N_PROG SIST II_A4 - Texto de apoio

Site: EAD Mackenzie
Tema: PROGRAMAÇÃO DE SISTEMAS II {TURMA 03B} 2023/1
Livro: N_PROG SIST II_A4 - Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: quinta, 4 mai 2023, 02:26

Descrição

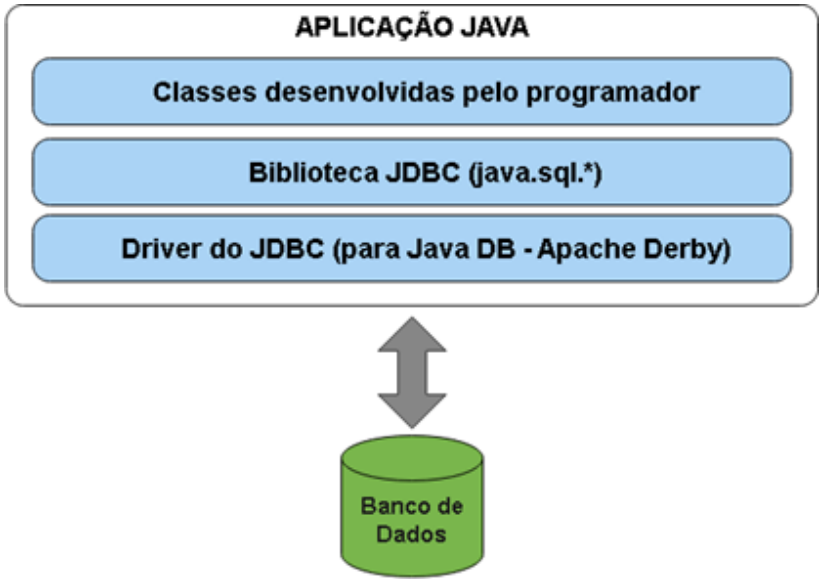
Índice

1. DATA ACCESS OBJECT (DAO) NO JAVA EE

Desenvolvimento de aplicação com acesso a um banco de dados em Java

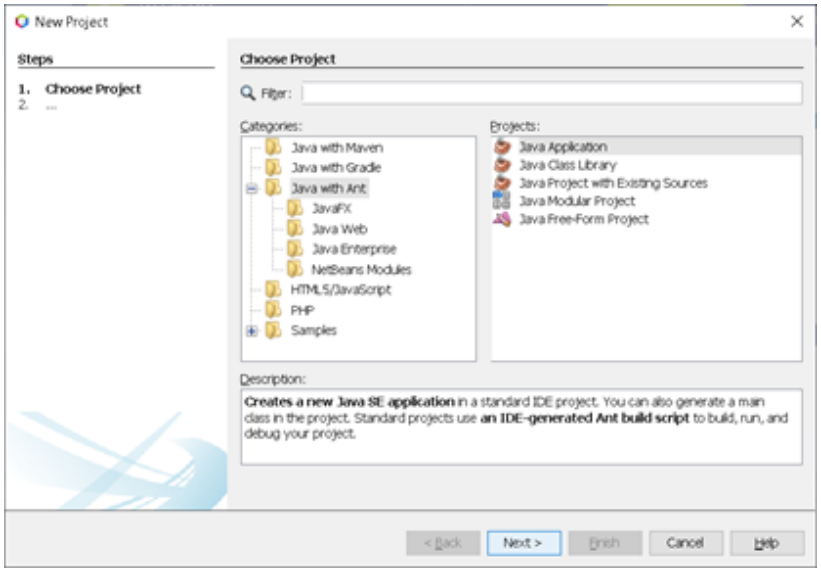
A conexão a um banco de dados na linguagem Java é realizada de maneira bastante simples. Para evitar que cada banco tenha sua própria API, temos uma única API para acesso a um banco de dados implementada no pacote **java.sql**. Nos referimos a essa API como *Java DataBase Connectivity* (**JDBC**).

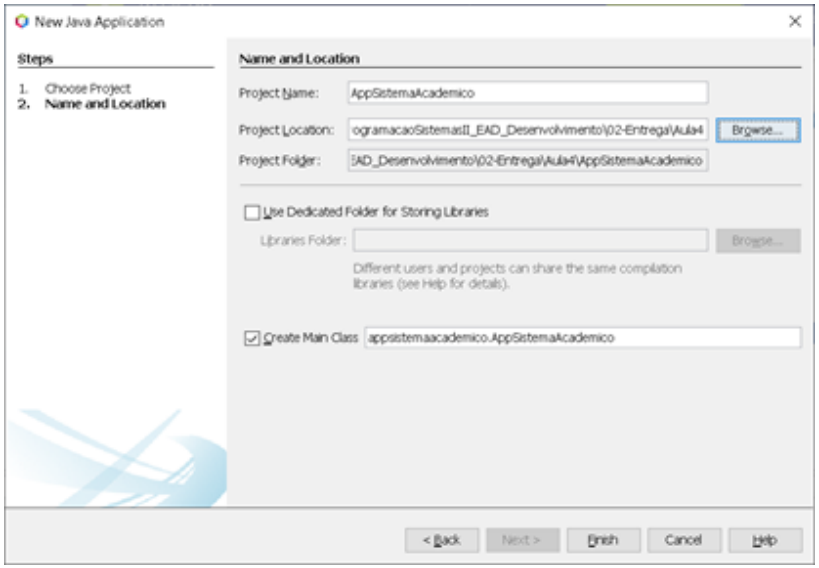
A API é quem fará a ponte entre o código cliente (aplicação Java) e o banco de dados. A comunicação com o banco é feita por meio de um protocolo proprietário do banco de dados, ou seja, cada um dos principais SGDB do mercado (Apache Derby, Oracle, MySQL, PostgreSQL...) possuem um protocolo implementado que permite acesso a seu banco de dados e a esse protocolo damos o nome de driver. O termo driver é análogo ao que usamos para impressoras: como é impossível que um sistema operacional saiba conversar com todo tipo de impressora existente, precisamos de um driver que faça o papel de "tradutor" dessa conversa. Veja a imagem abaixo ilustrando a comunicação com o banco de dados em uma aplicação na linguagem Java.



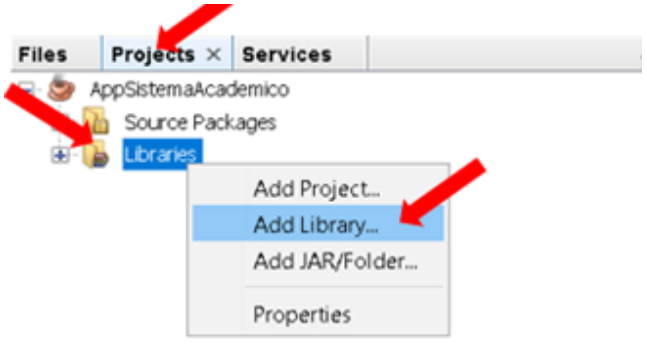
Fonte: Elaborada pelo autor.

Entendido isso, e após a instalação e configuração do servidor de aplicação **GlassFish** no NetBeans, podemos agora desenvolver uma aplicação no NetBeans que acessa a **tabela aluno** do **banco de dados sistema_academico** criado na **Aula 3**. Para isso, crie um projeto no NetBeans, conforme as janelas abaixo. Na primeira janela, você deve selecionar o tipo de aplicação; na janela seguinte, as informações referentes ao projeto: Nome do projeto (**Project Name**) e onde será salvo o projeto (**Project Location**). Veja que coloquei o nome do projeto como **AppSistemaAcademico**:

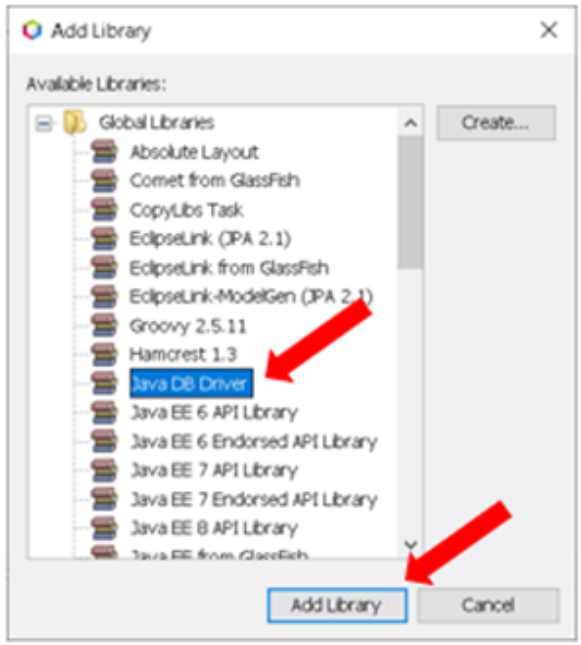




Com o projeto **AppSistemaAcademico** criado, devemos agora adicionar o driver do Java DB (Apache Derby) ao projeto. Para isso, acesse a aba **Projects** e clique com o botão direito do mouse em **Libraries**. Em seguida, abrirá um menu com a opções **Add Library**.

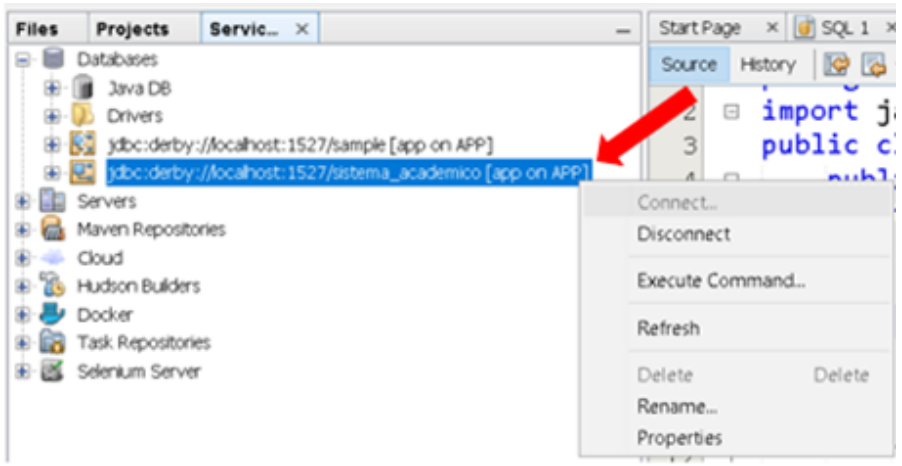


Em seguida, abrirá a janela abaixo para adicionar o driver do Java DB. Selecione **Java DB Driver** e, por fim, clique no botão **Add Library**.



Pronto, agora podemos fazer uma conexão ao banco de dados. Para abrir a conexão, usaremos a **classe DriverManager** e, a partir dela, o método estático **getConection()** com os seguintes argumentos: uma **String de conexão JDBC**, além do login e senha usuário definidos no momento da criação do banco de dados. Isso foi feito na **Aula 3**, lembra?

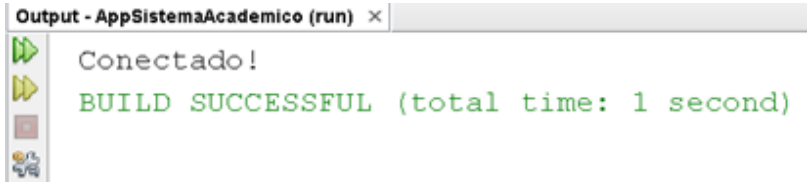
A **String de conexão** tem sempre o seguinte formato: **jdbc:sgdb://ip:porta/nome_do_banco**, e pode ser obtida a partir da aba de serviços (**Services**) no NetBeans. Veja a seguir:



Além disso, para que o exemplo funcione, é necessário que o banco de dados iniciado e conectado a IDE NetBeans. Abaixo, é apresentado a implementação do nosso primeiro exemplo:

```
2  import java.sql.*;
3  public class AppSistemaAcademico {
4      public static void main(String[] args) {
5          try {
6              Connection conexao = DriverManager.getConnection(
7              "jdbc:derby://localhost:1527/sistema_academico",
8              "app",
9              "app");
10             System.out.println("Conectado!");
11             conexao.close();
12         } catch (SQLException ex) {
13             System.out.println("Falha de conexao ao banco de dados!");
14         }
15     }
16 }
```

Na linha 2, é feita a importação do pacote **java.sql** que implementa a **API JDBC**. Na linha 7, é informada a **String de conexão** ao banco de dados **sistema_academico** (“**jdbc:derby://localhost:1527/sistema_academico**”). Nas linhas seguintes, temos o login (“**app**”) e senha (“**app**”). Repare que estamos tratando a exceção (a **SQLException)** com **try-catch**, pois essa exceção é lançada por muitos dos métodos da API JDBC. Caso ocorra algum erro, além disso, essa exceção é do tipo **checked exception**, ou seja, que deve obrigatoriamente ser tratada. Assim, caso tudo esteja configurado e codificado corretamente, teremos a saída abaixo no **Output** do NetBeans, **UFA!**



Estabelecida a conexão ao banco de dados, é possível criar uma consulta SQL e executá-la a partir da aplicação Java. Veja o exemplo abaixo que executa um comando SQL na **tabela aluno** do banco de dados **sistema_academico**.

```
2  import java.sql.*;
3  public class AppSistemaAcademico {
4      public static void main(String[] args) {
5          try {
6              Connection conexao = DriverManager.getConnection(
7              "jdbc:derby://localhost:1527/sistema_academico",
8              "app",
9              "app");
10             PreparedStatement stmt = conexao.prepareStatement("SELECT * from aluno");
11             ResultSet rs = stmt.executeQuery();
12             while( rs.next() ){
13                 System.out.println("id_aluno="+rs.getInt("id_aluno")+" "+
14                                    "nome="+rs.getString("nome")+" "+
15                                    "nota="+rs.getFloat("nota"));
16             }
17             conexao.close();
18         } catch (SQLException ex) {
19             System.out.println("Falha de conexao ao banco de dados!");
20         }
21     }
22 }
```

Observe que, para realizar uma consulta, é utilizado um objeto da **classe PreparedStatement**. Um objeto dessa classe pode ser obtido por meio **do método prepareStatement(String SQL)** da **classe Connection**. Perceba que o método tem como como argumento uma **String** com um comando SQL (**linha 10**).

Uma vez que um **objeto PreparedStatement** esteja disponível, é possível usar o **método executeQuery()** que retorna os registros selecionados na consulta SQL em um objeto da **classe ResultSet** (**linha 11**).

Inicialmente, o curso de registros é posicionado antes do primeiro registro. Para ir para o primeiro registro válido, é necessário chamar o **método next()** do **objeto ResultSet** no início da repetição. Esse método retornará **false** quando chegar no fim dos registros e, por isso, ele é normalmente utilizado para fazer um laço para navegar nos registros (**linha 12**).

Para retornar o valor de uma coluna, basta chamar um dos métodos **get** do **objeto ResultSet** referentes aos tipos de dados: **getInt()**, **getString()**, **getDouble()**, **getFloat()**, **getLong()**, entre outros. Com esses métodos, é possível recuperar os valores usando o nome da coluna (**linha 13**).

Considere que agora queremos fazer um programa que insira um novo aluno no banco de dados. Para inserir um novo registro em uma tabela de um banco de dados, use uma **String** com cláusula **INSERT** da linguagem SQL (**linha 10**).

```
2  import java.sql.*;
3  public class AppSistemaAcademico {
4      public static void main(String[] args) {
5          try {
6              Connection conexao = DriverManager.getConnection(
7              "jdbc:derby://localhost:1527/sistema_academico",
8              "app",
9              "app");
10             String SQL ="INSERT INTO aluno VALUES (?, ?, ?)";
11             PreparedStatement stmt = conexao.prepareStatement(SQL);
12             stmt.setInt(1, 123);
13             stmt.setString(2,"Fabio Lubacheski");
14             stmt.setDouble(3, 5.5);
15             System.out.println("linhas inseridas:"+stmt.executeUpdate());
16             conexao.close();
17         } catch (SQLException ex) {
18             System.out.println("Falha de conexao ao banco de dados!");
19         }
20     }
21 }
```

Perceba que não coloquei os pontos de interrogação de brincadeira, mas sim porque realmente não sei o que inseriremos. Executaremos o comando SQL, mas não sabemos ainda quais são os parâmetros que utilizaremos neste código SQL que será executado.

Como vimos no primeiro exemplo, uma consulta SQL é executada utilizando um objeto da **classe PreparedStatement** (**linha 11**).

Logo em seguida, podemos chamar um dos métodos **set** do **objeto PreparedStatement** para preencher os valores onde foram colocados os pontos de interrogação. Podemos passar a posição (começando em 1), trocando assim a interrogação pelo valor (**linhas 12, 13 e 14**) e, por fim, executar o comando INSERT usando o **método executeUpdate()** do objeto **PreparedStatement**. O **método executeUpdate()** pode ser usado quando queremos inserir um novo registro (**INSERT**), alterar registros (**UPDATE**), ou até mesmo remover registros (**DELETE**). Em todos os casos, o retorno desse método é a quantidade de linhas que foram inseridas, alteradas ou removidas (**linha 15**).

Implementando aplicações com Data Access Object no Java EE

Nos exemplos anteriores, você viu que o código fica um pouco confuso e complicado para fazer manutenção. Será que teria uma forma de remover o código de acesso ao banco de dados de suas classes de lógica e colocá-lo em uma classe responsável pelo acesso aos dados? Assim, o código de acesso ao banco de dados fica em um lugar só, tornando mais fácil a manutenção.

Que tal se pudéssemos chamar um método adiciona que insira um novo aluno ao banco dados?

```
// cria um novo aluno
Aluno aluno = new Aluno();
// preenche as informações dos alunos
aluno.setId(444);
aluno.setNome("Joao Ninguem");
aluno.setNota(8.5);
// adiciona o aluno no banco de dados
AlunoDao dao = new AlunoDao ();
dao.adiciona(aluno);
```

O código acima mostra que o acesso ao banco de dados é realizado somente pelo objeto da **classe AlunoDao**. A ideia de isolar todo o acesso ao banco por meio de um objeto é chamada de **Data Access Object**, ou simplesmente **DAO**, um dos mais famosos padrões de projeto (*Design Pattern*).

O padrão DAO é definido no livro *Core J2EE Patterns* como o padrão utilizado para abstrair e encapsular todos os acessos ao banco de dados. O DAO gerencia a conexão com o banco de dados para obter e armazenar informações. Com isso, a aplicação fica independente da implementação de um banco de dados específico. Isso desacopla a camada que implementa a regra de negócio da camada de acesso ao banco de dados. Assim, se mudarmos de um banco de dados Apache Derby para um banco de dados PostgreeSQL ou Oracle, o desacoplamento reduz significativamente qualquer impacto que poderia ocorrer.

Além disso, no padrão DAO, as informações que serão inseridas no banco de dados ficam armazenadas em objetos de **classes JavaBeans**. Uma classe do tipo JavaBeans possui o construtor sem argumentos (construtor padrão) e métodos de acesso do tipo **get** e **set**. Em nosso exemplo, a **classe Aluno** é uma classe JavaBeans que representa um registro na **tabela aluno**:

```
class Aluno {
    private int id;
    private String nome;
    private Double nota;
    public int getId() {
        return id;
    }
    public String getNome() {
        return nome;
    }
    public Double getNota() {
        return nota;
    }
    public void setId(int id) {
        this.id = id;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public void setNota(Double nota) {
        this.nota = nota;
    }
}
```

E, por fim, a **classe AlunoDao** é responsável por se conectar ao banco de dados e inserir um novo aluno na **tabela aluno**.

```
public class AlunoDao {
    // a conexão com o banco de dados
    private Connection connection;
    public AlunoDao() {
        this.connection = new ConnectionFactory().getConnection();
    }
    public void adiciona(Aluno aluno){
```

No construtor da **classe AlunoDao,** é realizada a conexão ao banco de dados, utilizando o **método getConnection()** da **classe ConnectionFactory**. O **método getConnection()** é uma **fábrica de conexões**, isto é, cria novas conexões com o banco de dados, não importando de onde elas vêm e eventuais detalhes de criação. Essa é a classe que deverá ser alterada caso precise trocar o banco de dados, pois a conexão com o banco de dados está encapsulada na **classe ConnectionFactory**.

```
public class ConnectionFactory {
    public Connection getConnection() {
        try {
            Connection conexao = DriverManager.getConnection(
            "jdbc:derby://localhost:1527/sistema_academico",
            "app",
            "app");
            return conexao;
        } catch (SQLException ex) {
            System.out.println("Falha de conexao ao banco de dados!");
        }
        return null;
    }
}
```

No código abaixo, temos o **método adiciona()** que insere um novo aluno passado por parâmetro. O novo aluno é um objeto da **classe JavaBeans Aluno**.

```
public void adiciona(Aluno aluno){
    String sql = "INSERT INTO aluno VALUES (?, ?, ?)";
    try {
        // prepared statement para inserção
        PreparedStatement stmt = this.connection.prepareStatement(sql);
        // seta os valores
        stmt.setLong(1,aluno.getId());
        stmt.setString(2,aluno.getNome());
        stmt.setDouble(3,aluno.getNota());
        // executa o comando
        stmt.executeUpdate();
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

A consulta (**SELECT**) também pode ser implementada na **classe AlunoDao**. Para isso, escreveremos o **método getLista()** que retorna uma lista de todos os alunos no banco de dados. Observe que dentro do método é utilizado um **objeto ArrayList** para armazenar a lista de alunos, sendo que cada elemento desse array é um objeto da **classe JavaBeans Aluno**. Veja o exemplo a seguir:

```java
public ArrayList <Aluno> getLista(){
    String sql = "SELECT * from aluno";
    ArrayList<Aluno> alunos = new ArrayList<Aluno>();

    try {
        PreparedStatement stmt = this.connection.prepareStatement(sql);
        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            // criando o objeto Aluno
            Aluno aluno = new Aluno();
            aluno.setId(rs.getInt( "id_aluno" ));
            aluno.setNome(rs.getString("nome"));
            aluno.setNota(rs.getDouble("nota"));

            // adicionando o objeto ao ArrayList
            alunos.add(aluno);
        }
        rs.close();
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return alunos;
}
```

Para testar o **método getList()**, podemos implementar o trecho de código conforme abaixo. Perceba que primeiro é criado um objeto da **classe AlunoDao**, depois é chamado o **método getList()** que retorna um **ArrayList** preenchido e, por fim, é impresso os **objetos do ArrayList** usando um **for-each** da linguagem Java, que itera sobre uma lista de objetos.

```java
AlunoDao dao = new AlunoDao ();
ArrayList <Aluno> listaAlunos = dao.getLista();

for (Aluno aluno : listaAlunos)
    System.out.println("ID: " + aluno.getId()+" "+
                       "Nome: " + aluno.getNome()+" "+
                       "Nota: " + aluno.getNota());
```