APP WORLD

PERSISTÊNCIA
DE DADOS LOCAIS

11A

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 PERSISTÊNCIA DE DADOS LOCAIS

1.1 Biblioteca Room

Durante o desenvolvimento de aplicações Android, nos deparamos com situações em que devemos armazenar dados localmente, seja porque não precisamos persisti-lo em um servidor remoto, seja porque precisamos que o usuário utilize a aplicação mesmo sem conexão de rede. Lembre-se que nem sempre o usuário estará em alguma localidade com rede disponível, então, permitir o uso da aplicação offline é uma habilidade que o desenvolvedor Mobile deve possuir.

Neste capítulo vamos aprender como persistir dados localmente utilizando a biblioteca Room, que oferece uma camada de abstração sobre o SQLite livrando o desenvolvedor da construção de códigos extensos. O Room faz boa parte do trabalho pesado. Vamos começar!

1.1.1 Implementação do Room no projeto Android

Para começarmos a trabalhar com o Room vamos criar um projeto no Android Studio com o nome “Meus Contatos”. Faça tudo do jeito que você já sabe 😉.

Antes de começarmos, precisamos adicionar as dependências da biblioteca Room em nosso projeto, então, abra o arquivo “build.gradle (Module: app)” e adicione na sessão “dependencies” as seguintes linhas, conforme a listagem abaixo:

```
dependencies {

    implementation 'androidx.core:core-ktx:1.8.0'
    implementation platform('org.jetbrains.kotlin:kotlin-bom:1.8.0')
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.3.1'
    implementation 'androidx.activity:activity-compose:1.5.1'
    implementation platform('androidx.compose:compose-bom:2022.10.00')
    implementation 'androidx.compose.ui:ui'
    implementation 'androidx.compose.ui:ui-graphics'
    implementation 'androidx.compose.ui:ui-tooling-preview'
    implementation 'androidx.compose.material3:material3'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation 'androidx.test.espresso:espresso-
core:3.5.1'
    androidTestImplementation platform('androidx.compose:compose-
bom:2022.10.00')
    androidTestImplementation 'androidx.compose.ui:ui-test-junit4'
```

```
    debugImplementation 'androidx.compose.ui:ui-tooling'
    debugImplementation 'androidx.compose.ui:ui-test-manifest'

    // Room DEPENDENCIES
    implementation 'androidx.room:room-runtime:2.5.2'
    annotationProcessor 'androidx.room:room-compiler:2.5.2'
    kapt 'androidx.room:room-compiler:2.5.2'

}
```

Código-fonte 1 – Dependências do Room em build.gradle
Fonte: Elaborado pelo autor (2023)

Antes de sincronizarmos as dependências, precisamos adicionar o plugin do processador de anotações do Kotlin (Kapt), então, no início do arquivo “build.gradle (Module: app)” adicione a seguinte linha na sessão “plugins”, conforme mostra a listagem abaixo:

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'kotlin-kapt'
}
```

Código-fonte 2 – Inclusão do plugin kapt em build.gradle
Fonte: Elaborado pelo autor (2023)

Esse plugin é necessário para que possamos utilizar as anotações do Room, que veremos mais adiante.

Com as alterações efetuadas, não se esqueça de sincronizar o Gradle, para que os downloads sejam feitos.

A seguir, temos o código fonte da tela da nossa aplicação, portanto, substitua todo o conteúdo do arquivo “MainActivity.kt” pela listagem abaixo:

```
package br.com.fiap.meuscontatos

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material3.Button
import androidx.compose.material3.Card
import androidx.compose.material3.CardDefaults
import androidx.compose.material3.Checkbox
```

```
import androidx.compose.material3.ExperimentalMaterial3Api
import androidx.compose.material3.Icon
import androidx.compose.material3.IconButton
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.input.KeyboardCapitalization
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import br.com.fiap.meuscontatos.ui.theme.MeusContatosTheme

class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MeusContatosTheme {
        Surface(
          modifier = Modifier.fillMaxSize(),
          color = MaterialTheme.colorScheme.background
        ) {
          Column {
            ContatosScreen()
          }
        }
      }
    }
  }
}

@Composable
fun ContatosScreen() {

  var nomeState = remember {
    mutableStateOf("")
  }

  var telefoneState = remember {
    mutableStateOf("")
  }

  var amigoState = remember {
    mutableStateOf(false)
  }

  Column {
    ContatoForm(
      nome = nomeState.value,
      telefone = telefoneState.value,
      amigo = amigoState.value,
      onNomeChange = {
        nomeState.value = it
      },
```

```
        onTelefoneChange = {
          telefoneState.value = it
        },
        onAmigoChange ={
          amigoState.value = it
        }
      )
      ContatoList()
    }
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ContatoForm(
  nome: String,
  telefone: String,
  amigo: Boolean,
  onNomeChange: (String) -> Unit,
  onTelefoneChange: (String) -> Unit,
  onAmigoChange: (Boolean) -> Unit
) {
  Column(
    modifier = Modifier.padding(16.dp)
  ) {
    Text(
      text = "Cadastro de contatos",
      fontSize = 24.sp,
      fontWeight = FontWeight.Bold,
      color = Color(
        0xFFE91E63
      )
    )
    Spacer(modifier = Modifier.height(8.dp))
    OutlinedTextField(
      value = nome,
      onValueChange = { onNomeChange(it) },
      modifier = Modifier.fillMaxWidth(),
      label = {
        Text(text = "Nome do contato")
      },
      keyboardOptions = KeyboardOptions(
        keyboardType = KeyboardType.Text,
        capitalization = KeyboardCapitalization.Words
      )
    )
    Spacer(modifier = Modifier.height(8.dp))
    OutlinedTextField(
      value = telefone,
      onValueChange = { onTelefoneChange(it) },
      modifier = Modifier.fillMaxWidth(),
      label = {
        Text(text = "Telefone do contato")
      },
      keyboardOptions = KeyboardOptions(
        keyboardType = KeyboardType.Phone
      )
    )
    Spacer(modifier = Modifier.height(8.dp))
    Row(
      verticalAlignment = Alignment.CenterVertically,
      modifier = Modifier.fillMaxWidth()
```

```
    ) {
      Checkbox(checked = amigo, onCheckedChange = {
        onAmigoChange(it)
      })
      Text(text = "Amigo")
    }
    Spacer(modifier = Modifier.height(16.dp))
    Button(
      onClick = { /*TODO*/ },
      modifier = Modifier.fillMaxWidth()
    ) {
     Text(
       text = "CADASTAR",
       modifier = Modifier.padding(8.dp)
     )
    }
  }
}

@Composable
fun ContatoList() {
  Column(modifier = Modifier
    .fillMaxSize()
    .padding(16.dp)
    .verticalScroll(rememberScrollState())
  ) {
    for (i in 0..10){
      ContatoCard()
      Spacer(modifier = Modifier.height(4.dp))
    }
  }
}

@Composable
fun ContatoCard() {
  Card(
    modifier = Modifier.fillMaxWidth(),
    colors = CardDefaults.cardColors(
      containerColor = Color.LightGray
    )
  ) {
    Row(
      verticalAlignment = Alignment.CenterVertically
    ) {
      Column(modifier = Modifier
        .padding(8.dp)
        .weight(2f)) {
        Text(
          text = "Nome do Contato",
          fontSize = 24.sp,
          fontWeight = FontWeight.Bold
        )
        Text(
          text = "8888-9999",
          fontSize = 16.sp,
          fontWeight = FontWeight.Bold
        )
        Text(
          text = "Amigo",
          fontSize = 16.sp,
          fontWeight = FontWeight.Bold
```

```
            )
        }
        IconButton(onClick = { /*TODO*/ }) {
            Icon(
                imageVector = Icons.Default.Delete,
                contentDescription = ""
            )
        }
    }
  }
}
```

Código-fonte 3 – Layout inicial da aplicação
Fonte: Elaborado pelo autor (2023)

O que temos de diferente nesta implementação é o uso do atributo “verticalScroll” do *composable* “Column” que permitirá que o conteúdo da coluna seja rolável caso a matéria seja maior do que a coluna. Além disso, utilizamos o composable “IconButton”, que é um botão que exibe apenas um ícone.

Execute a aplicação no emulador. O resultado esperado deverá se parecer com a figura “Layout inicial da aplicação”:

Figura 1 – Layout inicial da aplicação
Fonte: Elaborado pelo autor (2023)

1.1.2 Estrutura do projeto

Para que o projeto fique organizado, vamos criar os seguintes pacotes:

- **model**: para armazenar nossas classes de objeto;
- **database**: para armazenar as classes relacionados com o banco de dados.

No pacote “database” criaremos mais dois pacotes:

- **dao**: para guardar as interfaces que representam as instruções que vamos executar no banco de dados SQLite;
- **repository**: para guardar as classes utilizadas como fonte de dados da aplicação.

A estrutura de pacotes do projeto deverá se parecer com a figura “Estrutura de pacotes do Room”:

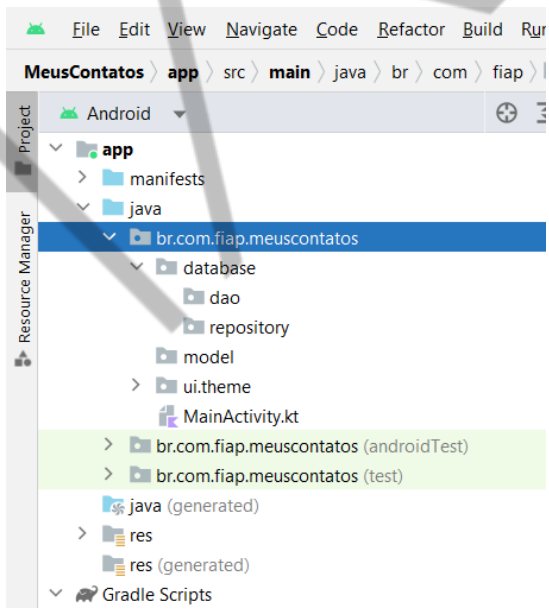

Figura 2 – Estrutura de pacotes para o Room
Fonte: Elaborado pelo autor (2023)

1.1.3 Criação do modelo Contato

No pacote “model” vamos criar uma classe chamada “Contato”, que representará cada um dos nossos contatos. O código da classe “Contato” está disponível na listagem abaixo:

```
package br.com.fiap.meuscontatos.model

data class Contato(
  val id: Long = 0,
  val nome: String = "",
  val telefone: String = "",
  val amigo: Boolean = false
)
```

Código-fonte 4 – Classe de dados Contato
Fonte: Elaborado pelo autor (2023)

A classe “Contato” é uma classe de dados, ou seja, possui apenas atributos que representa os dados de um objeto. Agora precisamos anotar essa classe de modo que o Room saiba que deve gerenciar esta classe e criar uma entidade relacionada no SQLite. Após as alterações, o código da classe “Contato” deverá se parecer com a listagem abaixo:

```
package br.com.fiap.meuscontatos.model

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "tbl_contato")
data class Contato(
  @PrimaryKey(autoGenerate = true) val id: Long = 0,
  val nome: String = "",
  val telefone: String = "",
  @ColumnInfo(name = "is_amigo") val amigo: Boolean = false
)
```

Código-fonte 5 – Classe Contato com anotações do Room
Fonte: Elaborado pelo autor (2023)

As alterações em destaque no código acima foram as seguintes:

@Entitiy: essa anotação indica ao Room que uma entidade para a classe “Contado” deverá ser criada no SQLite. A propriedade “table_name” indica que o nome da tabela no SQLite deverá ser “tbl_contato”. Se “table_name” for omitido será criado uma tabela com o mesmo nome da classe.

@PrimaryKey: indica que o atributo “id” da classe “Contato” será a chave-primária da tabela. O atributo “autoGenerate” indica que o identificador deverá ser gerado automaticamente, ou seja, não há a necessidade de cria-lo no momento de instanciação da classe “Contato”.

@ColumnInfo: essa anotação do atributo “amigo” permite a alteração de como o campo será criado na tabela no SQLite, neste caso estamos indicando que o campo deverá se chamar “is_amigo”. Se esta anotação for omitida, o campo terá o mesmo nome do atributo da classe.

1.1.4 Criação da interface dao (Data Access Object)

Para a implementação do Room na aplicação, é necessário criarmos uma Interface que abstrairá os métodos CRUD (Create, Read, Update e Delete) da aplicação para uma entidade. Assim, vamos criar a Interface “ContatoDao” no pacote “dao” do projeto. Nesta interface vamos adicionar os métodos que indicarão ao Room quais operações de CRUD deverão ser implementados. O código da Interface é apresentado na listagem abaixo:

```
package br.com.fiap.meuscontatos.database.dao

import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update
import br.com.fiap.meuscontatos.model.Contato

@Dao
interface ContatoDao {

  @Insert
  fun salvar(contato: Contato): Long

  @Update
  fun atualizar(contato: Contato): Int

  @Delete
  fun excluir(contato: Contato): Int

  @Query("SELECT * FROM tbl_contato WHERE id = :id")
  fun buscarContatoPeloId(id: Int): Contato

  @Query("SELECT * FROM tbl_contato ORDER BY nome ASC")
  fun listarContatos(): List<Contato>

}
```

Código-fonte 6 – Interface ContatoDao
Fonte: Elaborado pelo autor (2023)

No código da listagem “Interface ContatoDao” estamos definindo os métodos necessários para as operações CRUD no banco de dados. Para isso, utilizamos as seguintes anotações:

@Dao: indica que essa interface deve ser utilizada pelo Room.

@Insert: essa anotação indica ao Room que deve ser criado um método de inclusão de contato no banco de dados. O retorno da função “salvar” será o novo “id” gerado na tabela “tbl_contato” no banco de dados, por isso o retorno do tipo “Long”. A instrução “INSERT” do banco de dados será implementada pelo Room.

@Update: indica ao Room que deverá ser implementado um método para a atualização do contato no bando de dados. Essa função retorna à quantidade de registros que foram atualizados, por isso o retorno do tipo “Int”. A instrução “UPDATE” no banco de dados será implementada pelo Room.

@Query: indica ao Room que um método de consulta deverá ser implementado. Essa anotação permite a execução de qualquer estrutura da instrução SQL “SELECT”. Nesta anotação é necessário escrevermos a instrução “SELECT” que será executada no banco de dados. No código da listagem “Interface ContatoDao”, temos duas funções com essa anotação: a “buscarContatoPeloId” que deve resultar em apenas um contato, por isso o retorno do tipo “Contato”, e a função “listarContatos”, que terá como retorno uma lista de contatos, o que resultará no retorno de um objeto do tipo “List” de contatos.

1.1.5 Criação da classe que representa o banco de dados

Vamos criar uma classe que representará a instância do banco de dados na aplicação. Essa classe deverá retornar a instância do banco de dados para que os métodos CRUD possam ser executados. Deste modo, vamos criar uma classe com o nome “ContatoDb” no pacote “dao” do nosso projeto. O código dessa classe pode ser visualizado na listagem “Classe ContatoDb” abaixo:

```
package br.com.fiap.meuscontatos.database.dao

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import br.com.fiap.meuscontatos.model.Contato

@Database(entities = [Contato::class], version = 1)
abstract class ContatoDb : RoomDatabase() {

  abstract fun contatoDao(): ContatoDao

  companion object {

    private lateinit var instance: ContatoDb

    fun getDatabase(context: Context): ContatoDb {
      if (!::instance.isInitialized) {
        instance = Room
          .databaseBuilder(
            context,
            ContatoDb::class.java,
            "contato_db"
```

```
            )
            .allowMainThreadQueries()
            .fallbackToDestructiveMigration()
            .build()
        }
        return instance
    }
  }
}
```

Código-fonte 7 – Classe ContatoDb
Fonte: Elaborado pelo autor (2023)

A classe “ContatoDb” herda a classe “RoomDatabase”, que é abstrata. Deste modo, a classe “ContataoDb” dever ser abstrata também. Ainda nesta classe, temos uma função abstrata chamada “contatoDao" que é do tipo “ContatoDao”. Essa função é necessária para termos acesso aos métodos CRUD nela descritos.

A função “getDatabase()” e o atributo “instance” se encontram dentro de um bloco “companion object”, que as tornam estáticas. Fazemos isso para aplicar o conceito de “Singleton” no retorno da função “getDatabase”, para garantir que sempre entregaremos aos consumidores desta classe uma única instância do banco de dados; isso garante consistência e economia de recursos do dispositivo do usuário. Observe que antes de retornarmos a instância, verificamos se ela já está iniciada ou não.

Para a criação da instância do banco de dados, utilizamos a função “databaseBuilder”, onde fornecemos os seguintes parâmetros:

- **context**: que representa a nossa aplicação.
- **ContatoDb::class.java**: é a instância da classe que representa o banco de dados.
- **“contato_db”:** é uma String com o nome do banco de dados.

Além disso, adicionamos as funções:

- **allowMainThreadQueries:** permitir que a persistência de dados ocorra no mesmo processo que gerencia a IU.
- **fallbackToDestructiveMigration:** destrói o banco de dados e o recria a cada nova implementação.
- **build**: cria a instancia do banco de dados.

1.1.6 Criação do repositório de contatos

O repositório desempenha um papel muito importante na estrutura do projeto, já que ele será responsável por acessar os métodos CRUD da aplicação. Portanto, vamos criar no pacote “repository” uma classe com o nome “ContatoRepository”. O código dessa classe pode ser obtido na listagem “Classe ContatoRepository”, logo abaixo:

```
package br.com.fiap.meuscontatos.database.repository

import android.content.Context
import br.com.fiap.meuscontatos.database.dao.ContatoDb
import br.com.fiap.meuscontatos.model.Contato

class ContatoRepository(context: Context) {

  private val db = ContatoDb.getDatabase(context).contatoDao()

  fun salvar(contato: Contato): Long {
    return db.salvar(contato)
  }

  fun atualizar(contato: Contato): Int {
    return db.atualizar(contato)
  }

  fun excluir(contato: Contato): Int {
    return db.excluir(contato)
  }

  fun listarContatos(): List<Contato> {
    return db.listarContatos()
  }

  fun buscarContatoPeloId(id: Int): Contato {
    return db.buscarContatoPeloId(id)
  }

}
```

Código-fonte 8 – Classe ContatoRepository
Fonte: Elaborado pelo autor (2023)

A classe “ContatoRepository” recebe o parâmetro “context” da aplicação. Isso será necessário para quando formos obter uma instância do banco de dados.

Também injetamos o banco de dados, através do atributo “db”, que receberá a instância do nosso banco de dados.

Os métodos da classe “ContatoRepository” utilizam a instância do banco de dados para efetuarmos o CRUD no banco de dados, que foram declarados na Interface “ContatoDao”. Agora é só consumirmos tudo isso. Vamos lá!

1.1.7 Gravando nosso primeiro contato

Chegou o momento de ver o nosso trabalho recompensado! Vamos gravar o nosso primeiro contato. Abra o arquivo “MainActivity.kt” e na função “ContatoForm” acrescente as instruções necessárias para obtermos uma instância do repositório. Seu código deverá se parecer com a listagem “Instanciação do objeto ContatoRepository” abaixo:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ContatoForm(
  nome: String,
  telefone: String,
  amigo: Boolean,
  onNomeChange: (String) -> Unit,
  onTelefoneChange: (String) -> Unit,
  onAmigoChange: (Boolean) -> Unit
) {
  // obter instância do repositório
  val context = LocalContext.current
  val contatoRepository = ContatoRepository(context)
  Column(
    modifier = Modifier.padding(16.dp)
  ) {
    Text(
. . . trecho de código omitido
```

Código-fonte 9 – Instanciação do objeto ContatoRepository
Fonte: Elaborado pelo autor (2023)

Para criarmos uma instância do objeto “ContatoRepository” precisamos do contexto da aplicação, que está sendo obtido através da instrução “LocalContext.current”.

A partir de agora vamos implementar o clique do botão. Seu código deverá se parecer com a listagem abaixo:

```
Button(
  onClick = {
    val contato = Contato(
      nome = nome,
      telefone = telefone,
      amigo = amigo
    )
    contatoRepository.salvar(contato = contato)
  },
  modifier = Modifier.fillMaxWidth()
) {
  Text(
    text = "CADASTAR",
    modifier = Modifier.padding(8.dp)
  )
}
. . . trecho de código omitido
```

Código-fonte 10 – Implementando o código do botão de cadastro
Fonte: Elaborado pelo autor (2023)

O que fizemos nesta implementação foi criar um objeto “Contato” referenciado pela variável “contato”, com os dados que estão preenchidos no formulário. Em seguida, chamamos a função salvar do objeto “contatoRepository” passando o objeto “contato” como argumento.

IMPORTANTE: O banco de dados “contato_db” será criado em nossa aplicação durante a primeira tentativa de interação com o banco. Após a primeira interação, a aplicação passará a utilizar o banco que foi criado.

Antes de executarmos a aplicação, vamos verificar o local onde o banco de dados fica gravado no dispositivo Android. Com o emulador rodando, clique na aba “Device File Explorer”, de acordo com a figura “Device File Explorer”. Navegue no sistema de arquivos do dispositivo até a pasta de dados do aplicativo, de acordo com a figura “Pasta de dados do aplicativo”:

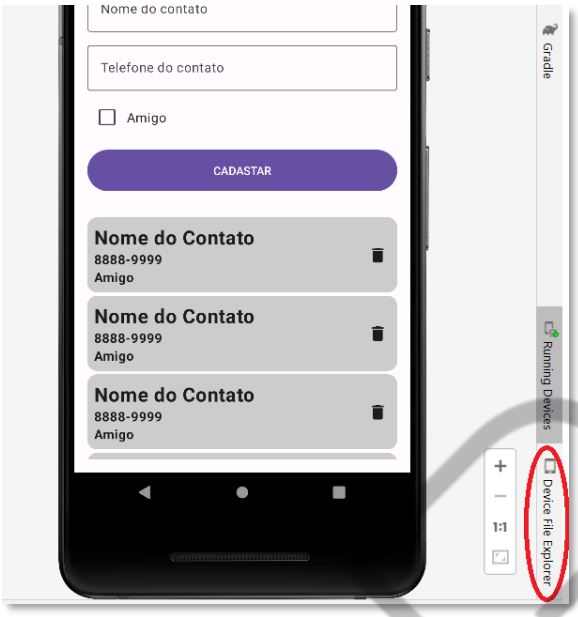

Figura 3 – Device File Explorer
Fonte: Elaborado pelo autor (2023)

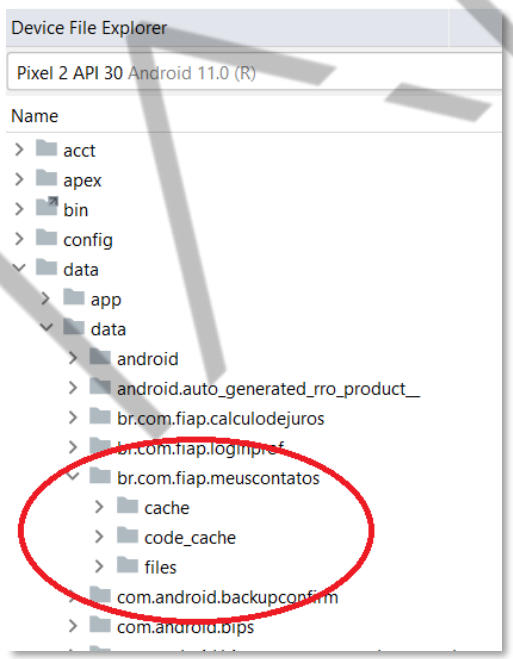

Figura 4 – Pasta de dados do aplicativo
Fonte: Elaborado pelo autor (2023)

Execute a aplicação no emulador, preencha o formulário com os dados de um amigo e pressione o botão “CADASTRAR”. Aparentemente nada aconteceu, mas se você não recebeu uma mensagem de erro o registro foi armazenado no banco de dados.

Em “Device File Explorer”, clique com o botão direito do mouse na pasta “br.com.fiap.meuscontatos” e clique na opção “Synchronize”, conforme a figura “Synchronize”. O resultado deverá se parecer com a figura “Banco de dados criado”.

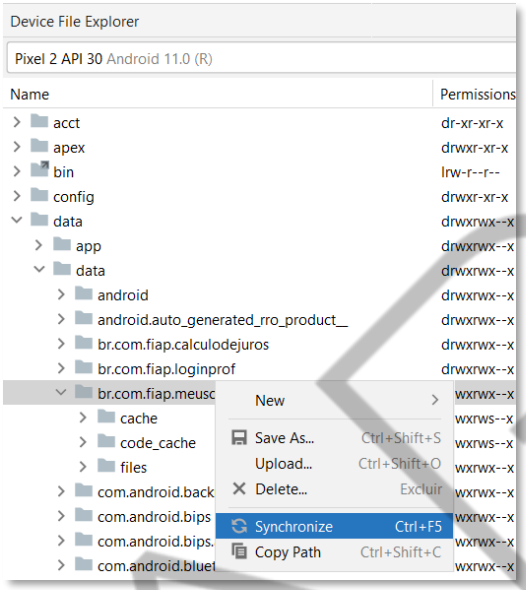

Figura 5 – Synchronize
Fonte: Elaborado pelo autor (2023)

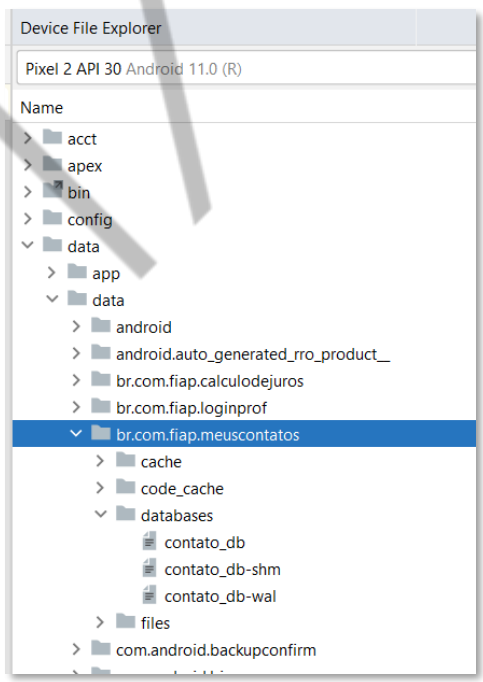

Figura 6 – Banco de dados criado
Fonte: Elaborado pelo autor (2023)

Também podemos explorar o banco de dados que foi criado no dispositivo clicando na aba “App Inspection” na parte inferior do Android Studio, conforme a figura “App Inspection”:

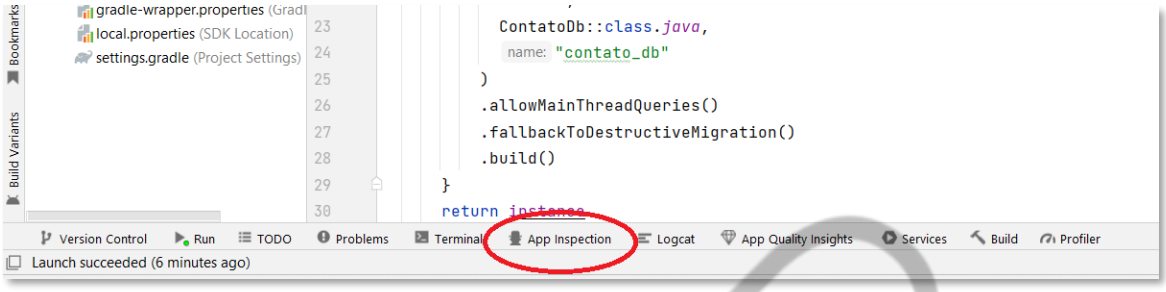

Figura 7 – App Inspection
Fonte: Elaborado pelo autor (2023)

Em seguida, dê um duplo clique no nome da tabela, conforme a figura “Visualizando os dados da tabela”. À direita veremos os dados do amigo que acabamos de cadastrar. Marque a opção “Live updates”, para acompanhar os registros sendo atualizados na tabela.

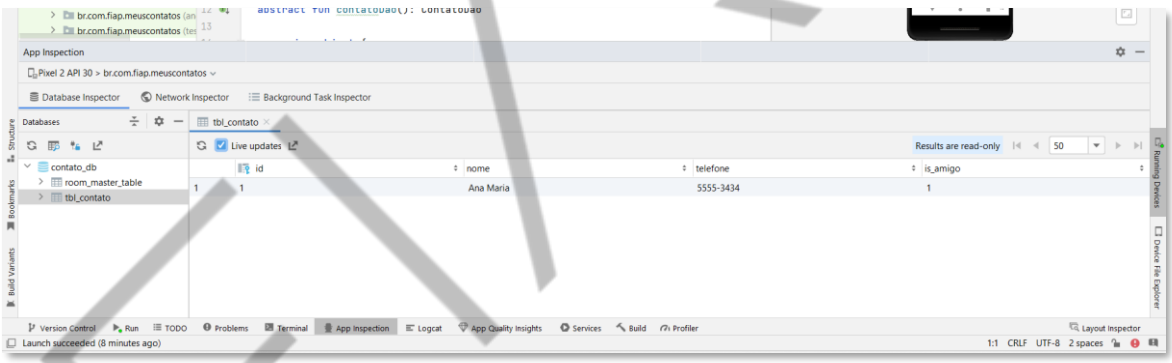

Figura 8 – Visualizando os dados da tabela
Fonte: Elaborado pelo autor (2023)

1.1.8 Listando os contatos

Nossa aplicação já permite o cadastro de novos contatos, agora precisamos exibir os contatos à medida que vão sendo cadastrados. Para isso, vamos criar uma variável de estado que será responsável por atualizar a lista de contatos da aplicação, além de criar uma instância da classe “ContatoRepository”. Após as alterações, a função “ContatosScreen” deve se parecer com a listagem abaixo:

```
@Composable
fun ContatosScreen() {

  val context = LocalContext.current
  val contatoRepository = ContatoRepository(context)

  var nomeState = remember {
    mutableStateOf("")
  }

  var telefoneState = remember {
    mutableStateOf("")
  }

  var amigoState = remember {
    mutableStateOf(false)
  }

  var listaContatosState = remember {
    mutableStateOf(contatoRepository.listarContatos())
  }
. . . trecho omitido
```

Código-fonte 11 – Variável de estado da lista de contatos
Fonte: Elaborado pelo autor (2023)

Em seguida, precisamos passar a "listaContatosState" para a função "ContatoList". O código desta função deve se parecer com a listagem abaixo:

```
@Composable
fun ContatoList(contatos: List<Contato>) {
  Column(
    modifier = Modifier
      .fillMaxSize()
      .padding(16.dp)
      .verticalScroll(rememberScrollState())
  ) {
    for (contato in contatos) {
      ContatoCard(contato)
      Spacer(modifier = Modifier.height(4.dp))
    }
  }
}
```

Código-fonte 12 – Passando a lista de contatos para a função ContatoCard
Fonte: Elaborado pelo autor (2023)

Precisamos ajustar a chamada para "ContatoList" na função "ContatosScreen" para que a lista seja passada como argumento. A função "ContatosScreen" deve se parecer com a listagem abaixo após as alterações:

```
. . . trecho de código omitido
Column {
  ContatoForm(
    nome = nomeState.value,
    telefone = telefoneState.value,
    amigo = amigoState.value,
    onNomeChange = {
      nomeState.value = it
    },
    onTelefoneChange = {
      telefoneState.value = it
    },
    onAmigoChange = {
      amigoState.value = it
    }
  )
  ContatoList(listaContatosState.value)
}
. . . trecho de código omitido
```

Código-fonte 13 – Passando a lista de contatos para a função ContatoScreen
Fonte: Elaborado pelo autor (2023)

Note que a função “ContatoList” recebe a lista de contatos como argumento. A lista é iterada pelo laço de repetição “for” que passa o contato para a função “ContatoCard”. Deste modo, vamos ajustar a função “ContatoCard” que deverá implementar o código da listagem abaixo:

```
@Composable
fun ContatoCard(contato: Contato) {
  Card(
    modifier = Modifier.fillMaxWidth(),
    colors = CardDefaults.cardColors(
      containerColor = Color.LightGray
    )
  ) {
    Row(
      verticalAlignment = Alignment.CenterVertically
    ) {
      Column(
        modifier = Modifier
          .padding(8.dp)
          .weight(2f)
      ) {
        Text(
          text = contato.nome,
          fontSize = 24.sp,
          fontWeight = FontWeight.Bold
        )
        Text(
          text = contato.telefone,
          fontSize = 16.sp,
          fontWeight = FontWeight.Bold
        )
        Text(
          text = if (contato.amigo) "Amigo" else "Contato",
          fontSize = 16.sp,
```

```
            fontWeight = FontWeight.Bold
          )
        }
        IconButton(onClick = { /*TODO*/ }) {
          Icon(
            imageVector = Icons.Default.Delete,
            contentDescription = ""
          )
        }
      }
    }
}
```

Código-fonte 14 – Função ContatoCard usando o objeto contato
Fonte: Elaborado pelo autor (2023)

Execute a aplicação no emulador e verifique se os contatos já cadastrados são listados, mas ao cadastrar um novo contato ele não irá aparecer na lista, pois não estamos atualizando a variável “listaContatosState” que está sendo observada.

Vamos refatorar a função “ContatoForm” de modo a atualizar a variável “listaContatosState” durante a inclusão de um novo contato. O código com a nova implementação deve se parecer com a listagem abaixo:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ContatoForm(
  nome: String,
  telefone: String,
  amigo: Boolean,
  onNomeChange: (String) -> Unit,
  onTelefoneChange: (String) -> Unit,
  onAmigoChange: (Boolean) -> Unit,
  atualizar: () -> Unit
) {
  // obter instância do repositório
  val context = LocalContext.current
  val contatoRepository = ContatoRepository(context)
. . . trecho omitido
```

Código-fonte 15 – Inclusão da função lambda atualizar em ContatoForm
Fonte: Elaborado pelo autor (2023)

Acrescentamos uma função lambda como parâmetro para a função “ContatoForm”, de modo que, ao chamarmos o formulário, possamos passar uma função para atualização da lista de contatos. Portanto, a implementação do clique do botão deve ser alterada para que fique de acordo com a listagem abaixo:

```
. . . trecho de código omitido
Button(
  onClick = {
    val contato = Contato(
      nome = nome,
      telefone = telefone,
      amigo = amigo
    )
    contatoRepository.salvar(contato = contato)
    atualizar()
  },
  modifier = Modifier.fillMaxWidth()
) {
  Text(
    text = "CADASTAR",
    modifier = Modifier.padding(8.dp)
  )
}
. . . trecho de código omitido
```

Código-fonte 16 – Uso da função atualizar durante clique do botão
Fonte: Elaborado pelo autor (2023)

Para finalizar, precisamos passar a função de atualização durante a chamada da função de composição “ContatoForm”. A função “ContatosScreen” deverá se parecer com a listagem abaixo:

```
. . . trecho de código omitido
Column {
  ContatoForm(
    nome = nomeState.value,
    telefone = telefoneState.value,
    amigo = amigoState.value,
    onNomeChange = {
      nomeState.value = it
    },
    onTelefoneChange = {
      telefoneState.value = it
    },
    onAmigoChange = {
      amigoState.value = it
    },
    atualizar = {
      listaContatosState.value = contatoRepository.listarContatos()
    }
  )
  ContatoList(listaContatosState.value)
}
. . . trecho de código omitido
```

Código-fonte 17 – Passagem da função atualizar
Fonte: Elaborado pelo autor (2023)

Vamos testar nossa aplicação e verificar se a lista é atualizada durante a inclusão de novos contatos.

1.1.9 Exclusão de contatos

Na lista de contatos há um botão para excluir o contato, sendo assim vamos implementar esta funcionalidade. O código da função “ContatoCard” deverá se parecer com a listagem abaixo:

```
@Composable
fun ContatoCard(
  contato: Contato,
  context: Context,
  atualizar: () -> Unit
) {
  Card(
    modifier = Modifier.fillMaxWidth(),
    colors = CardDefaults.cardColors(
      containerColor = Color.LightGray
    )
  ) {
    Row(
      verticalAlignment = Alignment.CenterVertically
    ) {
      Column(
        modifier = Modifier
          .padding(8.dp)
          .weight(2f)
      ) {
        Text(
          text = contato.nome,
          fontSize = 24.sp,
          fontWeight = FontWeight.Bold
        )
        Text(
          text = contato.telefone,
          fontSize = 16.sp,
          fontWeight = FontWeight.Bold
        )
        Text(
          text = if (contato.amigo) "Amigo" else "Contato",
          fontSize = 16.sp,
          fontWeight = FontWeight.Bold
        )
      }
      IconButton(onClick = {
        val contatoRepository = ContatoRepository(context)
        contatoRepository.excluir(contato)
        atualizar()
      }) {
        Icon(
          imageVector = Icons.Default.Delete,
          contentDescription = ""
        )
      }
    }
  }
}
```

Código-fonte 18 – ContatoCard implementando exclusão
Fonte: Elaborado pelo autor (2023)

A função “ContatoList” também deverá ser ajustada para passar os argumentos da função “ContatoCard”, que deverá se parecer com a listagem abaixo:

```
@Composable
fun ContatoList(contatos: List<Contato>, atualizar: () -> Unit) {
  Column(
    modifier = Modifier
      .fillMaxSize()
      .padding(16.dp)
      .verticalScroll(rememberScrollState())
  ) {
    for (contato in contatos) {
      ContatoCard(contato, context = LocalContext.current, atualizar)
      Spacer(modifier = Modifier.height(4.dp))
    }
  }
}
```

Código-fonte 19 – Passagem da função atualizar em ContatoList
Fonte: Elaborado pelo autor (2023)

Deste modo, a função “ContatosScreen” que chama a função “ContatoList” deverá fornecer a função atualizar. O código completo da função “ContatosScreen” pode ser consultado na listagem abaixo:

```
@Composable
fun ContatosScreen() {

  val context = LocalContext.current
  val contatoRepository = ContatoRepository(context)

  var nomeState = remember {
    mutableStateOf("")
  }

  var telefoneState = remember {
    mutableStateOf("")
  }

  var amigoState = remember {
    mutableStateOf(false)
  }

  var listaContatosState = remember {
    mutableStateOf(contatoRepository.listarContatos())
  }

  Column {
    ContatoForm(
      nome = nomeState.value,
      telefone = telefoneState.value,
      amigo = amigoState.value,
      onNomeChange = {
        nomeState.value = it
      },
      onTelefoneChange = {
        telefoneState.value = it
      },
      onAmigoChange = {
```

```
            amigoState.value = it
        },
        atualizar = {
            listaContatosState.value = contatoRepository.listarContatos()
        }
    )
    ContatoList(listaContatosState.value){
        listaContatosState.value = contatoRepository.listarContatos()
    }
  }
}
```

Código-fonte 20 – Função ContatosScreen implementando a função atualizar
Fonte: Elaborado pelo autor (2023)

Pronto! Execute a aplicação em um emulador e clique no ícone de exclusão. O contato deverá ser excluído e a lista atualizada. Bom trabalho! 😉

2 DESAFIO

Agora que você já sabe como interagir com o banco de dados, que tal implementar a atualização dos dados de um contato? Pra fazer isso, você pode tornar o Card responsável por exibir os dados do contato clicável através do modificador “modifier.clickable”, que criará uma função que vai ser disparada quando o *card* for clicado.

Lembre-se: já implementamos o método “atualizar” no repositório.

CONCLUSÃO

Neste capítulo aprendemos que nem sempre a persistência de dados poderá ocorrer de forma on-line, dependendo de uma conexão de rede. A persistência de dados local pode ser uma opção poderosa para essas situações.

Portanto, saber como manipular dados localmente é uma habilidade que todo desenvolvedor Android deve dominar.

REFERÊNCIAS

DEVELOPERS. **Scroll**. 2023. Disponível em: <https://developer.android.com/jetpack/compose/touch-input/pointer-input/scroll>. Acesso em: 7 jul. 2023.

DEVELOPERS. **Salvar dados em um banco de dados local usando o Room**. 2023. Disponível em: < https://developer.android.com/training/data-storage/room?hl=pt-br>. Acesso em: 7 jul. 2023.