Artigo – http://mballem.wordpress.com/

JDBC com Banco de Dados Standalone

Autor

Marcio Ballem: é formado em Sistemas de Informação e possui certificação Oracle Certified Professional, Java SE 6 Programmer. Trabalhou profissionalmente com desenvolvimento em Delphi 7 e Java JEE.

Introdução

Imagine que você está desenvolvendo uma aplicação em Java e que esta aplicação é simples, sem a necessidade de acesso via rede por outros computadores. E você acaba escolhendo um banco de dados como MySql, Oracle, DB2, PostgreSql, entre outros. Então terá que instalar no computador do seu cliente um gerenciador de banco de dados, tornando mais complicada uma coisa que seria muito simples.

Nesta situação muitos acabam decidindo usar o MS Access, por não precisar da instalação de um gerenciador de banco de dados. Bom, já descomplicou um pouco, mas por que o MS Access e não um banco de dados como Derby, HSQLDB, Firebird, SqlLite? Todos esses bancos de dados possuem a resposta para o problema, como o MS Access eles também possuem a versão chamada *Standalone*, ou embarcado como alguns costumam chamar.

Neste artigo irei comentar e exemplificar como utilizar o Derby e o HSQLDB de forma *standalone*, sem que em momento algum precise utilizar um gerenciador de banco de dados nem para a criação do banco de dados, de suas tabelas e colunas.

1. JDBC

Java Database Connectivity API, ou apenas JDBC, é uma maneira desenvolvida pela Sun para resolver os problemas de conexão de aplicações escritas em Java com gerenciadores de banco de dados, os SGDB's.

O JDBC torna a vida do programador mais fácil por que ele apenas se preocupará com o desenvolvimento da aplicação. Para cada banco de dados existe um *Driver* JDBC especifico, assim uma aplicação escrita em Java para o banco de dados MySql, pode acabar sem nenhuma ou quase nenhuma alteração quando utilizada com outro banco de dados qualquer, bastaria apenas o *driver* JDBC referente e a configuração da chamada "string de conexão", onde se indica usuário, senha e local onde se encontra o gerenciador de banco de dados.

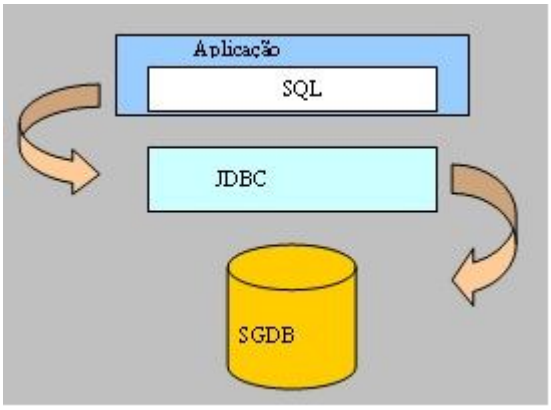

Figura 1

A figura 1 exemplifica como é realizada a comunicação com *driver* JDBC. No caso de aplicações *standalone*, não teremos o SGDB, então o próprio JDBC é quem simula o gerenciador de banco de dados.

2. SQL

SQL ou *Structured Query Language*, ou ainda Linguagem de Consulta Estruturada, é uma linguagem de pesquisa declarativa para banco de dados relacional. Muitas das características originais do SQL foram inspiradas na álgebra relacional.

Vale lembrar que o SQL é uma linguagem padrão, porém muitas vezes cada fornecedor de banco de dados como Oracle, DB2, entre outros, acabam criando novas funções que serão diferentes para cada banco de dados, então nesse caso um código SQL escrito para Oracle, poderá não funcionar quando utilizado no MySql.

3. Apache Derby

Apache Derby é um banco de dados relacional Java que pode ser embutido em programas Java. Consome aproximadamente 2 MB de espaço em disco. O Apache Derby é desenvolvido como um projeto open source sob a Apache 2.0 licence.

Primeiramente precisamos do *driver* JDBC do Derby, para isso devemos acessar a url http://db.apache.org/derby/derby_downloads.html. Vamos utilizar a versão 10.7.1.1. Faça o download do arquivo db-derby-10.7.1.1-bin.zip, ele possui documentação e o *driver* JDBC e também um manual de utilização.

Depois de efetuar o download, descompacte o arquivo. No diretório lib, iremos utilizar o o arquivo derby.jar, ele deverá ser adicionado as bibliotecas do projeto que será criado. No diretório docs/pdf, encontramos em vários idiomas, manuais de utilização do derby, o que pode ser muito útil caso queira utilizar por exemplo o gerenciador do Derby para criação e visualização do banco de dados.

4. Classe de Conexão

Crie um novo projeto com ou sem o uso de uma IDE, e nesse projeto crie um pacote chamado "br.mb.tutorialJDBC.dao". No pacote dao, iremos criar a classe de conexão chamada *ConnectionDataBase*, conforme a listagem 1.

Listagem 1. Classe de conexão com banco de dados

```
package br.mb.tutorialJDBC.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionDataBase {

    private static final String URL =
        "jdbc:derby:myDerby;create=true;user=derby;password=derby";
    private static final String DRIVER =
        "org.apache.derby.jdbc.EmbeddedDriver";

    public static Connection getConnection() {
        System.out.println("Conectando ao Banco de Dados");
        try {
            Class.forName(DRIVER);
            return DriverManager.getConnection(URL);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        return null;
    }
}
```

Preste muita atenção quando for fazer a importação as classes, todas elas são do pacote *java.sql*. Essa classe possui a variável estática URL, que é por onde passamos ao *driver* JDBC como iremos fazer a conexão com o banco, essa é a "string de conexão" que comentei anteriormente. Note que nessa string, não temos porta, nem o IP da máquina, isso acontece por que iremos acessar um banco de dados *standalone*.

A variável DRIVER, indica qual classe no *driver* JDBC iremos utilizar, isso significa o tipo de conexão que será criada, no caso *Embedded*, ou seja, do tipo *standalone*.

O método *getConnection()* irá retornar quando for necessário, uma conexão com o banco de dados. Mas o que acontece realmente aqui? Quando passamos para o método estático *Class.forName()* a classe do *Driver* Jdbc, ela irá registrar essa classe como um *driver* JDBC, pela classe *java.sql.DriverManager* e pelo método *registerDriver*, assim o *driver* já está registrado e então podemos abrir uma conexão com o banco de dados através da "string de conexão".

Agora o *DriverManager* vai perguntar para cada *Driver* registrado, se ele aceita a "string de conexão" passada. Se algum deles aceitar esta string, a conexão é aberta pelo *driver* que retorna uma conexão do tipo *java.sql.Connection*, caso contrário, uma exceção será lançada.

5. Classe para criar as tabelas

Agora que nossa conexão está criada, vamos criar uma classe para criar as tabelas e as colunas do nosso banco de dados.

Listagem 2. Classe para criação das tabelas

```
package br.mb.tutorialJDBC.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
```

```
public class CreateTables {
    private Connection connection;

    public CreateTables() {
        this.connection = ConnectionDataBase.getConnection();
    }

    public void createTablePessoas() {
        String sql = null;
        try {
            sql = "CREATE TABLE PESSOAS( " +
                    "id_pessoa integer not null GENERATED ALWAYS AS " +
                    "IDENTITY (START WITH 1, INCREMENT BY 1) " +
                    "CONSTRAINT PK_PESSOAS PRIMARY KEY, " +
                    "nome varchar(20) not null, " +
                    "idade integer not null" +
                    ")";

            PreparedStatement stmt = connection.prepareStatement(sql);
            stmt.execute();
            stmt.close();
            System.out.println("CreateTables.createTablePessoas Ok!");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void createTableTelefones() {
        String sql = null;
        try {
            sql = "CREATE TABLE TELEFONES( " +
                    "id_fone integer not null GENERATED ALWAYS AS " +
                    "IDENTITY (START WITH 1, INCREMENT BY 1) " +
                    "CONSTRAINT PK_TEFEFONES PRIMARY KEY, " +
                    "numero varchar(12), " +
                    "tipo varchar(11), " +
                    "pessoa integer not null, " +
                    "CONSTRAINT FK_PESSOAS FOREIGN KEY (pessoa) " +
                    "REFERENCES PESSOAS (id_pessoa) " +
                    ")";

            PreparedStatement stmt = connection.prepareStatement(sql);
            stmt.execute();
            stmt.close();
            System.out.println("CreateTables.createTableTelefones Ok!");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

O que acontece aqui? Primeiro criamos uma variável do tipo *java.sql.Connection*, onde "carregamos" ela com uma conexão aberta com o banco de dados através do construtor da classe.

Em seguida criamos os métodos para criação das tabelas. Dentro dos métodos criamos uma variável do tipo String, que recebe nosso Sql. Esse Sql é passado como parâmetro no método *prepareStatement()* que retorna um objeto do tipo *PreparedStatement*, que representa uma query. O método *execute()* irá executar essa

query no banco de dados e o método *close()* libera o objeto e os recursos Jdbc imediatamente, para assim, liberar o banco de dados para novos acessos.

6. Criando as classes Pessoa e Telefone

Agora devemos criar as classes *Pessoa* e *Telefone* com seus atributos referentes as colunas que serão criadas nas tabelas. A listagem 3 refere-se a classe *Pessoa* e a listagem 4 refere-se a classe *Telefone*. Crie no pacote tutorialJDBC, um novo pacote chamdo model.

Listagem 3. Classe Pessoa

```
package br.mb.tutorialJDBC.model;

public class Pessoa {
    private Integer id;
    private String nome;
    private int idade;

    //gerar os métodos getters and setters

    @Override
    public String toString() {
        return "Pessoa{" +
                "id=" + id +
                ", nome='" + nome + '\'' +
                ", idade=" + idade +
                '}';
    }
}
```

Listagem 4. Classe Telefone

```
package br.mb.tutorialJDBC.model;

public class Telefone {
    private Integer id;
    private String numero;
    private String tipo;
    private int idPessoa;

    //gerar os métodos getters and setters

    @Override
    public String toString() {
        return "Telefone{" +
                "id=" + id +
                ", numero='" + numero + '\'' +
                ", tipo='" + tipo + '\'' +
                ", idPessoa=" + idPessoa +
                '}';
    }

    public enum TipoFone {
        CEL(0,"Celular"), RES(1,"Residencial"), COM(2,"Comercial");

        private int indice;
        private String descricao;
```

```
        TipoFone(int indice, String descricao) {
            this.indice = indice;
            this.descricao = descricao;
        }

        public int getIndice() {
            return indice;
        }

        public String getDescricao() {
            return descricao;
        }
    }
}
```

Veja que em ambas classes, para facilitar a visualização dos dados, sobrescrevemos o método *toString()*. Na classe *Telefone* foi criado enumerados referentes ao tipo de telefone que será inserido no banco de dados. Se não conhece nada sobre enumerados em Java, pode dar uma lida sobre o assunto aqui http://pt.wikipedia.org/wiki/Enumera%C3%A7%C3%A3o_(tipo_de_dado).

7. Criando a classe GenericDao

DAO (*Data Access Object*), é um padrão para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados. Todas as funcionalidades de bancos de dados, tais como obter as conexões, mapear objetos Java para tipos de dados SQL ou executar comandos SQL, devem ser feitas por classes de DAO.

Já que todas os métodos referentes ao banco de dados devem ficar nas classes do tipo DAO, então para poupar código e aproveitar a orientação objetos, vamos criar uma classe que possa ser reutilizada e assim nos poupar trabalho para não criar em cada classe DAO um método *save*, *update* e *remove*.

Listagem 5. Classe GenericDao

```
package br.mb.tutorialJDBC.dao;

public abstract class GenericDao {
    private Connection connection;

    protected GenericDao() {
        this.connection = ConnectionDataBase.getConnection();
    }

    protected Connection getConnection() {
        return connection;
    }

    protected void save(String insertSql, Object... parametros) {
        try {
            PreparedStatement pstmt = getConnection().prepareStatement(insertSql);

            for (int i = 0; i < parametros.length; i++) {
                pstmt.setObject(i+1, parametros[i]);
            }

            pstmt.execute();
            pstmt.close();
```

```
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    protected void update(String updateSql, Object... parametros) {
        try {
            PreparedStatement pstmt = connection.prepareStatement(updateSql);

            for (int i = 0; i < parametros.length; i++) {
                pstmt.setObject(i+1, parametros[i]);
            }

            pstmt.execute();
            pstmt.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    protected void delete(String deleteSql, Object... parametros) {
        try {
            PreparedStatement pstmt = getConnection().prepareStatement(deleteSql);

            for (int i = 0; i < parametros.length; i++) {
                pstmt.setObject(i+1, parametros[i]);
            }

            pstmt.execute();
            pstmt.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    protected void shutdown() throws SQLException {
        getConnection().createStatement().executeUpdate("SHUTDOWN");
    }
}
```

Os métodos desta classe são bem parecidos com o da classe da listagem 2. A maior diferença é que aqui passamos para o objeto *pstmt* as colunas e os dados que serão inseridos nelas. O parametro String do método, passa o Sql criado e o parametro Object é uma lista de argumentos que contém todas as colunas que se referem ao Sql, por isso é feito um *for* para setar no objeto *pstmt* esses valores. Veremos mais sobre isso quando criarmos nossos *insert's, delete's* e *update's*.

8. Classe PessoaDao

A classe *PessoaDao* irá conter todos os comandos Sql que iremos utilizar quando referentes a tabela *Pessoas*. Esta classe receberá por herança os métodos criados na classe *GenericDao*, assim, não iremos precisar criar novamente aqueles três métodos e precisaremos apenas criar os métodos de consulta.

Quando utilizamos herança haverá alguns pontos importantes a serem observados, quem não sabe muito bem como funciona as chamadas por herança, pode dar uma lida no artigo, Declaração de Construtores em Java (http://mballem.wordpress.com/2011/01/19/declaracao-de-construtores-em-java/).

Listagem 6. Classe PessoaDao

```
package br.mb.tutorialJDBC.dao;

import br.mb.tutorialJDBC.model.Pessoa;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class PessoaDao extends GenericDao {

        public void save(Pessoa pessoa) {
                String sql = "INSERT INTO PESSOAS(nome, idade) VALUES (?,?)";
                save(sql, pessoa.getNome(), pessoa.getIdade());
        }

        public void update(Pessoa pessoa) {
                String sql = "UPDATE PESSOAS "
                                + "SET nome = ?, idade = ? "
                                + "where id_pessoa = ?";
                update(sql, pessoa.getNome(), pessoa.getIdade(), pessoa.getId());
        }

        public void delete(Pessoa pessoa) {
                String sql = "DELETE FROM PESSOAS WHERE id_pessoa = ? ";
                delete(sql, pessoa.getId());
        }

        public List<Pessoa> findPessoas() {
                List<Pessoa> pessoas = new ArrayList<Pessoa>();
                String sql = "SELECT * FROM PESSOAS";
                try {
                    PreparedStatement pstmt =
                        getConnection().prepareStatement(sql);

                    ResultSet rs = stmt.executeQuery();

                    while (rs.next()) {
                        Pessoa pessoa = new Pessoa();
                        pessoa.setId(rs.getInt("id_pessoa"));
                        pessoa.setNome(rs.getString("nome"));
                        pessoa.setIdade(rs.getInt("idade"));
                        pessoas.add(pessoa);
                    }
                    rs.close();
                    pstmt.close();
                    return pessoas;
                } catch (SQLException e) {
                        e.printStackTrace();
                }
                return null;
        }
}
```

Na listagem 6 foi criada a classe *PessoaDao* que estende *GenericDao*, assim, por herança herdamos os métodos *save()*, *update()* e *delete()*. Veja que foram criados três métodos com os mesmos nomes, mas

parâmetros diferentes. São esses três métodos que devem ser acessados quando for criado um objeto do tipo *Pessoa*, neles também criamos o Sql que será utilizado e a lista de argumentos com os campos que vamos persistir no banco de dados.

Cada símbolo de interrogação faz referência a um parâmetro, ou campo, que será enviado para o banco de dados. Na classe *GenericDao* o objeto *pstmt* irá unir esses parâmetros com o Sql, assim evita que façamos aqueles Sql cheios de campos e strings separadas por aspas duplas e aspas simples.

Lembre-se de manter a ordem do '?' com o campo que está passando por parâmetro, isso é muito importante. O primeiro '?' será referente ao primeiro argumento passado como parâmetro e assim por diante.

O método *findPessoas()* irá consultar no banco de dados todas as pessoas que estão cadastradas, adicionar cada uma delas em uma lista e retornar esta lista para ser exibida. A classe *ResultSet* recebe do método *executeQuery()* os dados da consulta e através dele podemos executar uma iteração no resultado da consulta para preencher o objeto *pessoa* e depois adicionar na lista. Por fim, fechamos o objeto *rs* e o objeto *pstmt* e retornamos uma lista de pessoas.

9. Classe Principal

Vamos agora testar a aplicação, para isso, vamos criar uma classe principal que será nosso ponto de partida para os testes. Crie uma pacote chamado testing dentro do pacote tutorialJDBC.

Listagem 7. Classe principal DerbyTest

```
package br.mb.tutorialJDBC.testing;

import br.mb.tutorialJDBC.dao.CreateTables;
import br.mb.tutorialJDBC.dao.PessoaDao;
import br.mb.tutorialJDBC.model.Pessoa;
import java.util.List;

public class DerbyTest {
    public static void main(String[] args) {
        criarTabelas();
        inserirPessoas();
        listarPessoas();
    }

    private static void criarTabelas() {
        new CreateTables().createTablePessoas();
        new CreateTables().createTableTelefones();
    }

    private static void inserirPessoas() {
        PessoaDao dao = new PessoaDao();

        Pessoa p1 = new Pessoa();
        p1.setNome("Ana Maria");
        p1.setIdade(65);
        dao.save(p1);

        Pessoa p2 = new Pessoa();
        p2.setNome("João Francisco");
        p2.setIdade(40);
        dao.save(p2);

        //crie quantos mais desejar
```

```
    }

    private static void listarPessoas() {
        List<Pessoa> pessoas = new PessoaDao().findPessoas();
        for (Pessoa pessoa : pessoas) {
            System.out.println("Derby.............:\n" + pessoa.toString());
        }
    }
}
```

Conforme a listagem 7, na criação da classe *DerbyTest* vamos executar primeiramente três métodos. O primeiro será para criar o banco de dados e as duas tabelas, o segundo irá inserir na tabela *Pessoas* e o terceiro irá listar os dados inseridos.

O banco de dados será criado no diretório de mesmo nível do diretório que possui seus arquivos fontes da aplicação. Depois de gerado o banco você encontrará lá uma pasta chamada ***myDerby*** e dentro dela todos os arquivos que o Derby criou referente ao seu banco de dados.

Se tudo funcionou perfeitamente, você terá no console da IDE a lista de pessoas cadastras e os arquivos do banco de dados no local onde citei. Vamos então criar mais três rotinas, um update, um delete e uma nova consulta.

9.1. Classe Principal novos métodos

Vamos agora testar a aplicação, para isso, vamos criar uma classe principal que será nosso ponto de partida para os testes.

Listagem 8. Métodos novos na classe DerbyTest e PessoaDao

```
public class PessoaDao extends GenericDao {
    //demais métodos foram omitidos

    public Pessoa findByName(String nome) {
        String sql = "SELECT * FROM PESSOAS WHERE nome = ?";
        Pessoa pessoa = null;
        try {
            PreparedStatement stmt = getConnection().prepareStatement(sql);
            stmt.setString(1, nome);
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                pessoa = new Pessoa();
                pessoa.setId(rs.getInt("id_pessoa"));
                pessoa.setNome(rs.getString("nome"));
                pessoa.setIdade(rs.getInt("idade"));
            }

            rs.close();
            stmt.close();
            return pessoa;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }
}

public class DerbyTest {
```

```
    public static void main(String[] args) {
        alterarPessoas();
        excluirPessoas();
    }

    private static void alterarPessoas() {
        Pessoa pessoa = new PessoaDao().findByName("Ana Maria");
        pessoa.setNome("Ana Amélia");
        pessoa.setIdade(28);

        new PessoaDao().update(pessoa);

        listarPessoas();
    }

    private static void excluirPessoas() {
        Pessoa pessoa = new PessoaDao().findByName("Ana Amélia");

        new PessoaDao().delete(pessoa);

        listarPessoas();
    }

    //métodos já criados foram omitidos
}
```

Veja na listagem 8 que no método *alterarPessoas()* foi feito primeiramente uma consulta por nome e o resultado retornou o objeto preenchido com os dados desta pessoa. Então foram adicionados novos valores neste objeto pessoas e em seguida executamos um *update()*, onde passamos por parâmetro este objeto *pessoa* alterado. Após a execução do *update()*, são listadas todas as pessoas cadastradas já com as devidas alterações.

No método *excluirPessoas()* foi feita uma pesquisa por nome, a qual retornou a pessoa em questão e então usamos esse objeto *pessoa* como parâmetro para executar o *delete()*. Por fim, a lista de pessoas é novamente impressa e desta vez sem a pessoa que foi excluída da base de dados.

10. Criando a classe TelefoneDao

A criação da classe *TelefoneDao* segue o mesmo principio da classe *PessoaDao*.

Listagem 9. Classe TelefoneDao

```
package br.mb.tutorialJDBC.dao;

import br.mb.tutorialJDBC.model.Telefone;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class TelefoneDao extends GenericDao {
    public void save(Telefone telefone) {
        String sql = "insert into TELEFONES(numero, tipo, pessoa) values (?,?,?)";
        save(sql, telefone.getNumero(),
                        telefone.getTipo(),telefone.getIdPessoa());
    }
```

```
public void update(Telefone telefone) {
            String sql = "UPDATE TELEFONES "
                               + "SET numero = ?, tipo = ?, pessoa = ? "
                               + "where id_fone = ?";
    update(sql, telefone.getNumero(), telefone.getTipo(),
                     telefone.getIdPessoa(), telefone.getId());
}

public void delete(Telefone telefone) {
    String sql = "DELETE FROM TELEFONES WHERE id_fone = ? ";
    delete(sql, telefone.getId());
}

public List<Telefone> findTelefones() {
    List<Telefone> telefones = new ArrayList<Telefone>();
    String sql = "SELECT * FROM TELEFONES";
    try {
        PreparedStatement stmt = getConnection().prepareStatement(sql);
        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            Telefone fone = new Telefone();
            fone.setId(rs.getInt("id_fone"));
            fone.setNumero(rs.getString("numero"));
            fone.setTipo(rs.getString("tipo"));
            fone.setIdPessoa(rs.getInt("pessoa"));
            telefones.add(fone);
        }
        rs.close();
        stmt.close();
        return telefones;
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return null;
}

public List<Telefone> findByNomePessoa(String nome) {
    List<Telefone> telefones = new ArrayList<Telefone>();
    String sql = "SELECT * FROM TELEFONES T, PESSOAS P " +
            "WHERE T.pessoa = P.id_pessoa AND " +
            "P.nome like ?";
    try {
        PreparedStatement stmt = getConnection().prepareStatement(sql);
        stmt.setString(1, nome);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Telefone fone = new Telefone();
            fone.setId(rs.getInt("id_fone"));
            fone.setNumero(rs.getString("numero"));
            fone.setTipo(rs.getString("tipo"));
            fone.setIdPessoa(rs.getInt("pessoa"));
            telefones.add(fone);
        }
        rs.close();
        stmt.close();
        return telefones;
    } catch (SQLException e) {
```

```
            e.printStackTrace();
        }
        return null;
    }
}
```

11. Manipulando a tabela Telefones

Vamos agora inserir, alterar, excluir e listar os dados referentes a tabela *Telefones*, também a partir da classe principal *DerbyTest*. Não esqueça de adicionar os novos *imports*.

Listagem 10. Manipulando telefones

```
import br.mb.tutorialJDBC.dao.TelefoneDao;
import br.mb.tutorialJDBC.model.Telefone;

public class DerbyTest {
    public static void main(String[] args) {
        inserirTelefones();
        alterarTelefones();
        excluirTelefones();
    }

    private static void inserirTelefones() {
        Pessoa pessoa = new PessoaDao().findByName("João Francisco");

        Telefone t1 = new Telefone();
        t1.setIdPessoa(pessoa.getId());
        t1.setNumero("55 9141 6598");
        t1.setTipo(Telefone.TipoFone.RES.getDescricao());
        new TelefoneDao().save(t1);

        Telefone t2 = new Telefone();
        t2.setIdPessoa(pessoa.getId());
        t2.setNumero("55 3333 0098");
        t2.setTipo(Telefone.TipoFone.RES.getDescricao());
        new TelefoneDao().save(t2);

        Telefone t3 = new Telefone();
        t3.setIdPessoa(pessoa.getId());
        t3.setNumero("55 2222 6008");
        t3.setTipo(Telefone.TipoFone.RES.getDescricao());
        new TelefoneDao().save(t3);

        listarTelefones();
    }

    private static void alterarTelefones() {
        List<Telefone> telefones =
                        new TelefoneDao().findByNomePessoa("João Francisco");

        for (Telefone telefone : telefones) {
            System.out.println("Derby.............:\n" + telefone.toString());
        }

        Telefone t1 = telefones.get(0);
        t1.setTipo(Telefone.TipoFone.CEL.getDescricao());
        new TelefoneDao().update(t1);
```

```
        Telefone t2 = telefones.get(2);
        t2.setTipo(Telefone.TipoFone.COM.getDescricao());
        new TelefoneDao().update(t2);

        listarTelefones();
    }

    private static void excluirTelefones() {
        List<Telefone> telefones =
                        new TelefoneDao().findByNomePessoa("João Francisco");

        for (Telefone telefone : telefones) {
            System.out.println("Derby.............:\n" + telefone.toString());
        }

        new TelefoneDao().delete(telefones.get(1));

        listarTelefones();
    }

    //métodos já criados foram omitidos
}
```

Conforme a listagem 10 o método *inserirTelefones()* funciona da seguinte maneira. Primeiro localizamos a pessoa que queremos inserir os telefones. Em seguida criamos objetos do tipo telefone onde inserimos os dados, inclusive o id pessoa retornado na pesquisa anterior. Então executamos um *save()* e depois listamos os telefones inseridos.

Veja que para o atributo *setTipo()*, utilizamos o enumerado criado na classe *Telefone* e inserimos ele como tipo residêncial.

No método *updateTelefones()* fazemos uma pesquisa em *Telefones* pelo nome da pessoa. Como uma pessoa pode ter vários telefones, precisamos retornar uma lista no método *findByNomePessoa()*.

Se você reparou bem, todos os telefones que incluímos foram do tipo residêncial, e agora então vamos alterar no banco de dados o 1° telefone para tipo celular e o 3° para o tipo comercial. Como temos uma lista com estes telefones, fica muito fácil, criamos um objeto *t1* onde atribuímos a ele o objeto da posição zero da nossa lista e no objeto *t2* atribuímos a partir da lista o objeto da posição dois, a posição um da lista manteremos como tipo residêncial.

Em seguida setamos o tipo de telefone para cada objeto criado e então executamos o método *upadate()* passando como parâmetro esses novos objetos. Por fim, a lista será novamente impressa com as alterações.

o método *excluirTelefones()* segue o mesmo principio do *alterarTelefones()*, criamos uma lista a partir de uma pessoa e então excluímos o telefone a partir desta lista e quando listamos novamente o telefone em questão não existirá mais na base de dados.

Sobre o Derby, finalizamos o tutorial, agora vamos mostrar algumas diferenças na criação e manipulação do HSQLDB.

12. HSQLDB

O HSQLDB (*Hyperthreaded Structured Query Language Database*) é um gerenciador de banco de dados (SGBD), de código aberto, escrito totalmente na linguagem Java.

Não é possível compará-lo, em termos de robustez e segurança com outros servidores SGBD, como Oracle ou Microsoft SQL Server, entretanto o HSQLDB é uma solução simples, que utiliza poucos recursos e que possui bom desempenho. Devido a essas características, ele é bastante utilizado em aplicações que são executadas em desktops e que necessitam interagir com uma camada de persistência através da linguagem SQL. A suíte office OpenOffice, na sua versão 2.0, inclui o HSQLDB como engine de armazenamento de dados, assim como o servidor de aplicações JBoss o utiliza para armazenar as filas de mensagens do tipo JMS.

Não podemos esquecer de adicionar o *drive*r do HSQLDB em nosso projeto, para isso vamos fazer o download em http://sourceforge.net/projects/hsqldb/files/hsqldb/. A versão usada neste tutorial é a 2.0.1.rc3. Após baixar e descompactar o arquivo, você deve copiar o arquivo hsqldb.jar que se encontra no diretório hsqldb-2.0.1-rce/hsqldb/lib. No diretório doc/guid é possivel encontrar um manual em pdf.

13. Configurações HSQLDB

Para configurar o HSQLDB, seguimos os mesmos passos do Derby. Altere na classe *ConnectionDataBase* os seguintes dados, conforme a listagem 11.

Listagem 11. Classe ConnectionDataBase com Hsqldb

```
public class ConnectionDataBase {

    private static final String URL =
        "jdbc:hsqldb:file:myHsqldb/db;user=sa;password=;";
    private static final String DRIVER =
        "org.hsqldb.jdbcDriver";
    //demais códigos não alterar nada
}
```

Um diferença no HSQLDB é que para criar o banco de dados devemos setar o usurário como 'SA' e a senha deve ficar fazia. Quando criarmos o banco, o HSQLDB irá criar no mesmo diretório que o Derby, uma pasta com o nome ***myHsqldb*** e o nome dos arquivos do banco serão **db**. Depois de criado o banco, você pode abrir o arquivo *db.script* e alterar o usuário e a senha, mas caso faça isso, não esqueça de alterar a "string de conexão" com o novo usuário e senha.

Neste arquivo *db.script* o Hsqldb guarda todas informações de criação de banco, de tabelas e os *insert's* executados.

Outra diferença é que o HSQLDB tem um sistema em que ele armazena os dados em memória e após um tempo ele passa para os arquivos do banco. Essa técnica foi criada para dar mais agilidade ao banco de dados, assim não precisa a cada alteração em banco ir lá e escrevê-la e sim fazer isso em lotes de tempos em tempos. Eu prefiro forçar a escrita do que esperar, e para fazer isso devemos usar o comando '"shutdown", veremos mais a frente.

14. Criando as tabelas no Hsqldb

Na classe *CreateTables* insira os Sql's para a criação de tabelas no HSQLDB, eles tem algumas diferenças em relação aos Sql's do Derby, veja na listagem 12.

Listagem 12. Classe CreateTables com Hsqldb

```
    sql = "CREATE TABLE PESSOAS( " +
                "id_pessoa integer " +
                "GENERATED BY DEFAULT AS IDENTITY (START WITH 1) not null, " +
```

```
                "nome varchar(20) not null, " +
                "idade integer not null, " +
                "CONSTRAINT PK_PESSOAS PRIMARY KEY(id_pessoa)" +
                ")";

    sql = "CREATE TABLE TELEFONES( " +
                "id_fone integer " +
                "GENERATED BY DEFAULT AS IDENTITY (START WITH 1) not null, " +
                "numero varchar(12), " +
                "tipo varchar(11), " +
                "pessoa integer not null, " +
                "CONSTRAINT PK_TEFEFONES PRIMARY KEY(id_fone), " +
                "CONSTRAINT FK_PESSOAS FOREIGN KEY (pessoa) " +
                "REFERENCES PESSOAS (id_pessoa) " +
                "ON UPDATE CASCADE ON DELETE CASCADE" +
                ")";
```

Agora veremos a questão do "shutdown" para a persistência dos dados em tempo de execução. Insira como na listagem 13, nos métodos *save()*, *update()* e *delete()* da classe *GenericDao*, logo entre as linhas *pstmt.close();* e *} catch (SQLException e) {...}* uma chamada ao método *shutdown()*.

Pronto, feito isso, vc pode executar os mesmos testes criados na classe *DerbyTest* sem precisar mudar mais nada. Pode criar uma classe nova para os testes ou usar a mesma *DerbyTest*.

Listagem 13. Classe GenericDao com Hsqldb

```
pstmt.close();

shutdown();

} catch (SQLException e) {
        e.printStackTrace();
}
```

Conclusão

Vimos como criar e configurar dois banco de dados do tipo standalone em aplicações Java, o Derby e o HSQLDB, e realizar o acesso a eles através dos drivers de conexão JDBC.

Seria interessante você ler os manuais dos dois banco de dados e realizar mais testes de inserção, alteração e exclusão. Lembre-se que o Sql será o mesmo que você usaria em um gerenciador de banco de dados.

Para quem não conhecia e nunca tinha utilizado, também teve acesso a criação de enumerados. Caso queira instar os gerenciadores do Derby ou do Hsqldb, consulte nos manuais a forma de como fazer.