02

Persistência com SQLite

De nada adianta fazermos um formulário para adicionar novos alunos se não temos como guardá-los. É necessário persisti-los de alguma forma.

Apesar da grande e crescente capacidade dos aparelhos móveis, em particular aqueles que usam Android, ainda não é viável guardar um banco de dados complexo nesses dispositivos. O Android já vem com o SQLite, um banco de dados mais simples e leve. Usaremos justamente ele para persistir os dados no Android.

Seguindo a ideia do padrão MVC, que pode ser visto mais detalhadamente no curso de Java para Web (https://www.caelum.com.br/curso-java-web/), dividiremos a nossa aplicação em três camadas. Uma camada que já temos trabalhado bastante é a de visualização. Além dela, já criamos alguns *listeners* que foram usados justamente para controlar a integração entre as nossas *views* e as outras classes, essa camada é conhecida como ***Controller***. Para completar o padrão vamos implementar nossa camada de modelo.

A camada de **Modelo**, a qual começaremos a trabalhar agora, é onde basicamente ficarão as lógicas da aplicação. Por exemplo, implementaremos nosso acesso ao Banco de Dados e criaremos algumas classes que representam elementos importantes da aplicação.

Agora que temos uma tela com um formulário vamos criar a funcionalidade de extrair os dados preenchidos.

Na `FormularioActivity` podemos buscar os componentes de tela e então extrair os dados preenchidos:

```
EditText campoNome = (EditText) findViewById(R.id.nome);
EditText campoSite = (EditText) findViewById(R.id.site);
// ...

String nome = campoNome.getText().toString();
String site = campoSite.getText().toString();
//...
```

Em nosso sistema as informações nome, site, telefone, nota e endereco constituem os dados de um aluno. Sendo assim, como estamos utilizando orientação a objetos, vamos criar uma classe que representa um **Aluno** com alguns atributos para guardar as informações que são relevantes.

```
public class Aluno {

    private Long id;
    private String nome;
    private String telefone;
    private String endereco;
    private String site;
    private String caminhoFoto;
    private double nota;

    //getters e setters
}
```

Assim podemos lidar com todas as informações de um `aluno` em um lugar isolado. Se tivermos algum comportamento específico de um `aluno` podemos criar métodos com regras de negócio nessa entidade.

Agora somos capazes de extrair os dados da tela de formulário e colocar em um objeto do tipo **Aluno**. Na classe `FormularioActivity` que controla a tela de formulario podemos fazer:

```
EditText campoNome = (EditText) findViewById(R.id.nome);
EditText campoSite = (EditText) findViewById(R.id.site);
// ...

Aluno aluno = new Aluno();

aluno.setNome(campoNome.getText().toString());
aluno.setSite(campoSite.getText().toString());
//...
```

Como podemos ter muita funcionalidade em uma tela, a tendência é termos uma `Activity` com muitas linhas de código. Sempre que possível é uma boa prática isolar pequenas responsabilidades em outras classes para que a `Activity` faça seu trabalho interagindo com outros pequenos especialistas. Essa abordagem facilita o reuso de código (sem necessariamente fazer-se uso de herança) e também a manutenção.

Vamos criar uma nova classe responsável por extrair os dados do formulário. Essa classe será a `FormularioHelper`, que receberá no construtor a instância do `FormularioActivity` e cuidará da parte de extrair um aluno a partir dos dados dos campos:

```
public class FormularioHelper {
    private EditText nome;
    private EditText telefone;
    //... outros campos
    private Aluno aluno;

    public FormularioHelper(FormularioActivity activity) {
        nome = (EditText) activity.findViewById(R.id.nome);
        telefone = (EditText) activity.findViewById(R.id.telefone);
        //... find no restante dos campos
        aluno = new Aluno();
    }

    public Aluno pegaAlunoDoFormulario() {
        aluno.setNome(nome.getEditableText().toString());
        aluno.setEndereco(endereco.getEditableText().toString());

        //... pega outros dados

        return aluno;
    }
}
```

Agora somos capazes de fazer com que o clique no botão de salvar crie um objeto do tipo `Aluno` a partir dos dados preenchidos no formulário:

```
protected void onCreate(Bundle savedInstanceState) {
    //...

    helper = new FormularioHelper(this);

    Button botao = (Button) findViewById(R.id.botao);

    botao.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Aluno aluno = helper.pegaAlunoDoFormulario();

            Toast.makeText(Formulario.this, "Objeto aluno criado: " + aluno.getNome(),
                    Toast.LENGTH_SHORT).show();
        }
    });
}
```

Agora que fomos capazes de criar um objeto do tipo Aluno como vamos armazenar essa informação de forma que quando sairmos da aplicação e voltarmos sejamos capazes de restaurá-la?

Para nossa facilidade o `Android` vem com um banco de dados relacional chamado `SQLite`.

Mas, como vamos traduzir os dados do mundo relacional com o mundo Orientado a Objetos? Para isso, utilizaremos um padrão de projeto (*Design Pattern*) conhecido como *Data Access Object* (DAO).

Para trabalhar com ele, basta seguir a abordagem padrão: criar uma classe chamada `AlunoDAO`. A diferença será que ela deve estender de `SQLiteOpenHelper` e ter um construtor que recebe um objeto do tipo `Context`, o qual traz informações sobre a aplicação.

Precisamos receber esse objeto porque, para construir corretamente o DAO, é necessário chamar o construtor da classe mãe, passando o contexto e algumas configurações do seu banco em particular.

Logo de cara teremos dois métodos:

- `onCreate`: executado na criação do banco;

- `onUpgrade` : executado na alteração da estrutura do banco.

O `onCreate` vai ser invocado sempre que a tabela não existir, por exemplo no primeiro acesso à aplicação. Já o método `onUpgrade` é mais especifico. Sempre que quisermos atualizar a estrutura da nossa tabela, este método será chamado para executar o **SQL** necessário para realizar a atualização. No nosso caso sempre apagaremos a tabela e criaremos uma nova.

Para podermos invocar métodos para executar operações no banco como *insert*, *update* ou *delete* usaremos o objeto `SQLiteDatabase` . Podemos obter este objeto com o método `getWritableDatabase` ou `getReadableDatabase` que herdamos de `SQLiteOpenHelper` . Nos métodos `onCreate` e `onUpgrade` o recebemos como parâmetro.

O método `getReadableDatabase` , que também devolve uma referência para `SQLiteDatabase` , gastando menos recursos porém possibilitando apenas leitura.

Para criarmos o banco, precisaremos executar uma ***query*** SQL indicando as colunas da nossa tabela e seus respectivos tipos. Queremos armazenar cada atributo da classe `Aluno` nessa tabela, então teremos uma `query` assim:

```
String = "CREATE TABLE " + TABELA +
        "(id INTEGER PRIMARY KEY," +
        " nome TEXT UNIQUE NOT NULL," +
        " telefone TEXT," +
        " endereco TEXT," +
        " site TEXT," +
        " nota REAL," +
        " foto TEXT" +
        ");";
```

Essa **query** deve ser executada na criação do banco de dados, portanto, no método `onCreate` e ela preparará nossa infraestrutura para guardar alunos. O `SQLiteDatabase` tem o método `execSQL` para isso.

Agora que temos a infraestrutura criada, basta adicionarmos métodos no nosso DAO para as ações que queremos executar.

Inserindo um novo aluno

Se você já trabalha com Java, vai se lembrar bastante do estilo da API do JDBC, isto é, teremos que converter manualmente um objeto na estrutura da tabela - sem a maior parte das facilidades de uma ferramenta ORM (Object Relational Mapping).

Contudo, não será necessário escrever a **query**, já que ela vem encapsulada pelos métodos do `SQLiteDatabase` , provido pelo Android. Apenas precisaremos preencher os parâmetros.

Um objeto `SQLiteDatabase` já tem pronto o método `insert` , que recebe o nome da tabela, um parâmetro que não nos é importante no momento e o objeto a ser inserido.

Para inserir um novo registro neste banco, não precisamos de código SQL, basta usar o método `insert` e o SQL correto já será gerado, desde que tenhamos preenchido corretamente os valores a serem adicionados. Faremos o seguinte:

```
getWritableDatabase().insert(TABELA, null, valores);
```

No método `insert` , o primeiro parâmetro é uma `String` indicando a tabela onde os `valores` deverão ser inseridos. O segundo `parâmetro` também é uma `String` com o nome de uma coluna que aceita valores nulos. Você deve mandar o nome da coluna caso queira que, mesmo que um objeto esteja com todas as informações em branco, ele seja inserido no banco. Por vezes isso pode não fazer muito sentido, caso não queira inserir um aluno totalmente não preenchido no banco, basta mandar nulo neste parâmetro.

Para preencher os valores corretamente, basta usar um objeto do tipo `ContentValues` , que é bem similar a um `Map` . Nele guardamos o conteúdo da linha do registro a ser adicionada e vamos preenchê-lo usando o método `put("nomeDaColuna", valor)` . Veja um exemplo abaixo:

```
ContentValues values = new ContentValues();
values.put("nome", aluno.getNome());
values.put("nota", aluno.getNota());
//outras informações do aluno aqui
```

Para fazer a transformação de uma entidade para `ContentValues`, podemos criar um método auxiliar no próprio DAO. Há aqui uma questão de gosto. Alguns vão preferir colocar na própria classe `Aluno` um método `toContentValues`, porém quebra aí um pouco da responsabilidade da entidade.

Com nosso DAO criado podemos fazer com que o clique no botão **Salvar** persista no `SQLite` o objeto `Aluno` criado:

```
botao.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Aluno aluno = helper.pegaAlunoDoFormulario();

        AlunoDAO dao = new AlunoDAO(this);
        dao.insere(aluno);
        dao.close();
    }
});
```

Repare que invocamos o método `close` herdado da classe `SQLiteOpenHelper`. Quando nosso DAO é instanciado ele acessa o banco abrindo uma conexão. Então, quando não precisamos mais utilizá-lo devemos fechar a conexão chamando o método `close`!

O que fazer agora que o usuário da nossa aplicação conseguiu salvar o novo `Aluno`? Provavelmente ele vai querer voltar para a tela de listagem para certificar-se que o aluno foi realmente inserido. Para isso ele clicará no botão `voltar` do aparelho. Vamos melhorar a usabilidade da aplicação e fazer isso para o usuário: se quisermos programaticamente fazer o papel do botão de voltar podemos invocar o método `finish` de nossa `Activity`:

```
botao.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Aluno aluno = helper.pegaAlunoDoFormulario();

        AlunoDAO dao = new AlunoDAO(this);
        dao.insere(aluno);
        dao.close();

        finish();
    }
});
```

Listando os alunos

Agora que fomos capazes de inserir alunos no SQLite vamos alterar a `ListaAlunosActivity` para que os elementos da lista não sejam as constantes de um Array de Strings criados por nós.

Vamos adicionar ao nosso DAO um método que vai retornar a lista de alunos cadastrados no nosso sistema.

Buscar todos os alunos é ligeiramente mais complicado: teremos que lidar com o **Cursor** do banco de dados. Similarmente ao `insert`, contamos com outro método que nos permite mais liberdade na busca a ser realizada. Esse é o método `rawQuery`, que recebe a query e uma `String` contendo a cláusula *where* caso você queira aplicar algum filtro.

```
String sql = "SELECT * FROM Alunos";
Cursor cursor = getReadableDatabase().rawQuery(sql, null);
```

Como é comum buscarmos por todas as colunas, podemos definir um *array* de `String` como constante, contendo todas as colunas da nossa tabela.

Do resultado dessa busca, receberemos um `Cursor` e, com ele em mãos, basta iterar sobre suas diversas linhas, preenchendo objetos `Aluno` para cada registro do banco:

```
Aluno aluno = new Aluno();
aluno.setNome(cursor.getString(cursor.getColumnIndex("nome")));
aluno.setTelefone(cursor.getString(cursor.getColumnIndex("telefone")));
//outros campos aqui
```

Como desejamos obter uma lista de alunos precisaremos repetir o procedimento de extrair os dados do `Cursor` enquanto existir um próximo registro. Podemos atingir esse objetivo criando um laço através do método `moveToNext` do `Cursor`, saber se existe um próximo registro.

```
List<Aluno> alunos = new ArrayList<Aluno>();

while(cursor.moveToNext()) {
    Aluno aluno = new Aluno();
    aluno.setNome(cursor.getString(cursor.getColumnIndex("nome")));
    aluno.setTelefone(cursor.getString(cursor.getColumnIndex("telefone")));
    //...

    alunos.add(aluno);
}
```

Note que poderíamos devolver o próprio cursor, mas não faremos isso porque essa prática vai contra o padrão dos DAOs. A função de um DAO é encapsular e isolar a interação com o banco de dados, tornando mais fácil e simples para quem utiliza.

Muitos dos métodos que estamos trabalhando aqui podem lançar `android.database.SQLException`, que é uma exceção de *runtime*, logo não somos obrigados a fazer o bloco de *try/catch* nem de declarar seu *throws*. Em especial, ao se trabalhar com recursos que podem ficar abertos, como o `Cursor`, é boa prática deixar seu fechamento dentro de um bloco *finally*, garantindo seu fechamento.

Agora precisaremos trocar o código que carrega a lista para que use este nosso novo método `getLista`. No `onCreate` da `ListaAlunosActivity` devemos trocar o *array* de *strings* pelo método `getLista` e fazer os devidos ajustes no `ArrayAdapter`.

```
AlunoDAO dao = new AlunoDAO(this);
List<Aluno> alunos = dao.getLista();

ArrayAdapter<Aluno> adapter = new ArrayAdapter<Aluno>(this, android.R.layout.simple_list_item_1, alunos);
lista.setAdapter(adapter);
```

Note que ao retornar para a `ListaAlunosActivity` o método do ciclo de vida que será chamado para ela é o `onRestart` e depois o `onStart`. Porém o código que busca os alunos no banco de dados está dentro do `onCreate`, logo, ele não irá recarregar a lista do mesmo jeito. O que temos que fazer é usar este mesmo código dentro do método `onResume` que é o método imediatamente anterior ao usuário ver a tela.

```
public class ListaAlunosActivity extends Activity {
    //...

    @Override
    protected void onResume() {
        super.onResume();
        //código para carregar a lista aqui
    }

    //...
}
```