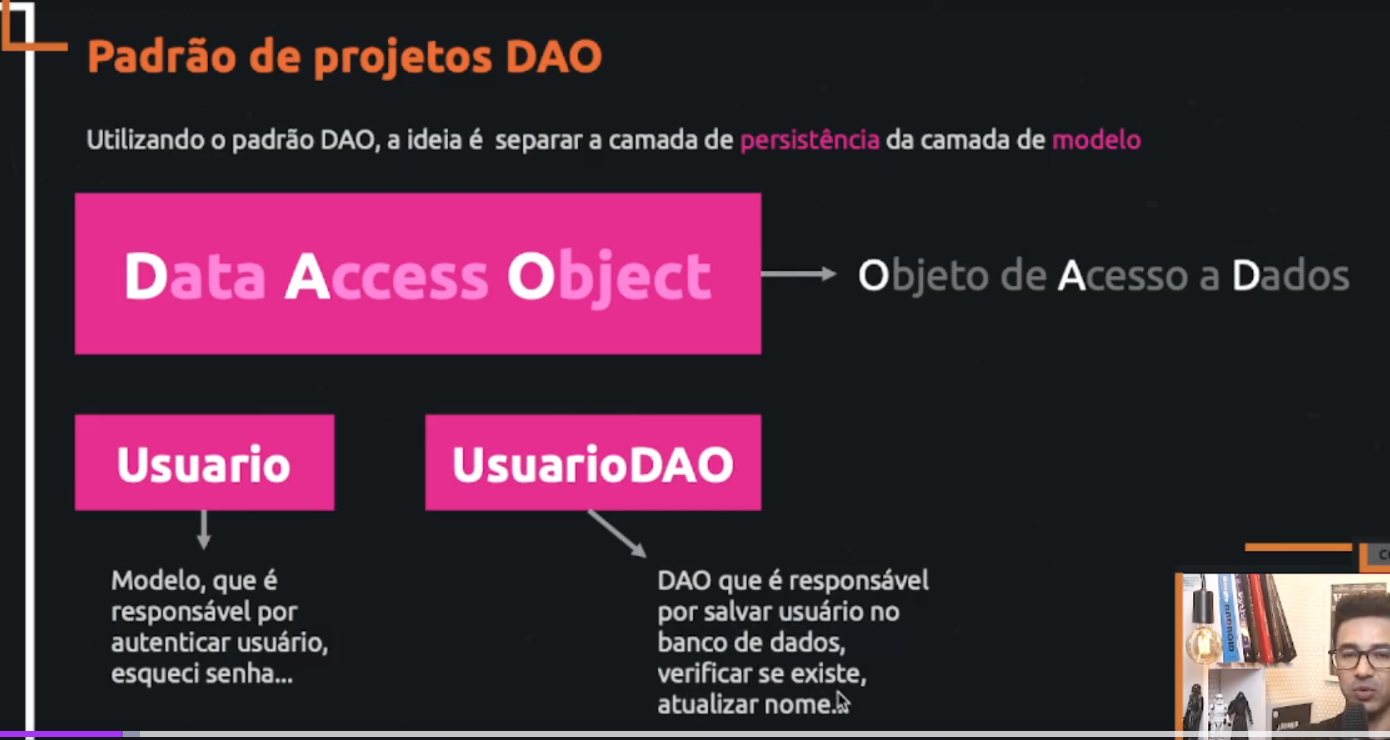


Modelos = Classes

Repare as classes a baixo.



Resumindo aqui



A persistência de dados é salvar os dados, e ai temos 2 classes diferentes, uma que vai ser para autenticar, fazer os métodos e a outra DAO é que esta ligada diretamente aos dados no banco de dados.

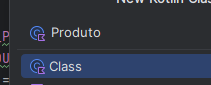
Pq não fazer tudo junto? Para não misturar regras de negocio com regras de acesso ao banco de dados.

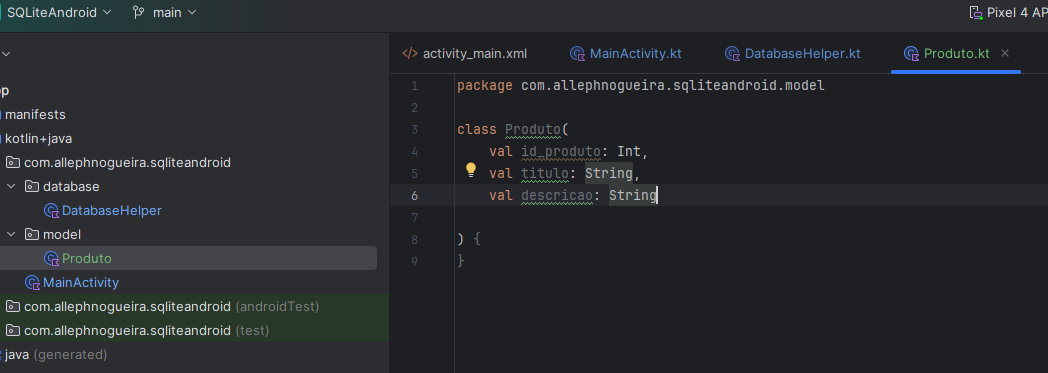
Vantagens de usar o DAO



# Vamos a pratica

Primeiro vamos criar um pacote chamado modelo.

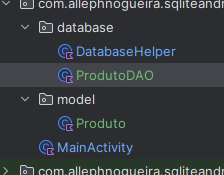
Como estamos trabalhando com produtos, vamos criar um Kotlin class de Produtos. 



# PRODUTO DAO

Já criamos nossa classe model de produto, agora vamos criar nosso produtoDAO

Vamos dentro do pacote database e vamos criar la dentro, lembra que ele vai ser responsável pelos dados do banco de dados.

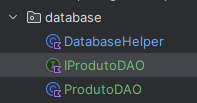


Ele vai ser responsável por salvar os dados do banco de dados, então vamos criar os métodos que vai salvar os dados

package com.allephnogueira.sqliteandroid.database  
  
import android.content.Context  
  
class ProdutoDAO(context: Context) {  
 */\*\* Essa classe vai ser responsavel por fazer o CRUD de PRODUTOS  
 \* Estamos encadeando os metodo para ser mais facil de usar depois, tanto escrita quanto leitura. \*\*/* val escrita = DatabaseHelper(context).*writableDatabase* // Esse metodo serve para escrita (UPDATE, DELETE, INSERT)  
 val leitura = DatabaseHelper(context).*readableDatabase* // Esse metodo serve para leitura (SELECT)  
  
  
   
  
  
}

# Interface para força a criar os métodos

Vamos criar essa interface para forçar a criar os métodos



package com.allephnogueira.sqliteandroid.database  
  
import com.allephnogueira.sqliteandroid.model.Produto  
  
interface IProdutoDAO {  
  
 */\*\* Estamos passando um produto inteiro para quase todos os metodos, menos para o remover pq só vamos precisar do ID  
 \* O Listar algo não vamos precisar passar parametros, apenas ele vai retornar uma Lista de produto. \*/* fun salvar(produto: Produto) : Boolean  
 fun atualizar(produto: Produto) : Boolean  
 fun remover(idProduto: Int) : Boolean  
 fun listar() : List<Produto>  
  
  
}

# ProdutoDAO Implementando a interface e seus métodos

Criando os métodos que vai ser chamado depois

class ProdutoDAO(context: Context) : IProdutoDAO {  
 */\*\* Essa classe vai ser responsavel por fazer o CRUD de PRODUTOS  
 \* Estamos encadeando os metodo para ser mais facil de usar depois, tanto escrita quanto leitura. \*\*/* val escrita = DatabaseHelper(context).*writableDatabase* // Esse metodo serve para escrita (UPDATE, DELETE, INSERT)  
 val leitura = DatabaseHelper(context).*readableDatabase* // Esse metodo serve para leitura (SELECT)  
  
  
   
 override fun salvar(produto: Produto): Boolean {  
  
 val comandoSQL =  
 "INSERT INTO ${DatabaseHelper.TABELA\_PRODUTOS} (id\_produto, titulo, descriacao) VALUES (null, '${produto.titulo}', 'Descrição...');"  
  
 try {  
 escrita.execSQL(comandoSQL) // writable para escrita  
 Log.i("info\_db", "Produto cadastrado com sucesso.")  
  
 } catch (e: Exception) {  
 e.printStackTrace()  
 return false */\*\* Se der tudo errado ele retorna um false \*\*/* }  
  
 return true  
}

override fun atualizar(produto: Produto): Boolean {  
 */\*\* Basicamente vamos atualizar o nome do produto onde o id seja igual a 1  
 \* Poderiamos deixar o usuario definir também qual vai ser o id, mas isso é mais para frente.  
 \*/* val codigoSQL = "UPDATE produtos SET titulo = '${produto.titulo}' WHERE id\_produto = 1 ;"  
  
 try {  
 escrita.execSQL(codigoSQL)  
 Log.i("info\_db", "Sucesso ao atualizar")  
 }catch (e: Exception) {  
 e.printStackTrace()  
 Log.i("info\_db", "Erro ao atualizar")  
 return false  
 }  
 return true  
}

override fun remover(idProduto: Int): Boolean {  
 */\*\* Remover um produto onde o ID seja igual a 01  
 \* Futuramente vamos fazer um recyclerView para poder remover um produto. \*\*/* val codigoSQL = "DELETE FROM produtos WHERE id\_produto = $idProduto"  
  
 try {  
 escrita.execSQL(codigoSQL)  
 Log.i("info\_db", "Sucesso ao remover")  
 }catch (e: Exception) {  
 e.printStackTrace()  
 Log.i("info\_db", "Erro ao remover")  
 return false  
 }  
 return true  
}

override fun listar(): List<Produto> {  
  
 val listaProdutos = *mutableListOf*<Produto>() // Criando uma lista para salvar os produtos  
  
 val codigoSQL = "SELECT \* FROM ${DatabaseHelper.TABELA\_PRODUTOS}"  
 val cursor = leitura.rawQuery(codigoSQL, null)  
  
  
 while ( cursor.moveToNext() ) {  
 val idProduto = cursor.getInt(0)  
 val titulo = cursor.getString(1)  
 val descricao = cursor.getString(2)  
  
  
 */\*\* Resumindo vamos percorrer todo nosso banco de dados e todos os produtos que ele encontrar ele vai salvar dentro da nossa lista \*/* listaProdutos.add( // Adicionando os produtos dentro da lista.  
 Produto(idProduto, titulo, descricao)  
 )  
  
 }  
  
 return listaProdutos // Retornando a lista de produtos  
  
}

}

# Transferindo os códigos para o DAO

Vamos pegar os métodos que estão la na MainActivity e vamos transferir tudo pra ca, para a classe ter a responsabilidade de salvar os dados.

No MainActivity vamos lá no método que salva e vamos fazer dessa forma, vamos instanciar o DAO, vamos criar nosso produto, e vamos chamar a classe DAO passando nosso novo produto

Esse -1 seria uma estratégia para quando a gente não quer usar nenhum valor.

Resumindo quando o usuário clicar em salvar, ele vai chamar o método salvar que vai transferir os dados para o DAO salvar no banco de dados.

private fun removerProduto() {  
  
 val idProduto = binding.editProduto.*text*.toString().*toInt*()  
 val produtoDAO = ProdutoDAO(this)  
  
  
 produtoDAO.remover(idProduto)  
  
  
  
 }  
  
 private fun atualizar() {  
 */\*\* Caso tenha duvida olhe o metodo de salvar \*\*/* val titulo = binding.editProduto.*text*.toString()  
 val produtoDAO = ProdutoDAO(this)  
 val produto = Produto (  
 -1, titulo, "descricao produto..."  
 )  
  
 produtoDAO.atualizar(produto)  
  
 }  
  
 private fun listar() {  
*/\*\* val comandoSQL = "SELECT \* FROM ${DatabaseHelper.TABELA\_PRODUTOS};"  
  
  
 val cursor = bancoDeDados.readableDatabase  
 .rawQuery(  
 comandoSQL,  
 null) // Readable Database é para fazer a leitura do banco de dados.  
  
  
 /\*\*Dessa forma em vez de lembrar qual é cada indice, vamos passar o nome e o proprio kotlin vai passar o indice para a gente  
 \* Agora ele vai retornar para a gente os dados que estao dentro de produto, titulo e descricao  
 \* ATENCAO AQUI ESTA COM ERRO PQ DIGITEI O NOME DA TABELA ERRADO.\*/  
  
// val indiceId = cursor.getColumnIndex("${DatabaseHelper.TABELA\_PRODUTOS}") // Aqui estamos pegando o valor ja definido para evitar erros.  
// val indiceTitulo = cursor.getColumnIndex("${DatabaseHelper.TITULO}")  
// val indiceDescricao = cursor.getColumnIndex("${DatabaseHelper.DESCRICAO}")  
  
 while ( cursor.moveToNext() ) {  
 /\*\* Cursor = Onde nosso mouse esta  
 \* Exemplo imagina os registros/linhas e nosso mouse em cima da linha 1  
 \* Então o cursor é isso é onde vai esta nosso mouse  
 \* moveToNext ele vai sair da primeira casa e vai avançar para a segunda, toda vez que ele avançar ele vai gerar um TRUE OU FALSE  
 \* Se for false ele para, pq é sinal que não tem mais registros  
 \*  
 \* Por isso estamos fazendo um while, pq enquanto for verdadeiro ele vai avançando...  
 \* MoveToNext ele começa sempre no primeiro registro e vai andando para baixo...  
 \*/  
  
 /\*\* Recuperar as colunas/Registros  
 \* getInt porque estamos recuperando o ID e sabemos que o ID vai ser sempre um Int  
 \* No parametro vamos colocar a casa da coluna que vai se comportar como se fosse um array  
 \* Temos 3 colunas  
 \* id\_produto = 0 / titulo = 1 / descrição = 2  
 \* \*/  
  
 val idProduto = cursor.getInt(0)  
 val tituloProduto = cursor.getString(1)  
 val descricaoProduto = cursor.getString(2)  
  
 Log.i("info\_db", "id: $idProduto - titulo: $tituloProduto - descrição: $descricaoProduto")  
  
  
 Log.i("info\_db", "Posição do cursor = ${cursor.position}") // Contagem para saber quantos registros ele vai pegar...  
 }  
  
\*/  
  
  
 /\*\* Agora sim, lembrando que ele vai retorna uma lista de produtos \*/* val produtoDAO = ProdutoDAO(this)  
 val listaProdutos = produtoDAO.listar()  
  
 if (listaProdutos.*isNotEmpty*()) {  
 */\*\* Conferindo se a lista esta retornando alguma coisa, pq ela pode vir vazia \*/* listaProdutos.*forEach* **{** Log.i("info\_db", "${**it**.id\_produto} - ${**it**.titulo}")  
 **}** }  
 }  
  
 fun salvar() {  
 */\*\* Os metodos foram transferidos para o produtoDAO para poder organizar melhor o codigo.  
  
 val produtoDigitado =  
 binding.editProduto.text.toString() // Capturando oque o usuario digitou.  
 val comandoSQL =  
 "INSERT INTO ${DatabaseHelper.TABELA\_PRODUTOS} (id\_produto, titulo, descriacao) VALUES (null, '$produtoDigitado', 'Descrição...');"  
  
 try {  
 bancoDeDados.writableDatabase.execSQL(comandoSQL) // writable para escrita  
 Log.i("info\_db", "Produto cadastrado com sucesso.")  
  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 \*\*/  
  
  
 /\*\* Capturando o titulo que o usuario escreveu \*/* val titulo = binding.editProduto.*text*.toString()  
  
 */\*\* Instanciando o produtoDAO e passando o contexto \*/* val produtoDAO = ProdutoDAO(this) // Devemos passar o contexto  
  
 */\*\* Criando nosso produto com os atributos \*\*/* val produto = Produto(  
 -1, // Isso é uma estrategia para quando nao queremos definir valor; Atenção o proprio banco de dados vai incrementar de 1 em 1  
 titulo, // Estamos pegando o titulo que o proprio usuario escreveu  
 "Produto novo com nota fiscal"  
  
 )  
  
 */\*\* Salvando o nosso produto criado no banco de dados, utilizando o metodo que criamos para isso \*\*/* produtoDAO.salvar(produto) // Aqui agora chamamos o metodo passando o produto que criamos.  
 }  
  
}