sãojudas universidade	FTCE Faculdade de Tecnologia e Ciências Exatas

USJT - 2019 – Desenvolvimento de Sistemas para Dispositivos Móveis

Professores: Bossini

Aula: 06
Assunto: SQLite

Passo 1 (SQLite no Android – Uma API, não um SGBD) O Android oferece uma API para acesso a bases SQLite. Antes de mais nada, é importante entender que o SQLite não é um SGBD, como o MySQL ou o SQL Server, por exemplo. Trata-se apenas de uma API (um conjunto de classes, interfaces, métodos) que permitem que o sistema de arquivos seja acessado utilizando operações escritas em um dialeto SQL. Quando um método é executado, por exemplo um método que faz um INSERT na base, isso é traduzido pela API para uma operação no sistema de arquivos, guardando o registro em uma estrutura de dados eficiente, que fica na memória interna. Não há, portanto, a arquitetura cliente/servidor em que o Android seria o cliente utilizando funcionalidades de um outro serviço, como um SGBD.

Passo 2 (Definindo o esquema da base de dados) Uma boa prática para definir o conjunto de tabelas, nomes de colunas, restrições etc é escrever uma classe com constantes para armazenar esses dados. Assim eles podem ser reutilizados e uma eventual alteração será replicada em todos os pontos de código que o utilizem. Essa classe é chamada de Contrato.

A documentação oficial sugere que se crie uma classe principal como contrato e uma classe interna para cada tabela.

2.1 Veja um exemplo de classe Contrato na Listagem 2.1. Não é necessário criar essa classe no projeto. É só um exemplo inicial.

Listagem 2.1

```
public final class PessoaContract {
    //evite que ela seja instanciada acidentalmente
    private PessoaContract (){
    }
    //classe interna para representar a tabela pessoa
    public static class Pessoa implements BaseColumns{
        public static final String TABLE_NAME = "pessoa";
        public static final String COLUMN_NAME_NOME = "nome";
        public static final String COLUMN_NAME_IDADE = "idade";
        public static final String COLUMN_NAME_FONE = "fone";
        public static final String COLUMN_NAME_EMAIL = "email";
    }
}
```

2.2 Note que a classe interna Pessoa implementa a interface **BaseColumns**. Essa interface define um campo chamado **_ID** que alguns adapters do framework Android

utilizam. Embora **não seja obrigatório**, dependendo do contexto em que for utilizar sua definição, esse campo pode ser importante.

2.3 Note também que o construtor padrão foi escrito explicitamente e marcado com private. Trata-se da aplicação do **princípio do menor privilégio**: essa classe não foi feita para ser instanciada, então tiramos esse privilégio das demais, evitando que ela seja instanciada acidentalmente.

Passo 3 (Contrato para chamado e fila) Nossa aplicação possui as classes Chamado e Fila, as quais deverão ser mapeadas para o modelo relacional do SQLite. A Listagem 3.1 mostra a definição de seu contrato.

Listagem 3.1

```
public class HelpDeskContract {
    private HelpDeskContract(){
    }
    public static class FilaContract implements BaseColumns {
        public static final String TABLE_NAME = "tb_fila";
        public static final String COLUMN_NAME_ID = "id_fila";
        public static final String COLUMN_NAME_NOME = "nome";
        public static final String COLUMN_NAME_ICON_ID = "icon_id";
    }
    public static class ChamadoContract implements BaseColumns{
        public static final String TABLE_NAME = "tb_chamado";
        public static final String COLUMN_NAME_ID = "id_chamado";
        public static final String COLUMN_NAME_descricao =
"descricao";
        public static final String COLUMN_NAME_STATUS = "status";
        public static final String COLUMN_NAME_DATA_ABERTURA =
"dt_abertura";
        public static final String COLUMN_NAME_DATA_FECHAMENTO =
"dt_fechamento";
    }
}
```

Passo 4 (Escrevendo os comandos para criação da base) Os comandos para criação da base também serão constantes armazenadas na classe de Contrato.

4.1 A Listagem 4.1 mostra ambos.

Listagem 4.1

```
public static String createTableFila (){
    return String.format(
            "CREATE TABLE %s (%s INTEGER PRIMARY KEY, %s TEXT, %s
INTEGER);",
            FilaContract.TABLE_NAME,
            FilaContract.COLUMN_NAME_ID,
            FilaContract.COLUMN_NAME_NOME,
            FilaContract.COLUMN_NAME_ICON_ID
    );
}
public static String createTableChamados(){
    return String.format(
            "CREATE TABLE %s (%s INTEGER PRIMARY KEY, %s TEXT, %s
TEXT, %s INTEGER, %s INTEGER, %s INTEGER, FOREIGN KEY (%s)
REFERENCES %s (%s));",
```

```
            ChamadoContract.TABLE_NAME,
            ChamadoContract.COLUMN_NAME_ID,
            ChamadoContract.COLUMN_NAME_DESCRICAO,
            ChamadoContract.COLUMN_NAME_STATUS,
            ChamadoContract.COLUMN_NAME_DATA_ABERTURA,
            ChamadoContract.COLUMN_NAME_DATA_FECHAMENTO,
            FilaContract.COLUMN_NAME_ID,
            FilaContract.COLUMN_NAME_ID,
            FilaContract.TABLE_NAME,
            FilaContract.COLUMN_NAME_ID
    );
}
```

Passo 5 (Executando os comandos de criação do esquema) A API do Android oferece uma classe chamada SQLiteOpenHelper que, como o nome sugere, é um auxiliar para a criação de bases SQLite. Para utilizá-la, basta escrever uma subclasse e sobrescrever alguns de seus métodos. Em particular, onCreate e onUpgrade são métodos abstratos e devem ser sobrescritos. Há ainda os métodos onDowngrade e onOpen que não precisam ser implementados, caso não seja necessário.

5.1 Todos esses métodos executam automaticamente em algum momento, como define a Tabela 5.1.

Tabela 5.1

Método	Momento em que é executado
onCreate	Quando a base é criada pela primeira vez. A criação das tabelas e a população de dados inicial devem ocorrer aqui.
onUpgrade	Quando o esquema muda, informamos uma nova versão (um número inteiro que fica associado a cada versão). Esse método é chamado quando o número informado aumenta.
onDowngrade	Análogo ao onUpgrade. É chamado caso o número que representa a versão da base seja decrementado.
onOpen	Chamado quando a base é aberta para operações. Sempre depois de qualquer um dos três acima.

5.2 A Listagem 5.1 mostra a definição de uma subclasse de SQLiteOpenHelper. Note a definição do nome do esquema, a versão (começaremos com 1) e a passagem de um contexto para o construtor da super classe. O argumento null se refere a uma fábrica de cursores que poderia ser personalizada. Como usaremos a fábrica padrão, especificamos null.

Listagem 5.1

```
public class ChamadoDBHelper extends SQLiteOpenHelper {
        private static final String DB_NAME = "chamados.db";
    private static final int DB_VERSION = 1;
    ChamadoDBHelper (Context context){
        super (context, DB_NAME, null, DB_VERSION);
    }
}
```

5.3 A Listagem 5.2 mostra uma sobrescrita para o método onCreate. Ele recebe um SQLiteDatabase, uma abstração para a base de dados. Usamos o método execSQL para executar os scripts de criação das tabelas.

Listagem 5.2

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(HelpDeskContract.createTableFila());
    db.execSQL(HelpDeskContract.createTableContrato());
}
```

Passo 6 (Populando a base com alguns chamados e filas) Até então, nossa aplicação mantém os dados em meio volátil, o que quer dizer que eles são perdidos a cada vez que ela é fechada ou a cada vez que o dispositivo é desligado. A partir de agora, iremos utilizar o SQLite para armazenar os chamados e filas em meio persistente.

6.1 Na classe HelpDeskContract, declare uma lista de filas como mostra a Listagem 6.1. Inicialize-a em um bloco inicializador estático. Note que a classe fila precisa ser adaptada. Ela deve ter um campo id e um construtor que o recebe como parâmetro. Crie também métodos getter e setter para o campo id na classe Fila, antes de prosseguir.

Listagem 6.1

```
private static List <Fila> filas;
static{
    filas = new ArrayList <>();
    filas.add (new Fila (1,
"Desktops",R.drawable.ic_computer_black_24dp));
    filas.add (new Fila (2,
"Telefonia",R.drawable.ic_phone_in_talk_black_24dp));
    filas.add (new Fila (3,
"Redes",R.drawable.ic_network_check_black_24dp));
    filas.add (new Fila (4,
"Servidores",R.drawable.ic_poll_black_24dp));
```

6.2 Crie o método da Listagem 6.2. Ele devolve uma string com os comandos necessários para a inserção de cada fila na base.

Listagem 6.2

```
public static String insertFilas (){
    String template = "INSERT INTO %s (%s, %s, %s) VALUES (%d, '%s', %d);";
    StringBuilder sb = new StringBuilder ("");
    for (Fila fila : filas){
        sb.append(
                String.format(
                        Locale.getDefault(),
                        template,
                        FilaContract.TABLE_NAME,
                        FilaContract.COLUMN_NAME_ID,
                        FilaContract.COLUMN_NAME_NOME,
                        FilaContract.COLUMN_NAME_ICON_ID,
                        fila.getId(),
                        fila.getNome(),
                        fila.getIconId()
                )
        );
    }
    return sb.toString();
}
```

6.3 Para inserir os chamados, o procedimento será análogo. Primeiro, crie uma lista de chamados na classe HelpDeskContract. A seguir, insira os chamados em um bloco inicializador estático. Veja a Listagem 6.3. Note que a **ordem** de definição de blocos inicializadores estáticos no Java importa. Aquele que você definir primeiro na classe será executado primeiro. Assim, dado que a construção de chamados depende da existência de suas filas, certifique-se de definir o bloco que inicializa a lista de filas primeiro. Também é necessário adicionar id, métodos getter e setter e um novo construtor à classe Chamado.

Listagem 6.3

```
static{
    chamados = new ArrayList<>();
    chamados.add(new Chamado (
            1, filas.get(0),
            "Computador da secretária quebrado.",
            new Date(),
            null,
            "Aberto"
    ));
    chamados.add(new Chamado (
            2,
            filas.get(1),
            "Telefone não funciona.",
            new Date(),
            null,
            "Aberto")
    );
    chamados.add(new Chamado (
            3,
```

```
                filas.get(2),
                "Manutenção no proxy.",
                new Date(),
                null,
                "Aberto")
        );
        chamados.add(new Chamado (
                4,
                filas.get(3),
                "Lentidão generalizada.",
                new Date(),
                null,
                "Aberto")
        );
        chamados.add(new Chamado (5,
                filas.get(4),
                "CRM",
                new Date(),
                null,
                "Aberto")
        );
        chamados.add(new Chamado (
                6,
                filas.get(4),
                "Gestão de Orçamento",
                new Date(),
                null,
                "Aberto")
        );
        chamados.add(new Chamado (
                7,
                filas.get(2),
                "Internet com lentidão",
                new Date(),
                null,
                "Aberto")
        );
        chamados.add(new Chamado (
                8,
                filas.get(4),
                "Chatbot",
                new Date(),
                null,
                "Aberto")
        );
        chamados.add(new Chamado (
                9,
                filas.get(4),
                "Chatbot",
                new Date(),
                null,
                "Aberto")
        );
}
```

6.4 O método da Listagem 6.4 gera uma sequência de comandos INSERT, um para cada chamado existente.

Listagem 6.4

```
public static String insertChamados () {
    String template = "INSERT INTO %s (%s, %s, %s, %s, %s, %s)
VALUES (%d, '%s', %d, %d, '%s', %d);";
    StringBuilder sb = new StringBuilder("");
    for (Chamado chamado : chamados) {
        sb.append(
                String.format(
                        Locale.getDefault(),
                        template,
                        ChamadoContract.TABLE_NAME,
                        ChamadoContract.COLUMN_NAME_ID,
                        ChamadoContract.COLUMN_NAME_DESCRICAO,
                        ChamadoContract.COLUMN_NAME_DATA_ABERTURA,
                        ChamadoContract.COLUMN_NAME_DATA_FECHAMENTO,
                        ChamadoContract.COLUMN_NAME_STATUS,
                        FilaContract.COLUMN_NAME_ID,
                        chamado.getId(),
                        chamado.getDescricao(),
                        chamado.getDataAbertura().getTime(),
                        chamado.getDataFechamento() != null ?
chamado.getDataFechamento().getTime() : 0,
                        chamado.getStatus(),
                        chamado.getFila().getId()
                )
        );
    }
    return sb.toString();
}
```

6.5 Agora vamos ajustar o método onCreate de ChamadoDBHelper para que as inserções sejam realizadas. Veja a Listagem 6.5.

Listagem 6.5

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(HelpDeskContract.createTableFila());
    db.execSQL(HelpDeskContract.createTableChamados());
    db.execSQL(HelpDeskContract.insertFilas());
    db.execSQL(HelpDeskContract.insertChamados());
}
```

Passo 7 (Apagando e recriando a base) A fim de prosseguir em ambiente de teste, iremos garantir que a base seja apagada e recriada a cada vez que a aplicação inicia. É claro que isso não será feito uma vez que ela seja colocada em produção. Para isso, vamos primeiro criar constantes nas classes Contrato que armazenam os comandos para apagar as tabelas.

7.1 A Listagem 7.1 mostra as constantes para apagar as tabelas.

Listagem 7.1

```
//na classe ChamadoContract
public static final String DROP_TABLE = String.format("DROP TABLE
%s", ChamadoContract.TABLE_NAME);

//na classe FilaContract
public static final String DROP_TABLE = String.format("DROP TABLE
%s", FilaContract.TABLE_NAME);
```

7.2 A Listagem 7.2 mostra a sobrescrita do método onOpen na classe ChamadoDBHelper. Ele apaga as tabelas e as recria, chamando o método onCreate.

Listagem 7.2

```
@Override
public void onOpen(SQLiteDatabase db) {
    db.execSQL(HelpDeskContract.ChamadoContract.DROP_TABLE);
    db.execSQL(HelpDeskContract.FilaContract.DROP_TABLE);
    onCreate(db);
}
```

Passo 8 (Ajustando a lista para utilizar dados vindos da base) Agora precisamos criar uma classe responsável por disponibilizar acesso à base.

8.1 A Listagem 8.1 mostra uma classe que usa o bom e velho padrão Data Access Object. Seu nome é ChamadoDAO. Veja que ela oferece um método de busca de chamados.

Listagem 8.1

```
public class ChamadoDAO {
    private Context context;
    public ChamadoDAO(Context context) {
        this.context = context;
    }
    public List<Chamado> busca(String chave) {
        ChamadoDBHelper dbHelper = new ChamadoDBHelper(context);
        SQLiteDatabase db = dbHelper.getReadableDatabase();
        List<Chamado> chamados = new ArrayList<>();
        String command = String.format(
                Locale.getDefault(),
                "SELECT * FROM %s INNER JOIN %s ON %s.%s = %s.%s",
                HelpDeskContract.FilaContract.TABLE_NAME,
                HelpDeskContract.ChamadoContract.TABLE_NAME,
                HelpDeskContract.FilaContract.TABLE_NAME,
                HelpDeskContract.FilaContract.COLUMN_NAME_ID,
                HelpDeskContract.ChamadoContract.TABLE_NAME,
                HelpDeskContract.FilaContract.COLUMN_NAME_ID,
                HelpDeskContract.FilaContract.TABLE_NAME,
                HelpDeskContract.FilaContract.COLUMN_NAME_NOME
        );
        Cursor cursor = db.rawQuery(command, null);
        while (cursor.moveToNext()) {
            int idChamado =
                    cursor.getInt(
                            cursor.getColumnIndex(String.format(Locale.getDefault(),
                                    "%s.%s", HelpDeskContract.ChamadoContract.TABLE_NAME,
                                    HelpDeskContract.ChamadoContract.COLUMN_NAME_ID)));
            String descricao =
                    cursor.getString(cursor.getColumnIndex(
                            String.format(
                                    Locale.getDefault(),
                                    "%s.%s",
                                    HelpDeskContract.ChamadoContract.TABLE_NAME,
                                    HelpDeskContract.ChamadoContract.COLUMN_NAME_DESCRICAO
                            )
                    ));
            String status =
                    cursor.getString(
                            cursor.getColumnIndex(
                                    String.format(
                                            Locale.getDefault(),
                                            "%s.%s",
                                            HelpDeskContract.ChamadoContract.TABLE_NAME,
                                            HelpDeskContract.ChamadoContract.COLUMN_NAME_STATUS
                                    )
                            )
                    );
            long dtAbertura =
                    cursor.getLong(
                            cursor.getColumnIndex(
                                    String.format(
                                            Locale.getDefault(),
                                            "%s.%s",
                                            HelpDeskContract.ChamadoContract.TABLE_NAME,

HelpDeskContract.ChamadoContract.COLUMN_NAME_DATA_ABERTURA
                                    )
                            )
                    );
            long dtFechamento =
                    cursor.getLong(
                            cursor.getColumnIndex(
                                    String.format(
                                            Locale.getDefault(),
                                            "%s.%s",
                                            HelpDeskContract.ChamadoContract.TABLE_NAME,

HelpDeskContract.ChamadoContract.COLUMN_NAME_DATA_FECHAMENTO
                                    )
                            )
                    );
            String nomeFila = cursor.getString(
                    cursor.getColumnIndex(
                            String.format(
                                    Locale.getDefault(),
                                    "%s.%s",
                                    HelpDeskContract.FilaContract.TABLE_NAME,
                                    HelpDeskContract.FilaContract.COLUMN_NAME_NOME)
                    )
            );
            int idFila = cursor.getInt(
```

```
                    cursor.getColumnIndex(
                            String.format(
                                    Locale.getDefault(),
                                    "%s.%s",
                                    HelpDeskContract.FilaContract.TABLE_NAME,
                                    HelpDeskContract.FilaContract.COLUMN_NAME_ID
                            )
                    )
            );
            int iconId = cursor.getInt(
                    cursor.getColumnIndex(
                            String.format(
                                    Locale.getDefault(),
                                    "%s.%s",
                                    HelpDeskContract.FilaContract.TABLE_NAME,
                                    HelpDeskContract.FilaContract.COLUMN_NAME_ICON_ID
                            )
                    )
            );
            Chamado chamado = new Chamado(idChamado, new Fila(idFila, nomeFila, iconId),
                    descricao, new Date(dtAbertura), dtFechamento > 0 ? new Date(dtFechamento) :
null, status);
            chamados.add(chamado);
        }
        cursor.close();
        db.close();
        dbHelper.close();
        return chamados;
    }
}
```

8.2 Finalmente, a classe ListaChamadosActivity deve instanciar um ChamadoDAO e utilizar seu método de busca. Veja a Listagem 8.2.

Listagem 8.2

```
//variável de instância
private ChamadoDAO chamadoDAO;

//no método onCreate
chamadoDAO =  new ChamadoDAO(this);
final List <Chamado> chamados = chamadoDAO.busca(nomeFila);
```

Exercício Prático

1. Ajuste sua aplicação para que os dados sejam armazenados em uma base SQLite.

2. Faça upload do projeto no Github e gere uma release.

3. O prazo de entrega é sempre de uma semana a partir da aula em que a atividade foi proposta.