de dados. Desta forma, o programador conseguirá gerenciar exatamente as atualizações de acordo com cada versão do banco.

Quando estendemos a classe ManagedSQLiteOpenHelper, passamos um Int com o número de versão. Geralmente começamos com 1 e, quando mudamos esse número de versão, o onUpgrade entra em ação. Então nesse método colocamos todas as alterações de banco de dados, novas tabelas, exclusão de tabelas, criação de novas colunas etc.

Imagine que você lançou um aplicativo na Play Store e ele utiliza um banco de dados com 2 tabelas. Depois de um tempo, você lançou uma atualização, na qual você incluiu uma nova tabela no banco de dados. Nesse cenário, o que acontece com os usuários que já tinham o aplicativo instalado? Será que eles terão que desinstalar o aplicativo, consequentemente perdendo os dados salvos, e instalar novamente para ter a nova versão de banco dedados? Isso me parece uma má ideia! Por isso, existe o onUpgrade e as versões de banco de dados, dessa forma o programador implementa as alterações dentro do método onUpgrade, incrementa a versão do banco e, quando o usuário atualizar o aplicativo, ele receberá somente as atualizações.

Vamos fazer a implementação de ambos os métodos:

import android.content.Context

import android.database.sqlite.SQLiteDatabase

import org.jetbrains.anko.db.ManagedSQLiteOpenHelper

class ListaComprasDatabase(context: Context) : ManagedSQLiteOpenHelper(ctx = context, name = "listaCompras.db", version = 1) {

override fun onCreate(db: SQLiteDatabase) {

// Criação de tabelas

}

override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {

// Atualização do banco de dados

}

}

Agora vamos implementar o código de criação da tabela em si. Para nosso projeto, precisaremos de uma tabela que reflita a estrutura da classe Produto. Então, ela deverá ter colunas para o nome do produto, a quantidade, o valor e sua foto. Além disso, também devemos prever uma coluna de id, para que possamos diferenciar um registro do outro no banco de dados. Esse id deverá ser um número inteiro e autoincremental, de modo que, quando inserirmos um novo registro, o próprio banco de dados criará um id para ele incrementalmente.

7

Também é importante que o id seja uma chave primária. No contexto de banco de dados, chaves primárias são campos que não se repetem na mesma tabela. Nesse caso, não podemos ter dois produtos com o mesmo id, e por isso definir o campo como chave primária vai garantir que o id não se repetirá.

Para a criação das tabelas, usaremos o método createTable da variável db. Esse método cria uma tabela no banco de dados de acordo com alguns parâmetros passados, que serão 3:

• tableName: String : uma String com o nome databela que será criada.

• ifNotExists: Boolean = false : um booleano informando que a criação da tabela seria feita somente se ela ainda não existe; este campo é falso por padrão. Devemos passar true se quisermos que a tabela seja criada se ela não existir ainda.

• vararg columns: Pair<String, SqlType> : são as colunas que a tabela terá, no formato sendo do tipo Pair com o nome da coluna e seu tipo.

Portanto, vamos criar uma tabela chamada produtos com colunas de id, nome, quantidade, valor e foto. A implementação ficará da seguinte maneira:

override fun onCreate(db: SQLiteDatabase) {

// Criação de tabelas

db.createTable("produtos", true,

"id" to INTEGER + PRIMARY\_KEY + UNIQUE,

"nome" to TEXT,

"quantidade" to INTEGER,

"valor" to REAL,

"foto" to BLOB )

}

Esse código precisará do import da biblioteca Anko SQLite:

import org.jetbrains.anko.db.\*

*Quando fazemos um import utilizando \*, estamos importando todas as classes daquele determinado pacote. Se fôssemos importar individualmente, precisaríamos dos seguintes imports :*

*import org.jetbrains.anko.db.ManagedSQLiteOpenHelper*

*import org.jetbrains.anko.db.createTable*

*import org.jetbrains.anko.db.INTEGER*

*import org.jetbrains.anko.db.PRIMARY\_KEY*

*import org.jetbrains.anko.db.UNIQUE*

*import org.jetbrains.anko.db.TEXT*

*import org.jetbrains.anko.db.REAL*

*import org.jetbrains.anko.db.BLOB*

8

Os tipos de dados que podemos usar para criar as colunas são:

• INTEGER : para colunas que guardam valores inteiros;

• REAL : para colunas que guardam valores reais, ou seja, com números quebrados; • TEXT : para guardar valores em texto;

• BLOB : para guardar qualquer valor de modo binário.

Agora criaremos mais alguns códigos nessa classe, com o intuito de facilitar seu uso. Ao final, quando formos fazer qualquer interação com o banco, simplesmente usaremos:

database.use {

//aqui faremos as interações com o banco de dados

}

Como ter uma instância única (Singleton)

Para conseguirmos esse efeito, teremos que fazer um Singleton da nossa classe e dar acesso à instância através do contexto. Singleton significa instância única. É a ideia de ter uma instância única dessa classe juntamente com a linguagem Kotlin que vai trazer toda a facilidade de uso nas manipulações com o banco de dados. Se nossa classe não fosse um Singleton, teríamos que criar a instância dela toda vez que fôssemos usar e não teríamos como simplesmente usar o comando database.use { }.

O padrão Singleton pode ser particularmente útil em algumas situações. Para acesso a bancos de dados é comum termos classes Singleton porque não é necessário ficar criando novas instâncias da classe toda vez que for fazer um acesso ao banco. Como o banco geralmente é único, a instância de acesso também pode ser única.

Vamos a um exemplo prático. Imagine que seu aplicativo possua uma classe Calculadora com um método que efetue a soma de dois números:

Class Calculadora{

fun Adicionar(x: Int, y:Int) = x + y

}

Para usar essa classe basta instanciá-la e chamar o método Adicionar :

val c = Calculadora() // criando a instância

c.Adicionar(1, 3) // chamando o método Adicionar

Esse é um código bem trivial e não temos nenhum problema com ele. Dessa maneira, cada vez que criamos uma instância desse objeto Calculadora(),

val c1 = Calculadora() // criando outra instância

val c2 = Calculadora() // criando mais uma instância

9

teremos uma instância diferente, ou seja, outro espaço na memória RAM foi alocado. Mas em alguns casos não precisamos e nem queremos esse efeito; queremos que que uma classe possua uma instância única, e toda vez que formos usá-la simplesmente acessamos a que já existe, em vez de criar instâncias novas sem necessidade.

Para criar uma classe Singleton devemos seguir a seguinte lógica: criar uma variável estática do mesmo tipo da classe, geralmente chamada de instance, e implementar um método estático que retorna essa variável, geralmente chamado de getInstance. O método getInstance verifica se a instância da classe já existe e, caso sim, retorna a instância existente; caso contrário, cria uma instância na hora.

Em Kotlin, por definição, não temos variáveis estáticas, mas existem os blocos companion object que, no final das contas, são a mesma coisa. Quando tenho variáveis e métodos dentro de blocos companion object, consigo acessá-los sem a necessidade de criar uma instância da classe. Então usaremos esse recurso para criar classes Singletons.

Voltando ao exemplo da classe Calculadora, se fôssemos fazer um Singleton dela poderíamos fazer assim:

class Calculadora{

companion object {

//variável de instância

private var instance: Calculadora? = null

//método que verifica se a instância é nula

fun obterInstancia(): Calculadora {

if (instance == null) {

instance = Calculadora()

}

return instance!!

}

}

fun Adicionar(x: Int, y: Int) = x + y

}

E na hora de usar:

val c = Calculadora.obterInstancia() // obtendo a instância

c.Adicionar(1, 3) // chamando o método Adicionar

Desta forma, se eu chamar 10 vezes a instância desse objeto, nas 10 vezes eu estaria acessando exatamente a mesma instância.

*Para entender um pouco mais sobre o padrão Singleton, recomendo a leitura adicional do artigo: https://www.devmedia.com.br/padrao-de-projeto-singleton-em-java/26392.*

10

Aplicando o padrão Singleton na nossa lista de compras

Vamos por partes, primeiro vamos implementar o padrão Singleton na classe, para isso faremos uso do companion object do Kotlin, utilizamos companion object quando queremos definir uma função ou um objeto estático na classe.

A implementação do código é a seguinte:

//Padrão Singleton

companion object {

private var instance: ListaComprasDatabase? = null

@Synchronized

fun getInstance(ctx: Context): ListaComprasDatabase {

if (instance == null) {

instance = ListaComprasDatabase(ctx.getApplicationContext()) }

return instance!!

}

}

*A anotação @Synchronized protege o método contra acesso concorrente de múltiplas threads.* E para dar acesso à instância, criaremos uma função de extensão:

// Acesso a propriedade pelo contexto

val Context.database: ListaComprasDatabase get() = ListaComprasDatabase.getInstance(getApplicationContext()) Atenção: essa função fica fora da classe, mas pode mantê-la no mesmo arquivo.

Com isso, conseguiremos acessar o banco de dados de qualquer parte do aplicativo simplesmente usando: database.use {

}

Por fim, a nossa classe ficará assim:

import android.content.Context

import android.database.sqlite.SQLiteDatabase

import org.jetbrains.anko.db.\*

class ListaComprasDatabase(context: Context) : ManagedSQLiteOpenHelper(ctx = context, name = "listaCompras.db", version= 1) {

//singleton da classe

companion object {

private var instance: ListaComprasDatabase? = null

@Synchronized

fun getInstance(ctx: Context): ListaComprasDatabase {

if (instance == null) {

11

instance = ListaComprasDatabase(ctx.getApplicationContext())

}

return instance!!

}

}

override fun onCreate(db: SQLiteDatabase) {

// Criação de tabelas

db.createTable("produtos", true,

"id" to INTEGER + PRIMARY\_KEY + UNIQUE,

"nome" to TEXT,

"quantidade" to INTEGER,

"valor" to REAL,

"foto" to BLOB )

}

override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {

// Atualização do banco de dados

}

}

// Acesso à propriedade pelo contexto

val Context.database: ListaComprasDatabase get() = ListaComprasDatabase.getInstance(getApplicationContext())

Você pode conferir a implementação dessa classe na documentação do Anko SQLite: https://github.com/Kotlin/anko/wiki/Anko-SQLite.

A tabela produtos que criamos no banco de dados deve refletir o modelo de dados da classe Produto, por isso criamos a tabela com base nas propriedades da classe. No entanto, a tabela possui uma coluna de id que a classe não possui, por isso devemos voltar na classe Produto e adicionar uma propriedade para o id do tipo Int, pois é o mesmo tipo que definimos no banco de dados.

Abra a classe Produto e inclua a propriedade id como no código a seguir:

data class Produto(val id:Int, val nome:String, val quantidade:Int, val valor:Double, val foto: Bitmap? = null )

Feita essa alteração, o código vai parar de compilar e gerará um erro na CadastroActivity na linha val prod =Produto(produto, qtd.toInt(), valor.toDouble(), imageBitMap). Isso porque agora a classe tem um atributo a mais que não está sendo passado. Não se preocupe com esse erro, vamos resolvê-lo mais adiante.

Feitas essas alterações, a estrutura do nosso banco de dados estará pronta. Observe que, se o banco de dados tivesse mais tabelas, eu poderia criar todas no método onCreate chamando a função createTable para cada uma. Com essa estrutura pronta, quando formos acessar o banco de dados usaremos database.use{ } e entre as chaves faremos as ações relacionadas ao banco de dados.

Esse modelo parece complexo, mas no final das contas é bem simples porque, como a estrutura da classe se manterá a mesma, em um outro aplicativo bastaria seguir a mesma estrutura e modificar os parâmetros relacionados ao projeto.

12

Inserindo dados no banco de dados

Com o banco de dados pronto, podemos agora passar a inserir nele os valores da lista. Para inserir dados, usaremos o método insert, que recebe como parâmetro o nome da tabela e uma lista de chave-valor com os dados a serem inseridos.

Por exemplo, vamos supor que temos uma tabela usuários com as colunas de nome e e-mail e queremos inserir um novo usuário nessa tabela, com o nome de "Januário" e e-mail de"januario@teste.com". Para inseri-lo, faríamos o seguinte código:

database.use {

insert("usuarios", "nome" to "Januário", "email" to "januario@teste.com") }

A função insert necessita do seguinte import :

import org.jetbrains.anko.db.insert

Perceba como o código ficou simples, bastou passar o nome da tabela e, em seguida, o nome das colunas e seus valores. Repare também que eu não precisei passar a coluna id, isso porque o próprio banco de dados vai gerar um id sequencial.

O que precisamos aqui é pegar esse id que foi gerado e guardar em uma variável para poder usar mais adiante. Para isso, basta utilizar o retorno do método insert. Esse método retorna o id do registro inserido, ou retorna -1 caso ocorra algum erro na inserção. Então, podemos fazer:

val idUsuario = insert("usuarios",

"nome" to "Januário",

"email" to januario@teste.com

)

Vamos implementar a função de insert na tela de CadastroActivity, no clique do botão. Vamos seguir a seguinte lógica: após o usuário clicar no botão, vamos continuar verificando se os campos estão preenchidos e, caso sim, vamos já inserir no banco de dados; então, somente se a inserção no banco de dados ocorrer corretamente, vamos inserir esse item na lista.

Lembre-se de que para saber se o item foi inserido corretamente podemos verificar o valor retornado pelo método insert : se ele for diferente de -1 é porque a inserção foi feita com sucesso; caso contrário, exibiremos uma mensagem padrão.

Veja como fica em código:

val idProduto = insert("Produtos",

"nome" to produto,

"quantidade" to qtd,

"valor" to valor.toDouble(),

13

"foto" to imageBitMap

)

if (idProduto != -1L) {

toast("Item inserido com sucesso")

} else {

toast("Erro ao inserir no banco de dados")

}

Faça também o import da função toast :

import org.jetbrains.anko.toast

Enviando mensagens de aviso ao usuário

Toast: observe que no else utilizei o comando toast, esse comando exibe uma pequena mensagem na tela que desaparecerá sozinha depois de alguns segundos. É possível utilizar o comando toast dessa forma, toast("Mensagem"), porque estamos usando a biblioteca Anko, caso contrário, a forma padrão de se escrever um Toast é:

Toast.makeText(context, "Mensagem", Toast.LENGTH\_SHORT).show() Das duas formas, o resultado será o seguinte:



*Figura 6: Mensagem toast*

O Toast é muito útil para enviar pequenas mensagens de aviso ao usuário. Esse tipo de mensagem não necessita de nenhuma confirmação do usuário, ela simplesmente aparece na tela e desaparece depois de alguns segundos.

Alert: outra forma de enviar mensagens ao usuário é através do comando alert. Diferente do toast, o alert envia uma mensagem e espera uma ação do usuário antes de desaparecer da tela. A forma mais simples de se usar o alert é a seguinte:

alert("Mensagem").show()

Import necessário para função alert :

import org.jetbrains.anko.alert

Esse comando resultará no seguinte:

14



*Figura 7: Alert simples, sem título.*

A vantagem do alert é que ele nos dá mais opções de customizações, por exemplo, eu poderia criar uma caixa de alerta com título e mensagem:

alert("Mensagem", "Titulo").show()

E o resultado seria o seguinte:



*Figura 8: Alert, com título.*

Com o alert, podemos também fazer mensagens em que o usuário precisa confirmar alguma ação através de dois botões: Ok e Cancel. Para configurá-los, utilizamos o comando da seguinte maneira:

alert("Mensagem", "Titulo") {

//Botão de OK

yesButton {

//Ação caso escolheu a opção OK

}

//Botão de cancel

noButton {

//Ação caso escolheu a opção CANCEL

}

}.show()

E o resultado será:



*Figura 9: Alert, com botões OK e Cancel.*

15

Os textos "OK" e "CANCEL" são configurados automaticamente de acordo com a linguagem do aparelho. Caso a linguagem do aparelho esteja em português, esses textos serão respectivamente "OK" e "CANCELAR".

Em resumo, as duas formas são muito úteis e em cada situação podemos escolher entre uma ou outra. O alert possui mais opções de configurações em relação ao toast, mas em alguns casos simplesmente queremos enviar um aviso ao usuário sem a necessidade de nenhuma ação posterior, nesse caso o toast é a melhor opção.

Inserindo informações

Voltando ao nosso projeto, então somente depois que o registro for inserido no banco de dados atualizaremos a lista, não queremos correr o risco de exibir para o usuário um item que não foi realmente salvo.

Vamos começar a implementação desse código na CadastroActivity. Abra o arquivo CadastroActivity e navegue até o listener do clique do botão.

Vamos mexer dentro do if de verificação se as caixas de texto estão vazias porque é aqui onde um item é salvo. Comovamos usar outra estratégia para salvar os dados, um banco dedados, o código antigo não faz mais sentido, por isso apague essas duas linhas de código:

val prod = Produto(produto, qtd.toInt(), valor.toDouble(), imageBitMap)

produtosGlobal.add(prod)

Não usaremos mais essa variável produtosGlobal e também não precisaremos mais criar o objeto prod aqui, isso porque agora vamos inserir essas informações direto no banco de dados. O seu if deve ficar assim:

if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()) {

//Aqui implementaremos o código para inserir o produto no banco de dados txt\_produto.text.clear()

txt\_qtd.text.clear()

txt\_valor.text.clear()

}

Agora vamos fazer o código de inserção no banco de dados. Primeiro, vamos montar o comando insert e, em seguida, verificar o retorno da função para ver se o item foi inserido com sucesso no banco. Para isso insira o seguinte código dentro do if :

database.use {

val idProduto = insert("Produtos",

"nome" to produto,

16

"quantidade" to qtd,

"valor" to valor.toDouble(),

"foto" to imageBitMap

)

if (idProduto != -1L) {

toast("Item inserido com sucesso")

txt\_produto.text.clear()

txt\_qtd.text.clear()

txt\_valor.text.clear()

} else {

toast("Erro ao inserir no banco de dados")

}

}

Esse código vai precisar dos seguintes imports :

import org.jetbrains.anko.db.insert

import org.jetbrains.anko.toast

Só temos mais uma coisa para nos preocuparmos agora: para gravar a foto no banco de dados, estamos enviando diretamente o objeto Bitmap e, apesar de não ocorrer nenhum erro de compilação, com certeza teremos um erro na execução desse projeto, porque o tipo BLOB que usamos na coluna de foto do banco de dados não suporta um Bitmap diretamente. Devemos enviar um vetor de bytes em Kotlin um objeto do tipo ByteArray.

Por isso, antes de enviar ao banco, devemos transformar esse Bitmap em ByteArray e depois, quando resgatarmos as informações do banco, devemos fazer o processo contrário, transformar o ByteArray em Bitmap.

Extensões

Vamos implementar essas duas funções como extensões, esse é um recurso muito legal da linguagem Kotlin. Extensão é um recurso com o qual eu posso adicionar uma funcionalidade em um objeto sem precisar utilizar uma herança de classes, isto é, eu não preciso ter uma nova classe que herda a classe pai para adicionar funcionalidades à classe pai, basta criar uma extensão!

Para o nosso projeto, vamos criar 2 extensões, uma para classe Bitmap e uma para classe ByteArray. A extensão da classe Bitmap terá a função de devolver um ByteArray e a extensão da classe ByteArray vai

17

fazer o contrário, devolver um Bitmap. Desta forma, vamos poder usar as extensões em qualquer objeto do tipo Bitmap ou ByteArray.No nosso projeto, já existe um arquivo chamado Utils.kt. Vamos utilizá-lo para criar as extensões dentro dele, então abra o arquivo.

Vamos fazer primeiro a extensão do Bitmap. Ela se chamará toByteArray() para deixar claro que ela transformará um Bitmap em ByteArray. Para declarar a extensão, devemos criar uma função com o nome da classe que receberá a extensão como prefixo. Vamos à prática, vou criar uma função de extensão para classe Bitmap com o nome toByteArray e ela retornará um objeto ByteArray :

fun Bitmap.toByteArray(): ByteArray {

}

Agora vamos implementar o conteúdo dessa função. Nesse contexto, o uso da palavra this representa o Bitmap que chamou a função. Então, quando usamos this aqui estamos nos referindo ao próprio objeto.

Essa função será dividida em 3 partes: primeiro, vamos declarar um objeto do tipo ByteArrayOutputStream, que será responsável por fazer a transformação do Bitmap em ByteArray :

val stream = ByteArrayOutputStream()

Em seguida, faremos a compressão da imagem dentro do objeto stream. Para isso, vamos passar 3 parâmetros: o formato da imagem, a qualidade da compressão, sendo 0 como qualidade máxima e 100 como o máximo de compressão, e por fim o objeto stream :

//comprimindo a imagem

this.compress(android.graphics.Bitmap.CompressFormat.PNG, 0, stream) Por último, vamos retornar o resultado da função toByteArray do objeto stream :

return stream.toByteArray()

Confira a implementação completa:

fun Bitmap.toByteArray(): ByteArray {

val stream = ByteArrayOutputStream()

//comprimindo a imagem

this.compress(android.graphics.Bitmap.CompressFormat.PNG, 0, stream)

//transformando em um array de caracteres

return stream.toByteArray()

}

Você precisará dos seguintes imports:

18

import android.graphics.Bitmap

import java.io.ByteArrayOutputStream

A próxima extensão é mais simples, ela será um objeto do tipo ByteArray e devemos retornar um objeto do tipo Bitmap. Para isso, existe o método estático decodeByteArray da classe BitmapFactory, que faz a decodificação de um ByteArray para Bitmap :

fun ByteArray.toBitmap() :Bitmap{

return BitmapFactory.decodeByteArray(this, 0, this.size);

}

Aqui você precisará do seguinte import :

import android.graphics.BitmapFactory

Implemente essas duas funções no arquivo Utils.kt. A ideia de implementar essas funções dentro desse arquivo é simplesmente para deixar nosso código um pouco mais modularizado. Como essas duas funções são de extensão, elas poderiam estar em qualquer arquivo do projeto, no entanto, isso dificultaria sua manutenção.

Agora volte à CadastroActivity e localize a função de insert no banco de dados. Precisamos mudá-la para enviar um ByteArray ao banco, e não mais um Bitmap diretamente.

database.use {

val idProduto = insert("Produtos",

"nome" to produto,

"quantidade" to qtd,

"valor" to valor.toDouble(),

"foto" to imageBitMap?.toByteArray() //Acrescentamos a chamada a função de extensão

)

...

//restante do código

E pronto! A parte de inserir no banco de dados está completa! Confira o código completo do listener do botão:

//definição do ouvinte do botão

btn\_inserir.setOnClickListener {

//pegando o valor digitado pelo usuário

val produto = txt\_produto.text.toString()

val qtd = txt\_qtd.text.toString()

val valor = txt\_valor.text.toString()

//verificando se o usuário digitou algum valor

if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()) {

database.use {

val idProduto = insert("Produtos",

"nome" to produto,

"quantidade" to qtd,

"valor" to valor.toDouble(),

"foto" to imageBitMap?.toByteArray() //Acrescentamos a chamada à função de extensão

)

19

if (idProduto != -1L) {

toast("Item inserido com sucesso")

txt\_produto.text.clear()

txt\_qtd.text.clear()

txt\_valor.text.clear()

} else {

toast("Erro ao inserir no banco de dados")

}

}

} else {

txt\_produto.error = if (txt\_produto.text.isEmpty()) "Preencha o nome do produto" else null txt\_qtd.error = if (txt\_qtd.text.isEmpty()) "Preencha a quantidade" else null

txt\_valor.error = if (txt\_valor.text.isEmpty()) "Preenchao valor" else null

}

}

Selecionando registros do banco de dados

A essa altura, nosso aplicativo já cadastra as informações no banco de dados! Mas você deve ter percebido que, ao fechar o aplicativo e abrir novamente, os dados não se mantêm, a lista vem vazia! O que houve? Será que a inserção no banco de dados não foi bem-sucedida?

Não é nada disso, os dados estão sim sendo armazenados no banco de dados, no entanto não estamos fazendo a busca das informações na hora de abrir o aplicativo!

O que devemos fazer é acessar o banco de dados e buscar todos os registros cadastrados quando o aplicativo for aberto, assim, quando o usuário abrir o aplicativo ele encontrará todos os itens que ele previamente cadastrou.

Para buscar dados do banco, utilizamos o método select, que, essencialmente, faz uma busca nos registros da tabela especificada. Veremos como podemos usar o select para algumas das buscas mais comuns.

Para fazer uma busca em todos os registros de uma tabela, basta usar o comando select e passar o nome da tabela em String. Por exemplo:

select("produtos")

Realmente simples! Caso precise filtrar a consulta, podemos usar o comando whereArgs e passar os argumentos que queremos. Vamos supor que você queira trazer todos os produtos com quantidade menor que 2, o comando ficaria:

select("produtos").whereArgs( "quantidade < {qtd}", "qtd" to 2 )

No entanto, até esse ponto esses comandos ainda não executam no banco de dados, ou seja, estamos simplesmente montando a consulta. Precisamos executar essa consulta no banco, e para isso existe a função

20

exec. Ela é responsável por executar, de fato, a consulta no banco de dados e, a partir daí, podemos trabalhar seus resultados.

Veja no exemplo:

select("produtos").whereArgs( "quantidade < {qtd}", "qtd" to 2 ).exec {

//Aqui conseguimos trabalhar com o resultado da consulta ao banco }

No nosso projeto, precisamos de uma busca em todos os registros do banco de dados, então nosso comando será simplesmente:

select("produtos").exec {

}

Precisaremos também tratar o resultado da busca dentro da função exec. Para isso, existem 3 tipos de funções para converter o resultado da busca:

• parseSingle : converte exatamente 1 linha; usamos essa função quando sabemos que nossa consulta retornará exatamente 1 registro.

• parseOpt : converte 1 ou nenhuma linha; diferente do parseSingle, o parseOpt pode ter um retorno nulo caso não tenha nenhum registro.

• parseList : converte vários registros em uma lista, para consultas que retornaram mais de 1 resultado.

Esses 3 métodos recebem como parâmetro um objeto rowParser, que tem como objetivo converter cada coluna da tabela em um tipo do Kotlin.

Entenda o objeto rowParser como um conversor, ele vai pegar os resultados do banco de dados e converter para algum tipo de objeto que faça sentido para nosso aplicativo. No nosso caso, oque faz sentido é a criação de um objeto do tipo Produto, então será esse rowParser a ser implementado que converterá uma linha do banco de dados em um objeto do tipo Produto!

A implementação do rowParser é muito simples, basta declararmos variáveis na sequência que retornará a consulta, e depois podemos usá-las para criação do objeto.

Observe o código para criação do nosso rowParser :

val parser = rowParser {

id: Int,

nome: String,

quantidade: Int,

valor:Double,

21

foto:ByteArray? ->

//Colunas do banco de dados

//Montagem do objeto Produto com as colunas do banco

Produto(id, nome, quantidade, valor, foto?.toBitmap() )

}

*Aqui encontramos novamente a notação de flecha -> que, nesse contexto, está fazendo a separação dos parâmetros recebidos do corpo da função.*

Em seguida, podemos utilizar o método parseList pois a nossa consulta retornará mais de um registro, retornará uma lista:

val listaProdutos = parseList(parser)

Temos uma lista de produtos com todos os registros que estão salvos no banco de dados! A partir daí, podemos inserir essa lista inteira no adaptador e assim exibir todos os itens para o usuário. Para adicionar uma lista inteira ao adaptador, podemos usar o método addAll, que funciona de forma semelhante ao add, porém não recebe um único item, mas sim uma lista ou um array.

Veja no código:

adapter.clear() //Limpando os itens antigos do adapter

adapter.addAll(listaProdutos)

Agora vamos implementar esse código no método onResume da MainActivity, porque é lá que está o código para preencher alista. Então devemos modificar o código atual para acessar o banco de dados e preencher a lista a partir do banco.

O código do método ficará assim, observe que o código está todo comentado e algumas coisas ainda se mantêm em relação ao código anterior:

database.use {

//Efetuando uma consulta no banco de dados

select("produtos").exec {

//Criando o parser que montará o objeto produto

val parser = rowParser {

id: Int,

nome: String,

quantidade: Int,

valor:Double,

foto:ByteArray? ->

//Colunas do banco de dados

22

//Montagem do objeto Produto com as colunas do banco Produto(id, nome, quantidade, valor, foto?.toBitmap() ) }

//criando a lista de produtos com dados do banco

var listaProdutos = parseList(parser)

//limpando os dados da lista e carregando as novas informações

adapter.clear()

adapter.addAll(listaProdutos)

//efetuando a multiplicação e soma da quantidade e valor val soma = listaProdutos.sumByDouble { it.valor \* it.quantidade }

//formatando em formato moeda

val f = NumberFormat.getCurrencyInstance(Locale("pt", "br")) txt\_total.text = "TOTAL: ${ f.format(soma)}"

}

}

Para esse código, você usará os seguintes imports :

import org.jetbrains.anko.db.parseList

import org.jetbrains.anko.db.rowParser

import org.jetbrains.anko.db.select

Agora execute novamente o aplicativo e veja se ele carrega os dados que foram cadastrados no banco!

Removendo um registro

Para remover um registro do banco de dados, usamos o método delete, passando o nome da tabela e os argumentos para remoção. Isto é, quando vamos deletar um registro do banco temos que especificar qual registro é esse, e isso pode ser feito de várias formas. Por exemplo, imagine que eu queira deletar todos os produtos com quantidade maior que 5, faria um código assim:

delete("produtos", "quantidade > {qtd}", "qtd" to 5)

Perceba que a String que eu passei de argumento foi quantidade > {qtd}, em que quantidade é o nome da colunada tabela e {qtd} indica que o valor aqui será trocado por um argumento de mesmo nome qtd, por isso, no terceiro parâmetro eu passo o argumento com o valor: "qtd" to 5.

No nosso projeto, vamos deletar um item com base no id do produto. Quando o usuário clicar e segurar algum item da lista, vamos deletá-lo com base em seu id, então nosso código ficará assim:

23

delete("produtos", "id = {id}", "id" to idProduto)

Claro que a variável idProduto deve guardar o id do produto que será deletado da lista. Para deixar o código um pouco melhor, vamos separar o ato de deletar em uma função dentro da MainActivity. Essa função deve receber uma variável idProduto:Int com o id do produto e executar o comando de delete no banco.

A implementação da função ficará assim:

fun deletarProduto(idProduto:Int) {

database.use {

delete("produtos", "id = {id}", "id" to idProduto)

}

}

Agora vamos chamar essa função no listener de clique longo da lista. Localize esse listener que foi implementado no método onCreate. Dentro do listener, basta chamar a função deletarProduto passando o id do item que foi selecionado:

deletarProduto(item.id)

Confira como ficará o código completo do listener:

//definição do ouvinte da lista para clicks longos

list\_view\_produtos.setOnItemLongClickListener {

adapterView: AdapterView<\*>, view1: View, i: Int, l: Long ->

//buscando o item clicado

val item = produtosAdapter.getItem(i)

//removendo o item clicado da lista

produtosAdapter.remove(item)

//deletando do banco de dados

deletarProduto(item.id)

true

}

Podemos ainda colocar um toast para avisar o usuário de que o item foi deletado com sucesso.

24

...

//deletando do banco de dados

deletarProduto(item.id)

toast("item deletado com sucesso")

true

}

Atualizando um registro

Nosso projeto está completo assim como planejamos. No entanto, não temos um modo de atualizar algum registro que já foi cadastrado. Caso o usuário perceba que digitou algum dado errado, ele terá que apagar o registro e cadastrar de novo.

Esta seção é destinada à atualização de um registro no banco dedados, mas eu não farei a atualização direto no projeto em que vínhamos trabalhando. Vou explicar como podemos atualizar um registro cadastrado no banco de dados e deixarei de desafio para você implementar a funcionalidade de editar um registro!

Para atualizar um registro do banco, a biblioteca nos disponibiliza a função update, que vai executar uma atualização no banco de dados de acordo com os parâmetros passados a ela. Podemos dividir a função update em 3 partes, sendo a primeira onde passamos o nome da tabela que será atualizada e os valores a serem atualizados, por exemplo:

update("Usuarios", "nome" to "Januário")

Nesse exemplo, a tabela se chama Usuarios e estamos atualizando a coluna nome com o valor Januário. Mas se executarmos esse comando dessa forma, todos os registros do banco de dados seriam atualizados com o nome Januário e muito provavelmente não é isso que você vai querer!

Vamos à segunda parte do comando, a passagem dos argumentos. Os argumentos vão dizer qual registro queremos atualizar. Não são todos, mas um específico. Então especificamos qual registro vamos atualizar através dos argumentos que pode ser um id específico, um nome específico ou até mesmo um conjunto de dados. Para passagens desses argumentos, usamos a função whereArgs na qual podemos especificar um ou mais argumentos. Veja um exemplo:

update("Usuarios", "nome" to "Januário").whereArgs(" id = {id}", "id" to 14)

Aqui estamos atualizando com base no id. Perceba que a função whereArgs recebe 2 argumentos: primeiro, o argumento em si, nesse caso id = {id} e aqui o {id} é um curinga que será substituído com o valor passado no segundo argumento; já no segundo, passamos o valor do curinga: "id" to 14, nesse caso estamos definindo que o id que será atualizado é o que possui valor 14.

Por fim, devemos chamar a função exec(), que executa de fato a atualização no banco de dados, retornando um Int com a quantidade de registros afetados pela atualização.

25

Agora que você sabe como funciona a função update, que tal tentar implementar no projeto de lista de compras? Será um bom desafio para seu aprendizado.

Resumo

Chegamos ao final de mais um projeto! Foi um projeto mais complexo, com muito mais conteúdo e aprendizado. Aqui, exploramos a biblioteca Anko para manipular um banco de dados local em SQLite.

26