Desenvolvimento para dispositivos móveis

Persistência

Professor Msc. Fabio Pereira da Silva
E-mail: fabio.pereira@faculdadeimpacta.com.br

Introdução

- Cada vez que iniciamos um novo projeto de desenvolvimento de software é necessário optar pela melhor ferramenta para manipulação de dados.
- As primeiras opções para esta persistência de dados a serem suportadas por um banco de dados foram: Oracle, Informix, ProstgreSQL, MySQL e Firebird.
- Com a necessidade de agilidade, simplicidade e de fácil configuração, surgiu o SQLite.

Definição

- SQLite é uma base de dados relacional de código aberto:
  - Não é uma biblioteca cliente utilizada para se conectar a um servidor de banco de dados.
  - É uma biblioteca que é o próprio servidor.
  - Escreve e lê diretamente do arquivo do Banco de Dados.
  - Trabalha todas as informações necessárias para o processo de CRUD.

História

- Em janeiro de 2000, D. Richard Hipp trabalhava com sua equipe na Força Naval dos Estados Unidos, em um projeto de software, para misseis teleguiados.
- Neste momento era utilizado o banco de dados Informix, o que gerava alguns problemas de reinicialização do sistema.

História

- Para solucionar o problema com o Informix a equipe optou por migrar para o banco de dados PostgreSQL; mas o gerenciamento deste banco de dados ficou mais complexo que o esperado.
- Então, surgiu a ideia de se escrever um motor de banco de dados SQL que fosse simples para ler e escrever dados no disco rígido.
- 5 meses mais tarde foi iniciada a escrita da primeira versão da biblioteca de banco de dados SQLite.

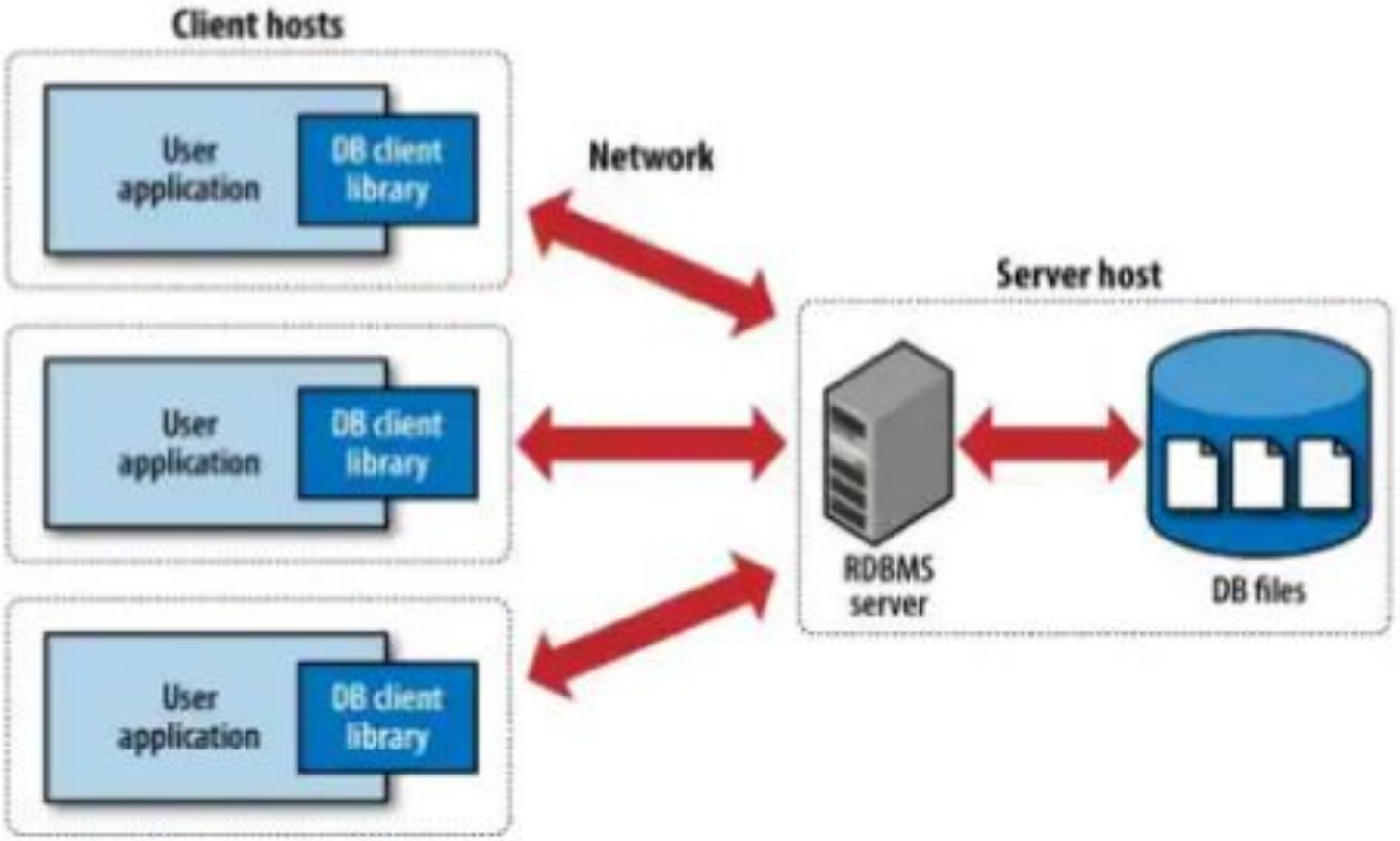
Principais usuários

- Adobe:
  - Utilizado no Photshop e no AdobeReader
- Apple Mac OS X:
  - Apple Mail e Safari WebBrowser
- Mozilla
  - Armazenamento de Metadados do Firefox Web Browser
- Google
  - Utilizado no Google Desktop e Google Gears
  - Utilizado na plataforma móvel Android

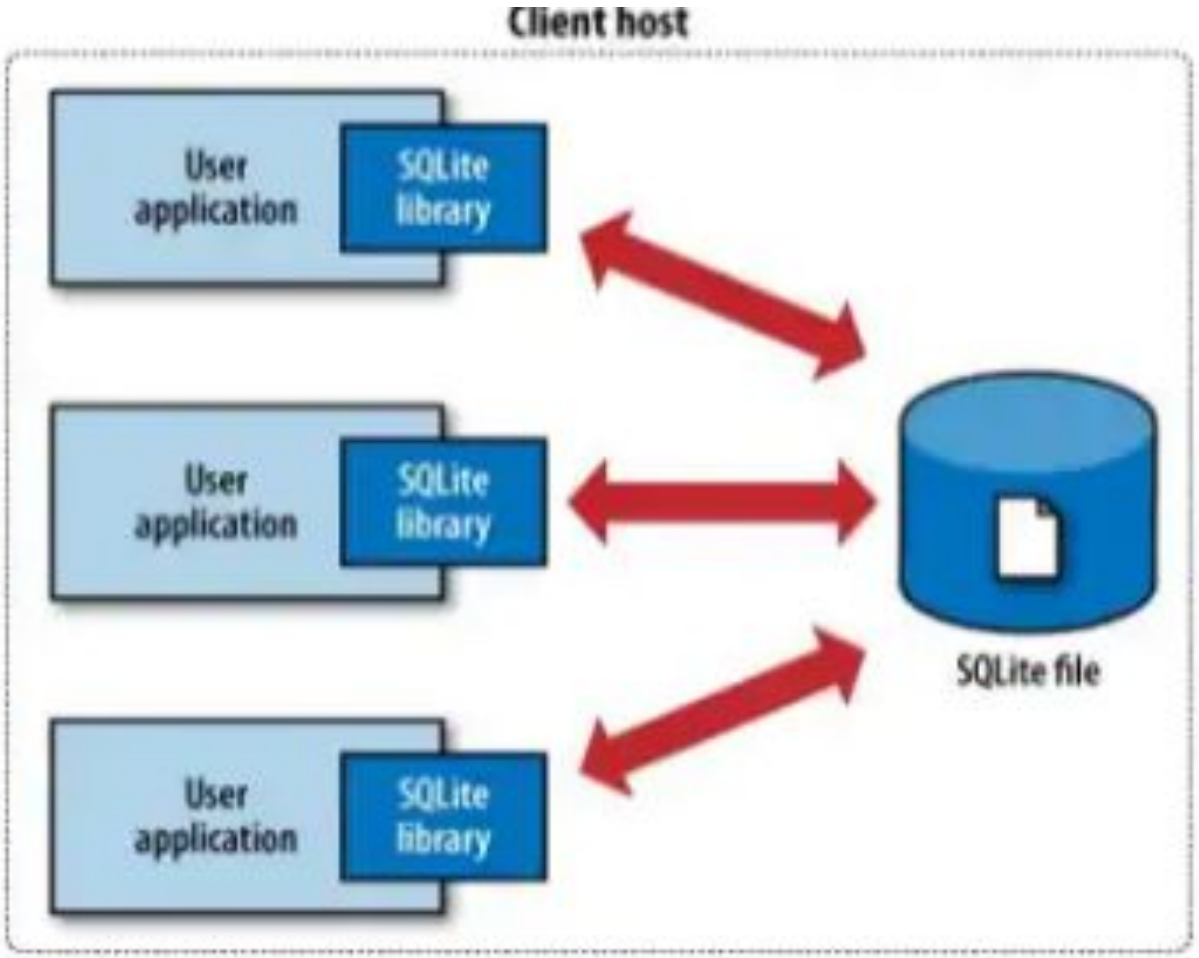
Características

- Simplicidade
- Portável: Totalmente escrita em ANSI C
- Confiável: Biblioteca OpenSource 100% testada
- Pequeno: Tamanho da biblioteca é inferior a 500KB
- Transações atômicas, consistentes, isoladas e duráveis (ACID).
- Zero-Configuração: Não necessita de configuração ou administração.
- Um banco de dados completo armazenado em um arquivo de disco multiplataforma.

Banco Relacional Padrão

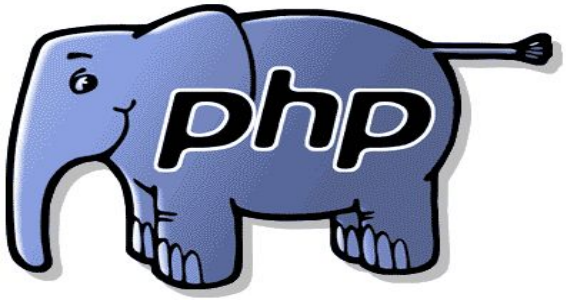


Banco Relacional SQLite



Características da biblioteca

é uma biblioteca,
programada em linguagem C que
implementa um banco de dados SQL
embutido.



Características da biblioteca

Não é um biblioteca cliente usada para
conectar com um servidor de BD,

mas sim o proprio servidor.

Introdução

- A persistência em aplicativos Android podem ser realizadas de 4 formas:
  - Bancos de dados SQLite
  - Memória interna
  - Cartão SD
  - Preferências
- Vamos trabalhar com as preferências e com o banco de dados SQLite, este utilizando a biblioteca Room

Onde usar

- Sites com menos de 100.000 requisições por dia.
- Dispositivos e sistemas embarcados.
- Aplicações Desktop.
- Aprendizado de Banco de Dados.
- Seu uso é recomendado onde é necessário simplicidade de administração, implementação e manutenção.

Desvantagens

- Não é recomendada a utilização em sistemas que possuem grande concorrência de leitura/escrita de dados.
- Não possui controle de acesso.
- Não suporta base de dados maiores de 2TB.

Desvantagens

- Não possui suporte cliente/servidor nativo.
- Não possui suporte interno de replicação e redundância.
- Não possui chave estrangeira.
- Limitação em uso de clausulas Where limitadas.

Tipos de Dados

- NULL - Como em qualquer outro banco de dados.
- INTEGER - Inteiro com sinal armazenado de até 8 bytes.
- REAL - Valor de ponto flutuante armazenado em 8 bytes
- TEXT - String armazenada em UTF-8, UTF-16BE ou UTF-16LE
- BLOB - Armazena em BLOB (objeto binário grande)

Tipos de Dados

- Boolean - Não existe, pode ser substituído por um INTEGER armazenado 0 (false) ou 1 (verdadeiro).
- Data e hora: Não existe, pode ser armazenado como TEXT, REAL ou Integer.

Tipos de Dados

- Data e Hora: a partir do seu armazenamento são acessados por funções:
- Date (timestring, modificador): Retorna a data no formato YYYY-MM-DD.
- Time (timestring, modificador): Retorna a hora no formato HH:MM:SS.
- DateTime (timestring, modificador): Retorna a data e a hora no formato YYYY-MM-DD HH:MM:SS.

Tipos de Dados

```
sqlite> select date('now');
2016-04-03
```

```
sqlite> select date('now','start of month','+1 month','-1 day');
2016-04-30
```

```
sqlite> select strftime('%s','now');
1459727985
sqlite> //segundos desde 01-01-1970
```

```
sqlite> select date(1092941466,'unixepoch');
2004-08-19
```

Recursos SQL Omitidos

- RIGHT e FULL OUTER JOIN.
- ALTER TABLE:
  - DROP COLUMN, ALTER COLUMN, ADD CONSTRAINT.
- TRIGGER FOR EACH STATMENT.
- VIEW:
  - DELETE INSERT e UPDATE.
- GRANT e REVOKE:
  - Somente permissões baseados no SO.
- FOREIGN KEY.

Outras informações importantes

TCL - Transaction Control Language

controle de processamento e exposição de mudanças;

Possui comandos DDL e DML

como todos SGBD's.

Outras informações importantes

BEGIN[DEFERRED | IMMEDIATE | EXCLUSIVE] [TRANSACTION]

palavras-chaves DEFERRED, IMMEDIATE ou EXCLUSIVE

DEFERRED= permite que outros clientes para continuar acessando e usando o banco de dados até que a transação não tem outra escolha a não ser bloqueia-los;

IMMEDIATE= adquirir um bloqueio imediatamente;
garante bloqueio para write;
Libera para operações
somente leitura;

EXCLUSIVE= bloquear todos os outros clientes, incluindo read-only dos clientes.

Procedimento para persistência de dados

- SQLite é um banco de dados predominantemente utilizado no contexto de desenvolvimento mobile.
- Além disso, pode ser utilizado em outras aplicações mesmo sem a finalidade de serem executadas em dispositivos móveis.

Preferências – SharedPreferences

- Por exemplo:
  - Salvar uma string

```
// contexto da
aplicação  val contexto
= this
// retorna a SharedPreferences STORAGE. Se não existir, cria
val prefs = contexto.getSharedPreferences(“STORAGE”,  0)
// habilita para edição
val editor =
prefs.edit()
// coloca uma string
editor.putString(“nome”,
“Fernando”)
// salva
editor.apply(
)
```

  - Ler uma String

```
// contexto da
aplicação  val contexto
= this
// retorna a SharedPreferences STORAGE. Se não existir, cria
```

Preferências – SharedPreferences

- Por exemplo:
  - Salvar um booleano

```
// contexto da
aplicação  val contexto
= this
// retorna a SharedPreferences STORAGE. Se não existir,
cria  val prefs = contexto.getSharedPreferences("STORAGE",
0)
// habilita para edição
val editor = prefs.edit()
// coloca um booleano
editor.putBoolean("flag", true)
// salva
editor.apply(
)
```

  - Ler um booleano

```
// contexto da
aplicação  val contexto
= this
// retorna a SharedPreferences STORAGE. Se não existir,
cria  val prefs = contexto.getSharedPreferences("STORAGE",
```

Preferências – SharedPreferences

- É comum criar uma classe para gerenciar as preferências
  - Organização
  - Encapsular as chamadas para a classe SharedPreferences

Sobre LMSApplication.kt

- No Android, uma classe Application é uma classe que armazena informações globais da aplicação
  - Evita a passagem de parâmetros
- Essa classe application herda de android.app.Application, que faz parte do ciclo de vida
- Quando é necessário criar uma classe Application customizada é preciso declara isso no AndroidManifest, no marcador <application>, atributo app:name

```
<manifest ...
    <application android:name=".LMSApplication" ...
</manifest>
```

Sobre LMSApplication.kt

- Com essa configuração o Android cria uma instância da classe declarada em app:name junto do processo do app
- A classe LMSApplication terá um método getInstance para retornar a instância atual do app, que terá informações como o contexto (context)
- O código completo dela está a seguir:

Sobre LMSApplication.kt

```
class LMSApplication: Application() {
    // chamado quando android iniciar o processo da aplicação
    override fun onCreate() {
        super.onCreate()
        appInstance = this
    }
    companion object {
        // singleton
        private var appInstance: LMSApplication?  = null
        fun getInstance(): LMSApplication {
            if (appInstance == null) {
                throw IllegalStateException("Configurar application no Android Manifest")
            }
            return appInstance!!
        }
    }

    // chamado quando android terminar processo da aplicação
    override fun onTerminate() {
        super.onTerminate()
    }
}
```

Sobre LMSApplication.kt

- Para acessar a classe e ler ou escrever informações globais basta utilizar da seguinte forma:

```
val app = LMSApplication.getInstance()
```

- O uso mais frequente da classe application é retornar o contexto global da aplicação, da seguinte forma:

```
val context = LMSApplication.getInstance().applicationContext
```

Singleton

Prefs

- O Singleton Prefs foi criado para encapsular as chamadas para a classe SharedPreferences
  - Salvar e ler dados de SharedPreferences
  - Singleton é como uma classe estática
- Ela deve ter um método para cada tipo de dado que deseja salvar e ler
  - String, boolean, int...
- Para nosso app, ela tem métodos para ler e salvar String e booleano
- Também tem um método prefs() que retorna o armazém das preferências, e uma constante que identifica seu nome

Classe Prefs

```
object Prefs {
    val PREF_ID = "LMS"

    // retorna o armazém de preferências PREF_ID
    private fun prefs(): SharedPreferences {
        val context = LMSApplication.getInstance().applicationContext
        return context.getSharedPreferences(PREF_ID, 0)
    }
    fun setBoolean(flag: String, valor: Boolean) = prefs().edit().putBoolean(flag, valor).apply()

    fun getBoolean(flag: String) = prefs().getBoolean(flag, false)


    fun setString(flag: String, valor: String) = prefs().edit().putString(flag, valor).apply()

    fun getString(flag: String) = prefs().getString(flag, "")

}
```

- Para salvar valores:

```
Prefs.setString("nome",
"Fernando")
```

- Para ler valores:

```
Prefs.getString("nome")
```

Exercíci

o

- login.xml
  - Adicionar checkbox no layout

```
<CheckBox
    android:id="@+id/checkLembrar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Lembrar login"
/>
```

Preferências – SharedPreferences

- Para finalizar, recupere as preferências no onCreate da MainAcitivity e popule os campos da tela de login

```
// procurar pelas preferências, se pediu para guardar usuário e senha
var lembrar = Prefs.getBoolean("lembrar")
if (lembrar) {
    var lembrarNome    = Prefs.getString("lembrarNome")
    var lembrarSenha       =
    Prefs.getString("lembrarSenha")
    campo_usuario.setText(lembrarNome)
    campo_senha.setText(lembrarSenha)
    checkBoxLogin.isChecked = lembrar

}
```

Banco de dados SQLite

- O Android tem suporte ao SQLite, um banco de dados leve e poderoso
- Cada aplicação pode criar quantos bancos de dados forem necessários
  - Ficam localizados na pasta /data/data/pacote.do.app/databases do aparelho
    - O BD é visível somente pela aplicação que o criou

Banco de dados SQLite

- Há duas maneiras de criar um BD SQLite no app
  - Utilizando a API do Android para SQLite
    - A API também possibilita executar qualquer comando SQL após a criação do banco
  - Utilizando um cliente SQLite
    - SQLite Expert Personal - http://www.sqliteexpert.com/
- Vamos utilizar a criação pela API do Android
- Utilizaremos a biblioteca Room para manipular e fazer a persistência no banco SQLite

SQLite e Room

- Primeiro, é preciso adicionar as dependências para o Room em app/build.gradle
  - No começo do arquivo, coloque a seguinte linha:

```
apply plugin: 'kotlin-kapt'
```

  - Nas dependências, adicione as seguintes linhas:

```
dependencies {
    //dependências existentes
    // Google Room - Banco de dados
    implementation "androidx.room:room-runtime:2.2.3"
    kapt "androidx.room:room-compiler:2.2.3"

}
```

SQLite e Room - Entity

- O Room usa anotações (@) para indicar que uma classe deve ser persistida
- Toda classe que deve ser persistida no banco local deve ser antada com @Entity
- O campo que representa a chave primária deve ser marcado com @PrimaryKey
- Faça essas anotações na classe Disciplina, para persistir os dados da disciplina em uma tabela chamada disciplina

```
@Entity(tableName = "disciplina")
class Disciplina : Serializable {

    @PrimaryKey
    var id:Long = 0
    ...
}
```

SQLite e Room - DAO

- Para cada classe anotada com @Entity é necessário criar uma interface DAO (Data Access Objects), que define as operações que serão realizadas no banco
  - Select (Query), insert, update e delete
- Crie [illegible] seguinte con[illegible]

```
@Dao
interface DisciplinaDAO {
    @Query("SELECT * FROM disciplina where id =
    :id")  fun getById(id: Long) : Disciplina?

    @Query("SELECT * FROM
    disciplina")  fun findAll():
    List<Disciplina>

    @Insert
    fun insert(disciplina: Disciplina)

    @Delete
    fun delete(disciplina: Disciplina)
}
```

SQLite e Room - DAO

- O DAO é uma interface, ou seja, seus métodos não precisam de implementação
- Coloca-se apenas os métodos que queremos e anotamos com @Query, @Insert ou @Delete
- O Room entende em tempo de execução o que o método faz baseado nos parâmetros, no tipo de retorno e na anotação
  - Segundo a documentação, se o método fizer sentido, o Room entende
- Se for uma consulta, é necessário colocar o SQL como parâmetro da anotação
  - Se houver algum filtro (where), o valor deve ser o mesmo nome do parâmetro do método
- Por fim, a interface deve ser anotada com @Dao

SQLite e Room – Gerenciamento do banco

- Também é necessário criar uma classe abstrata para gerenciar todo o banco de dados do Room
- Esta classe define na anotação a lista com todas as entidades que precisam ser persistidas e a versão do banco
- No corpo da classe, é preciso criar um método abstrato para cada DAO criado
  - Também não precisa de implementação, uma vez que o Room vai fornecer essa implementação
- Crie a classe DisciplinasDatabase com o

```
// anotação define a lista de entidades e a versão do banco
@Database(entities = arrayOf(Disciplina::class), version = 1)
abstract class LMSDatabase: RoomDatabase() {
    abstract fun disciplinaDAO(): DisciplinaDAO
}
```

SQLite e Room – Gerenciamento do banco

- Para fazer o gerenciamento global do banco, vamos criar uma classe DatabaseManager, que cria um singleton do LMSDatabase para ser utilizado em todo o aplicativo

```
object DatabaseManager {

    // singleton
    private var dbInstance:
    LMSDatabase  init {
        val appContext =
        LMSApplication.getInstance().applicationContext  dbInstance =
        Room.databaseBuilder(
                appContext, // contexto global
                LMSDatabase::class.java, // Referência da classe do
                banco
                "lms.sqlite" // nome do arquivo do banco
        ).build()
    }

    fun getDisciplinaDAO(): DisciplinaDAO
        {  return
        dbInstance.disciplinaDAO()
```

SQLite e Room – Gerenciamento do banco

- Feito isso, as operações no banco podem ser feitas de qualquer lugar no código, obtendo a instância do DAO a partir dos métodos do singleton DatabaseManager. Por exemplo:

```
val dao = DatabaseManager.getDisciplinaDAO()
```

SQLite e Room – Disciplinas

- Agora que a estrutura para trabalhar com o banco está pronta, vamos alterar o DisciplinaService para:
  - Salvar a lista de disciplinas offline quando retornar a lista do WS
  - Retornar a lista do banco local quando estiver offline
  - Remover a disciplina do banco local quando estiver offline

SQLite e Room – Disciplinas

- getDisciplinas

```
fun getDisciplinas (context: Context): List<Disciplina>
    {  var disciplinas = ArrayList<Disciplina>()
    if (AndroidUtils.isInternetDisponivel(context))
        {  val url = "$host/disciplinas"
        val json =
        HttpHelper.get(url)
        disciplinas =
        parserJson(json)
        // salvar offline
        for (d in disciplinas) {
            saveOffline(d)
        }
        return disciplinas
    } else {
        val dao =
        DatabaseManager.getDisciplinaDAO()   val
        disciplinas = dao.findAll()
        return disciplinas
    }

}
```

SQLite e Room – Disciplinas

- saveOffline

```
fun saveOffline(disciplina: Disciplina) : Boolean {
    val dao = DatabaseManager.getDisciplinaDAO()

    if (! existeDisciplina(disciplina)) {
        dao.insert(disciplina)
    }
    return true
}
```

- existeDisciplina

```
fun existeDisciplina(disciplina: Disciplina): Boolean
    {  val dao = DatabaseManager.getDisciplinaDAO()
    return dao.getById(disciplina.id) != null


}
```

SQLite e Room – Disciplinas

- delete

```
fun delete(disciplina: Disciplina): Response {
    if (AndroidUtils.isInternetDisponivel(LMSApplication.getInstance().applicationContext)) {
        val url = "$host/disciplinas/${disciplina.id}"
        val json = HttpHelper.delete(url)

        return parserJson(json)
    } else {
        val dao = DatabaseManager.getDisciplinaDAO()
        dao.delete(disciplina)
        return Response(status = "OK", msg = "Dados salvos localmente")
    }

}
```

Visualiza banco

- Abra a janela Device file Explorer (canto inferior direito)
- Navegue até: /data/data/{pacote_do_seu_app}/databases e identifique o arquivo do banco (lms.sqlite)
- Clique com o botão direito e salve no seu computador
- Você pode abrir o arquivo no SQLite Manager

Referências

- Aula baseada nos textos do livro:
  - LECHETA, R. R. Android Essencial com Kotlin. Edição: 1ª ed. Novatec, 2017.

https://fdocumentos.tips/download/sqlite-introducao

Github

- Link com os códigos gerados nesta aula:
- https://github.com/fabiodasilva500/Aulas-Mobile/tree/Aula-10-Persistencia