



Integração com banco de dados em Java

Você vai estudar a utilização de tecnologia Java, com base no middleware JDBC, para manipulação e consulta aos dados de bancos relacionais, com base em comandos diretos ou por meio de mapeamento objeto-relacional.

Prof. Tomás de Aquino

Preparação

Antes de iniciar o conteúdo, é necessário configurar o ambiente, com a instalação do JDK e Apache NetBeans, definindo a plataforma de desenvolvimento que será utilizada na codificação e execução dos exemplos práticos.

Objetivos

- Descrever os recursos para acesso a banco de dados no ambiente Java.
- Descrever o modelo de persistência baseado em mapeamento objeto-relacional.
- Aplicar tecnologia Java para a viabilização da persistência em banco de dados.

Introdução

Neste conteúdo, analisaremos as ferramentas para acesso a banco de dados com uso das tecnologias JDBC e JPA, pertencentes ao ambiente Java, incluindo exemplos de código para efetuar consulta e manipulação de registros.

Após compreender os princípios funcionais dessas tecnologias, construiremos um sistema cadastral simples e veremos como aproveitar os recursos de automatização do NetBeans para construção de componentes JPA.

Introdução



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Conceitos básicos

As aplicações de back-end são programas de software que lidam com tarefas de processamento de dados, lógica de negócios e gerenciamento de recursos. O middleware permite a comunicação entre as aplicações front-end e back-end para que a aplicação distribuída funcione sem problemas.

Veja neste vídeo os conceitos básicos relacionados com front-end e back-end, middleware e banco de dados Derby.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Front-end e back-end

Quando falamos de **front-end**, estamos nos referindo à camada de software responsável pelo interfaceamento do sistema, com o uso de uma linguagem de programação. Aqui, utilizaremos os aplicativos Java como opção de front-end.

Já o **back-end** compreende o conjunto de tecnologias com a finalidade de fornecer recursos específicos, devendo ser acessadas a partir de nosso front-end, embora não façam parte do mesmo ambiente, como os bancos de dados e as mensagerias. Para nossos exemplos, adotaremos o banco de dados Derby como back-end.

As mensagerias são um bom exemplo de back-end, com uma arquitetura voltada para a comunicação assíncrona entre sistemas, efetuada por meio da troca de mensagens. Essa é uma tecnologia crucial para diversos sistemas corporativos, como os da rede bancária.

Um grande problema, enfrentado pelas linguagens de programação mais antigas, é a demanda por versões específicas do programa para acesso a cada tipo de servidor de banco de dados, como Oracle, Informix, DB2 e SQL Server, entre diversos outros. Isso também ocorria com relação aos sistemas de mensagerias, em que podemos citar, como exemplos:

- MQ Series
- JBossMQ
- ActiveMQ
- Microsoft MQ

Middleware

Com diferentes componentes para acesso e modelos de programação heterogêneos, a probabilidade de ocorrência de erros é simplesmente enorme, levando à necessidade de uma camada de software intermediária, responsável por promover a comunicação entre o front-end e o back-end.

O termo middleware foi definido para a classificação desse tipo de tecnologia, que permite integração de forma transparente e mudança de fornecedor com pouca ou nenhuma alteração de código.

No ambiente Java, temos o **JDBC (Java Database Connectivity)** como middleware para acesso aos diferentes tipos de bancos de dados. Ele permite que utilizemos produtos de diversos fornecedores, sem modificações no código do aplicativo, sendo a consulta e manipulação de dados efetuadas por meio de comandos **SQL (Structured Query Language)** em meio ao código Java.



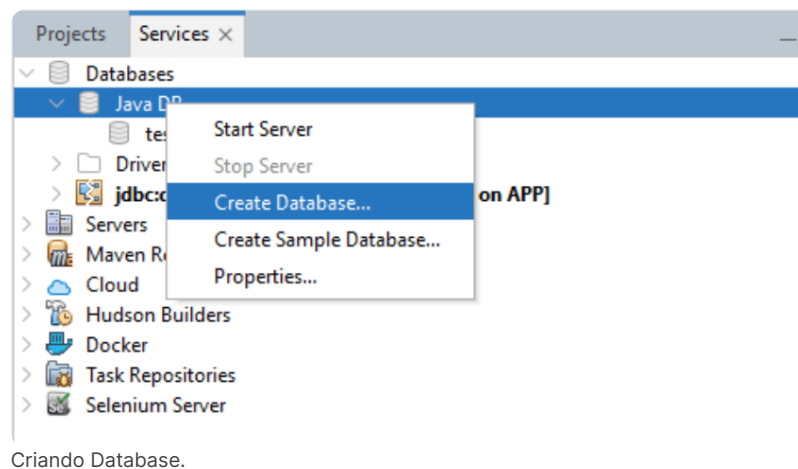
Atenção

Devemos evitar a utilização de comandos para um tipo de banco de dados específico, mantendo sempre a sintaxe padronizada pelo SQL ANSI, pois, caso contrário, a mudança do fornecedor de back-end poderá exigir mudanças no front-end.

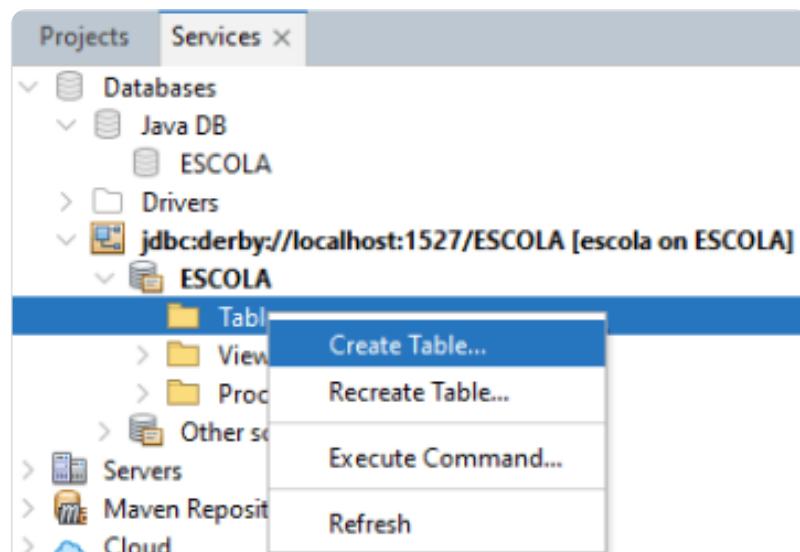
Banco de dados Derby

Entre as diversas opções de repositórios existentes, temos o Derby, ou Java DB, um banco de dados relacional construído totalmente com tecnologia Java, que não depende de um servidor e faz parte da distribuição padrão do JDK. Apache Derby é um subprojeto do Apache DB, disponível sob licença Apache, e que pode ser embutido em programas Java, bem como utilizado para transações on-line.

Podemos gerenciar nossos bancos de dados Derby de modo muito simples, por meio da aba Services do NetBeans, na divisão Databases. Para isso, devemos clicar com o botão direito sobre o driver Java DB, escolher a opção **Create Database** e preencher as informações necessárias para a configuração da nova instância do banco de dados, confira na imagem!



Vamos agora criar um banco de dados Java DB chamado "escola". Para isso, na janela que será aberta, devemos preencher o nome de nosso novo banco de dados com o valor "escola". Também iremos inserir usuário e senha, sendo interessante defini-los também como "escola", tornando mais fácil lembrar os valores utilizados. Ao clicar no botão de confirmação, o banco de dados será criado e ficará disponível para conexão, sendo identificado por sua **Connection String**, formada a partir do endereço de rede (**localhost**), da porta padrão (**1527**) e da instância que será utilizada (**escola**). Veja!



Criando um banco de dados.

A conexão é aberta com o duplo clique sobre o identificador, ou o clique com o botão direito e a escolha da opção Connect. Com o banco de dados aberto, podemos criar uma tabela, navegando até a divisão Tables, no esquema Escola, e utilizando o clique com o botão direito, para acessar a opção Create Table no menu de contexto.

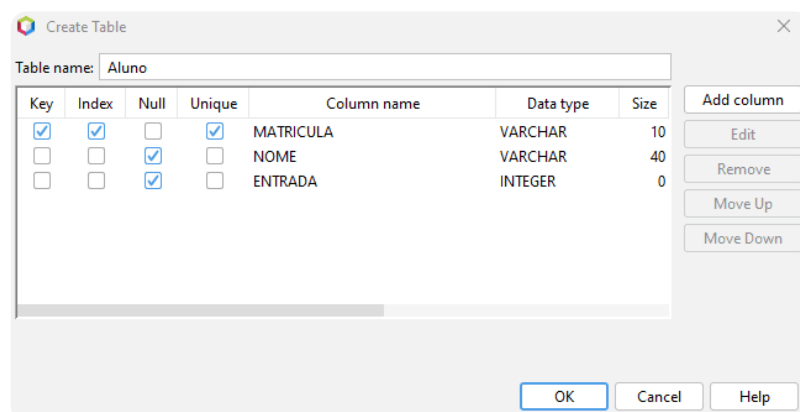
Na janela que se abrirá, configuraremos a tabela ALUNO, com os campos definidos como na próxima imagem.

Campo	Tipo	Complemento
MATRÍCULA	VARCHAR	Tamanho: 10 e Chave primária
NOME	VARCHAR	Tamanho: 40
ENTRADA	INTEGER	Sem atributos complementares

Tabela: Configuração da tabela ALUNO.

Tomás de Aquino

Definindo o nome da tabela e adicionando os campos, teremos a configuração que pode ser observada a seguir, com o processo sendo finalizado por meio do clique em OK. Cada campo deve ser adicionado individualmente, com o clique em **Add Column**. Confira!



teremos a abertura de uma área para execução de SQL e visualização de dados no editor de código, sendo possível acrescentar registros de forma visual com o uso de ALT+I, ou clique sobre o ícone referente.



Dica

Experimente acrescentar alguns registros, na janela de inserção que será aberta com as teclas ALT+I, utilizando Add Row para abrir novas linhas e preenchendo os valores dos campos para cada linha.

Ao final do preenchimento dos dados, devemos clicar em OK para finalizar, e o NetBeans executará os comandos INSERT necessários. Confira!

A imagem mostra uma janela de diálogo para inserção de dados no NetBeans. No topo, há uma barra de título com o texto "SELECT * FROM ESCOLA.ALUN...". Abaixo, há uma barra de ferramentas com ícones para adicionar, remover e atualizar registros, além de campos para "Max. rows: 100" e "Fetched Rows: 3". O corpo da janela contém uma tabela com quatro colunas: "#", "MATRICULA", "NOME" e "ENTRADA". A tabela possui três linhas de dados preenchidas e duas linhas vazias para inserção adicional.

#	MATRICULA	NOME	ENTRADA
1	E2020.0342	Ana Maria	2020
2	M2002.4002	João Alves	2020
3	E2019.0987	Luiz Sampaio	2019

Ferramenta visual para inserção de registros.

Atividade 1

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A Somente as afirmações I e II estão corretas.

B Somente as afirmações I e III estão corretas.

C Somente as afirmações II e III estão corretas.

D Somente a afirmação II está correta.

E Somente a afirmação I está correta.



A alternativa B está correta.

Consideradas as mesmas condições de aplicação, o middleware JDBC promove a comunicação entre o front-end e o back-end, funcionando com diferentes tipos de BD e com pouca ou nenhuma alteração de código.

Java Database Connectivity (JDBC)

Com o banco criado, podemos codificar o front-end em Java, utilizando os componentes do middleware JDBC, os quais são disponibilizados com a importação do pacote `java.sql`, tendo como elementos principais as classes `DriverManager`, `Connection`, `Statement`, `PreparedStatement` e `ResultSet`.

Confira neste vídeo quais são os componentes do JDBC e veja como utilizar as funcionalidades básicas.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Componentes do JDBC

Existem quatro tipos de drivers JDBC, apresentados a seguir.

JDBC-ODBC Bridge

Faz a conexão por meio do Open Database Connectivity (ODBC). O ODBC é uma especificação para uma API de banco de dados. Essa API é independente de qualquer SGBD ou sistema operacional e linguagem. A API ODBC baseia-se nas especificações da CLI do Open Group e do ISO/IEC. ODBC 3. x implementa totalmente essas duas especificações.

JDBC-Native API

Utiliza o cliente do banco de dados para a conexão.

JDBC-Net

Acessa servidores de middleware via Sockets, em uma arquitetura de três camadas.

Pure Java

Com a implementação completa em Java, não precisa de cliente instalado. Também chamado de Thin Driver.

Se considerarmos o caso específico do Oracle, são disponibilizados um driver Tipo 2, com acesso via OCI (Oracle Call Interface), e um driver Tipo 4, com o nome Oracle Thin Driver. A utilização da versão thin driver é mais indicada, pois não necessita da instalação do Oracle Cliente na máquina do usuário.

Utilização das funcionalidades básicas do JDBC

O processo para utilizar as funcionalidades básicas do JDBC segue quatro passos simples, acompanhe!

Instanciar a classe do driver de conexão.

Obter uma conexão (connection) a partir da Connection String, do usuário e da senha.

Instanciar um executor de SQL (statement).

Executar os comandos DML (linguagem de manipulação de dados).

Por exemplo, se quisermos efetuar a inclusão de um aluno na base de dados, podemos escrever o trecho de código apresentado a seguir, com o objetivo de demonstrar os passos descritos anteriormente, confira!

```
java
// passo 1
Class.forName("org.apache.derby.jdbc.ClientDriver");
// passo 2
Connection c1 = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/escola",
    "escola", "escola");
// passo 3
Statement st = c1.createStatement();
// passo 4
st.executeUpdate(
    "INSERT INTO ALUNO VALUES('E2018.5004', 'DENIS', 2018)");
st.close();
c1.close();
```

No início do código, temos a instância do driver Derby sendo gerada a partir de uma chamada para o método `forName`. Com o driver na memória, podemos abrir conexões com o banco de dados por meio do middleware **JDBC**.

Em seguida, é instanciada a **conexão c1**, por meio da chamada ao método **getConnection**, da classe **DriverManager**, sendo fornecidos os valores para Connection String, usuário e senha. Com relação à Connection String, ela pode variar muito, sendo iniciada pelo driver utilizado, seguido dos parâmetros específicos para aquele driver, os quais, no caso do Derby, são o endereço de rede, a porta e o nome da instância de banco de dados.

A partir da conexão c1, é gerado um executor de SQL de nome st, com a chamada para o método `createStatement`. Estando o executor instanciado, realizamos a inserção de um registro, invocando o método `executeUpdate`, com o comando SQL apropriado.

Na parte final, devemos fechar os componentes JDBC, na ordem inversa daquela em que foram criados, já que existe dependência sucessiva entre eles.



Atenção

As consultas ao banco são efetuadas utilizando `executeQuery`, enquanto comandos para manipulação de dados são executados por meio de `executeUpdate`.

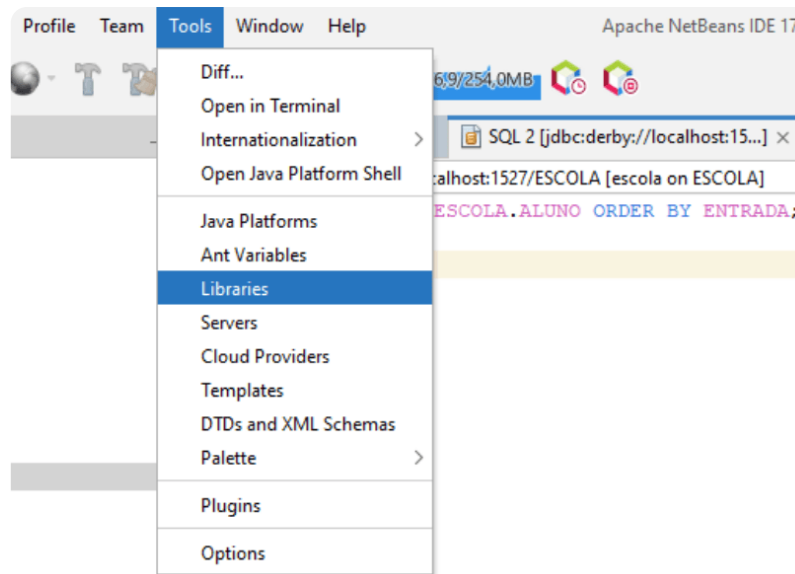
Para o comando de seleção, existe mais um detalhe: a recepção da consulta em um **ResultSet**, o que pode ser observado no trecho de código seguinte, no qual, com base na tabela criada anteriormente, efetuamos uma consulta aos dados inseridos:

```
java
Class.forName("org.apache.derby.jdbc.ClientDriver");
Connection c1 = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/loja",
    "loja", "loja");
Statement st = c1.createStatement();
ResultSet r1 = st.executeQuery("SELECT * FROM ALUNO");
while(r1.next())
    System.out.println("Aluno: " + r1.getString("NOME")+
        " - " + r1.getInt("ENTRADA"));
r1.close();
st.close();
c1.close();
```

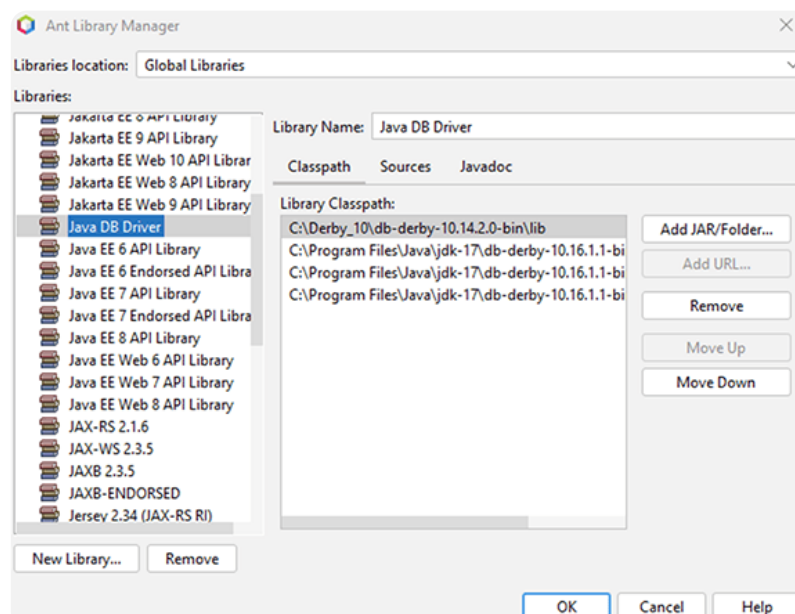
Após a recepção da consulta no objeto de nome **r1**, podemos nos movimentar pelos registros, com o uso de **next**, e acessar cada campo pelo nome, para a obtenção do valor, sempre utilizando o método correto para o tipo do campo, como **getString**, para texto, e **getInt**, para valores numéricos inteiros.

Ao efetuar a consulta, o **ResultSet** fica posicionado antes do primeiro registro, na posição **BOF** (beginning of file). Com o uso do comando **next** podemos mover para as posições seguintes, até atingir o final da consulta, na posição **EOF** (end of file). A cada registro visitado, efetuamos a impressão do nome do aluno e ano de entrada.

A biblioteca JDBC deve ser adicionada ao projeto, ou ocorrerá erro durante a execução. Para o nosso exemplo, devemos adicionar a biblioteca Java DB Driver, por meio do clique com o botão direito na divisão Libraries e uso da opção Add Library. Veja nas imagens!



Adicionando a biblioteca.



Adicionando a biblioteca.

Um recurso muito interessante, oferecido por meio do componente PreparedStatement, é a definição de comandos SQL parametrizados. Esses comandos são particularmente úteis no tratamento de datas, pois os bancos de dados apresentam interpretações diferentes para esse tipo de dado, e quem se torna responsável pela conversão para o formato correto é o próprio JDBC. Observe um exemplo da utilização do componente no trecho de código seguinte.

```
java
Class.forName("org.apache.derby.jdbc.ClientDriver");
Connection c1 = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/loja",
    "loja", "loja");
PreparedStatement ps = c1.prepareStatement(
    "DELETE FROM ALUNO WHERE ENTRADA = ?");
ps.setInt(1,2018);
ps.executeUpdate();
ps.close();
c1.close();
```

O uso de parâmetros facilita a escrita do comando SQL, sem a preocupação com o uso de apóstrofe ou outro delimitador, além de representar uma proteção contra os ataques do tipo SQL Injection.

Para definir os parâmetros, utilizamos pontos de **interrogação**, os quais assumem valores posicionais, a partir de **um**, o que é um pouco diferente da indexação dos vetores, que começa em zero.



Atenção

Cada parâmetro deve ser preenchido com a chamada ao método correto, de acordo com seu tipo, como `setInt`, para inteiro, e `setString`, para texto. Após o preenchimento, devemos executar o comando SQL, com a chamada para `executeUpdate`, no caso das instruções `INSERT`, `UPDATE` e `DELETE`, ou `executeQuery`, para a instrução `SELECT`.

No exemplo, o parâmetro foi preenchido com o valor 2018, e a execução do comando SQL resultante remove da base todos os alunos com entrada no referido ano.

Atividade 2

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A 2, 3, 1, 4.

B 2, 1, 3, 4.

C 1, 4, 2, 3.

D 1, 4, 2, 3.

E

2, 1, 4, 3.



A alternativa B está correta.

A API Java Database Connectivity (JDBC) fornece acesso universal a dados na linguagem de programação Java. Usando a API JDBC, você pode acessar praticamente qualquer fonte de dados, desde bancos de dados relacionais até planilhas e arquivos simples. Caso utilize o cliente do BD, a conexão é via JDBC-Native API. Utilize ODBC para fazer uma ponte entre o BD e o aplicativo. Com o Pure Java, você pode usar applets e aplicativos para conexão ao BD. Para acesso a servidores em uma arquitetura de três camadas, utilizamos o JDBC-Net.

JDBC na prática

Na aprendizagem de qualquer linguagem de programação, os exercícios práticos são fundamentais para a fixação do conteúdo e entendimento das particularidades, sintaxe e demais recursos.

Aprenda neste vídeo a criar o banco de dados Java DB, criar a tabela e incluir um aluno na base de dados.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Nesta prática, utilizaremos como ponto de partida os códigos vistos nas seções “Banco de Dados Derby” e “Utilização das funcionalidades básicas do JDBC”. Você deverá implementar os passos a seguir:

- Criar o banco de dados Java DB chamado “escola”.
- Criar a tabela Aluno no banco de dados Derby.
- Incluir um aluno na base de dados.
- Realizar uma consulta aos dados inseridos.
- Adicionar a biblioteca Java DB Driver.
- Utilizar componente PreparedStatement.

Atividade 3

Questão 1

HOVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A

II e III, apenas.

B II e IV, apenas.

C I, III e IV, apenas.

D I, II e IV, apenas.

E I e IV, apenas.



A alternativa D está correta.

A instalação do JDK precede a instalação do NetBeans e do Derby. O servidor do BD deve ser iniciado antes de qualquer uso do Derby. A criação de tabelas só é possível após a conexão do banco de dados. Na área para a introdução dos dados, é possível fazer a introdução manual e, ao clicar em OK, o NetBeans executa os comandos INSERT que vão popular a tabela do banco de dados.

Orientação a objetos e o modelo relacional

Bases de dados cadastrais seguem um modelo estrutural baseado na álgebra relacional e no cálculo relacional, áreas da matemática voltadas para a manipulação de conjuntos, apresentando ótimos resultados na implementação de consultas. Mesmo com grande eficiência, a representação matricial dos dados, com registros separados em campos e apresentados sequencialmente, não é a mais adequada para um ambiente de programação orientado a objetos.

Assista a este vídeo sobre o mapeamento objeto-relacional e data access object.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Mapeamento objeto-relacional

Torna-se necessário efetuar uma conversão entre os dois modelos de representação, o que é feito com a criação de uma classe de entidade para expressar cada tabela do banco de dados, à exceção das tabelas de relacionamento. Segundo a conversão efetuada, os atributos da classe serão associados aos campos da tabela, e cada registro poderá ser representado por uma instância da classe.

Para a nossa tabela de exemplo, podemos utilizar a classe apresentada a seguir, na qual, para simplificar o código, não iremos utilizar métodos getters e setters. Confira o código a seguir!

```
java
```

```
public class Aluno {  
    public String matricula;  
    public String nome;  
    public int ano;  
  
    public Aluno() { }  
  
    public Aluno(String matricula, String nome, int ano) {  
        this.matricula = matricula;  
        this.nome = nome;  
        this.ano = ano;  
    }  
}
```

Definida a entidade, podemos mudar a forma de lidar com os dados, efetuando a leitura dos dados da tabela e alimentando uma coleção que representará os dados para o restante do programa.

java

```
List lista = new ArrayList<>();
Class.forName("org.apache.derby.jdbc.ClientDriver");
Connection c1 = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/loja",
    "loja", "loja");
Statement st = c1.createStatement();
ResultSet r1 = st.executeQuery("SELECT * FROM ALUNO");
while(r1.next())
    lista.add(new Aluno(r1.getString("MATRICULA"),
        r1.getString("NOME"),
        r1.getInt("ENTRADA")));
r1.close();
st.close();
c1.close();
```

Como podemos observar, o código apresenta grande similaridade com o trecho para consulta apresentado anteriormente. Porém, aqui instanciamos uma coleção, e para cada registro obtido, adicionamos um objeto inicializado com seus dados. Após concluído o preenchimento da coleção, com o nome lista, não precisamos acessar novamente a tabela e podemos lidar com os dados segundo uma perspectiva orientada a objetos.

A conversão de tabelas e respectivos registros em coleções de entidades é uma técnica conhecida como mapeamento objeto-relacional.

Se quisermos listar o conteúdo da tabela, em outro ponto do código, após alimentarmos a coleção de entidades, podemos utilizar um código baseado na funcionalidade padrão das coleções do Java.

java

```
for (Aluno aluno : lista) {
    System.out.println("Aluno: " + aluno.nome +
        "( " + aluno.matricula + ") - " +
        aluno.ano);
}
```

Data access object

Agora que sabemos lidar com as operações sobre o banco de dados e definimos uma entidade para representar um registro da tabela, seria interessante organizar a forma de programar. É fácil imaginar a dificuldade para efetuar manutenção em sistemas com dezenas de milhares de linhas de código Java, contendo diversos comandos SQL espalhados ao longo das linhas.

Baseado nessa dificuldade, foi produzido um padrão de desenvolvimento com o nome DAO (data access object), cujo objetivo é concentrar as instruções SQL em um único tipo de classe, permitindo o agrupamento e a reutilização dos diversos comandos relacionados ao banco de dados.

Atividade 1

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A getConnection e executeQuery.

B createStatement e executeQuery.

C executeQuery e close.

D executeQuery e add.

E getConnection e add.



A alternativa D está correta.

O comando `st.executeQuery("SELECT * FROM ALUNO");` busca toda a tabela e traz para a memória principal do computador. No laço `"while(r1.next())"` temos o comando `"lista.add(new Aluno(r1.getString("MATRICULA"))"`, sendo executado até que toda a tabela esteja contida na lista denominada "lista".

O padrão DAO

Normalmente, temos uma classe DAO para cada classe de entidade relevante para o sistema. Como a codificação das operações sobre o banco apresenta muitos trechos comuns para as diversas entidades do sistema, podemos concentrar as similaridades em uma classe de base, e derivar as demais, segundo o princípio de herança.

A utilização de elementos genéricos também será um facilitador na padronização das operações comuns sobre a tabela, como inclusão, exclusão, alteração e consultas.

Confira neste vídeo a criação da classe de base DAO e da classe `AlunoDAO`, bem como os métodos de manipulação de dados da classe `AlunoDAO`.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Classe de base DAO

Vamos observar, no trecho de código a seguir, a definição de uma classe de base, a partir da qual serão derivadas todas as classes DAO do sistema.

java

```
public abstract class GenericDAO {
    protected Connection getConnection() throws Exception{
        Class.forName("org.apache.derby.jdbc.ClientDriver");
        return DriverManager.getConnection(
            "jdbc:derby://localhost:1527/escola",
            "escola", "escola");
    }
    protected Statement getStatement() throws Exception{
        return getConnection().createStatement();
    }
    protected void closeStatement(Statement st) throws Exception{
        st.getConnection().close();
    }
    public abstract void incluir(E entidade);
    public abstract void excluir(K chave);
    public abstract void alterar(E entidade);
    public abstract E obter(K chave);
    public abstract List obterTodos();
}
```

Iniciamos criando os métodos **getStatement** e **closeStatement**, com o objetivo de gerar executores de SQL e eliminá-los, efetuando também as conexões e desconexões nos momentos necessários. Outro método utilitário é o **getConnection**, utilizado apenas para encapsular o processo de conexão com o banco.

Nossa classe **GenericDAO** é abstrata, definindo de forma genérica as assinaturas para os métodos que acessam o banco, onde **E** representa a classe da **entidade** e **K** representa a classe da **chave primária**. Os descendentes de **GenericDAO** deverão implementar os métodos abstratos, preocupando-se apenas com os aspectos gerais do mapeamento objeto-relacional e fazendo ampla utilização dos métodos utilitários.



Comentário

Uma grande vantagem da estratégia adotada é viabilizarmos a mudança de fornecedor de banco de dados de forma simples, já que o processo de conexão pode ser encontrado em apenas um método, reutilizado por todo o restante do código.

Se você quiser utilizar um banco de dados Oracle, com acesso local e instância padrão XE, mantendo o usuário e a senha definidos, modifique o corpo do método **getConnection**, conforme sugerido no trecho de código seguinte.

java

```
Class.forName("oracle.jdbc.OracleDriver");
return DriverManager.getConnection(
    jdbc:oracle:thin:@localhost:1521:XE",
    "escola", "escola");
```

Classe AlunoDAO

Com a classe de base definida, podemos implementar a classe **AlunoDAO**, concentrando as operações efetuadas sobre nossa tabela, a partir da entidade **Aluno** e da chave primária do tipo **String**. Veja o início de sua codificação.

```

java

public class AlunoDAO extends GenericDAO{
    @Override
    public List obterTodos() {
        List lista = new ArrayList<>();
        try {
            ResultSet r1 = getStatement().executeQuery(
                "SELECT *
FROM ALUNO");
            while(r1.next())
                lista.add(new
Aluno(r1.getString("MATRICULA"),
r1.getString("NOME"),r1.getInt("ENTRADA")));
            closeStatement(r1.getStatement());
        } catch (Exception e){
        }
        return lista;
    }
    @Override
    public Aluno obter(String chave) {
        Aluno aluno = null;
        try {
            PreparedStatement ps =
                getConnection().prepareStatement(
                    "SELECT * FROM ALUNO WHERE MATRICULA
= ?");
            ps.setString(1, chave);
            ResultSet r1 = ps.executeQuery();
            if (r1.next())
                aluno = new
Aluno(r1.getString("MATRICULA"),
r1.getString("NOME"),
r1.getInt("ENTRADA"));
            closeStatement(ps);
        } catch (Exception e) {
        }
        return aluno;
    }
}

```

O código ainda não está completo, e certamente apresentará erro, pois não implementamos todos os métodos abstratos definidos, mas já temos o método `obterTodos` codificado nesse ponto. Serão retornados todos os registros de nossa tabela, no formato de um `ArrayList` de entidades do tipo `Aluno`. Inicialmente será executado o SQL necessário para a consulta e, em seguida, adicionada uma entidade à lista para cada registro obtido no cursor.

Também podemos observar o método `obter`, para consulta individual, retornando uma entidade do tipo `Aluno` para uma chave fornecida do tipo `String`. A implementação do método envolve a execução de uma consulta parametrizada, em que o campo matrícula deve coincidir com o valor da chave, sendo retornado o registro equivalente por meio de uma instância de `Aluno`.



Atenção

Como a consulta foi efetuada a partir da chave, sempre retornará um registro ou nenhum, sendo necessário apenas o comando if para mover do BOF para o primeiro e único registro. Caso a chave não seja encontrada, a rotina não entrará nessa estrutura condicional e retornará um produto nulo.

Métodos de manipulação de dados da classe AlunoDAO

Agora que a consulta aos registros foi implementada, devemos acrescentar os métodos de manipulação de dados na classe AlunoDAO.

```
java

@Override
public void incluir(Aluno entidade) {
    try {
        PreparedStatement ps = getConnection().prepareStatement(
            "INSERT INTO ALUNO
VALUES (?, ?, ?)");

        ps.setString(1, entidade.matricula);
        ps.setString(2, entidade.nome);
        ps.setInt(3, entidade.ano);
        ps.executeUpdate();
        closeStatement(ps);
    } catch (Exception e) { }
}

@Override
public void excluir(String chave) {
    try {
        PreparedStatement ps = getConnection().prepareStatement(
            "DELETE FROM ALUNO WHERE MATRICULA
= ?");

        ps.setString(1, chave);
        ps.executeUpdate();
        closeStatement(ps);
    } catch (Exception e) { }
}

@Override
public void alterar(Aluno entidade) {
    try {
        PreparedStatement ps = getConnection().prepareStatement(
            "UPDATE ALUNO SET NOME = ?, ENTRADA = ? "+
            " WHERE MATRICULA
= ?");

        ps.setString(1, entidade.nome);
        ps.setInt(2, entidade.ano);
        ps.setString(3, entidade.matricula);
        ps.executeUpdate();
        closeStatement(ps);
    } catch (Exception e) { }
}
```

Todos os métodos para manipulação de dados utilizam **PreparedStatement**, obtido a partir de **getConnection**, com o fornecimento da instrução SQL **parametrizada**. As linhas seguintes sempre envolvem o preenchimento

de parâmetros e chamada para o método **executeUpdate**, em que o comando SQL resultante, após a substituição das interrogações pelos valores, é efetivamente executado no banco de dados.

O mais simples dos métodos implementados se refere a excluir, por necessitar apenas da chave primária e uso da instrução DELETE condicionada à chave fornecida. Os demais métodos seguem a mesma forma de implementação, com a obtenção dos valores para preenchimento dos parâmetros a partir dos atributos da entidade fornecida. O método **incluir** está relacionado ao comando **INSERT**, e o método **alterar** representa as instruções SQL do tipo **UPDATE**.

Uma regra para efetuar mapeamento objeto-relacional, seguida por qualquer framework com esse objetivo, é a de que a chave primária da tabela não pode ser alterada. Isso permite manter o referencial dos registros ao longo do tempo.

Após construir a classe DAO, podemos utilizá-la ao longo de todo o sistema, consultando e manipulando os dados sem a necessidade de utilização direta de comandos SQL. Veja o trecho de exemplo a seguir, que permitiria imprimir o nome de todos os alunos da base de dados.

```
java

AlunoDAO dao = new AlunoDAO();
dao.obterTodos().forEach(aluno -> {
    System.out.println(aluno.nome);
});
```

Atividade 2

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A INSERT.

B CREATE.

C DELETE.

D SELECT.

E UPDATE



A alternativa D está correta.

Na construção de uma classe DAO, precisamos minimamente dos métodos obterTodos, incluir, excluir e alterar, que estarão relacionados, respectivamente, aos comandos SELECT, INSERT, DELETE e UPDATE. Com base nesses métodos, temos a possibilidade de listar os registros, acrescentar um novo registro, alterar os dados do registro ou, ainda, remover um registro da base de dados.

Java Persistence Architecture (JPA)

Devido à padronização oferecida com a utilização de mapeamento objeto-relacional e classes DAO, e considerando a grande similaridade existente nos comandos SQL mais básicos, foi simples chegar à conclusão de que seria possível criar ferramentas para a automatização de diversas tarefas referentes à persistência. Surgiram então frameworks de persistência, para as mais diversas linguagens, como Hibernate, Entity Framework, Pony e Speedo, entre diversos outros.

Conheça neste vídeo o framework JPA e aprenda a fazer a configuração da conexão com banco e aplicações de JPA.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Framework JPA

Atualmente, no ambiente Java, concentramos os frameworks de persistência sob uma arquitetura própria, conhecida como Java Persistence Architecture (JPA).

O modelo de programação **anotado** é adotado na arquitetura, simplificando o mapeamento entre objetos do Java e registros do banco de dados.

Tomando como exemplo nossa tabela de alunos, a entidade Java receberia as anotações observadas no fragmento de código seguinte, para que seja feito o mapeamento com base no JPA.

```

java
@Entity
@Table(name = "ALUNO")
@NamedQueries({
    @NamedQuery(name = "Aluno.findAll",                query = "SELECT a FROM Aluno
a"))})
public class Aluno implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "MATRICULA")
    private String matricula;
    @Column(name = "NOME")
    private String nome;
    @Column(name = "ENTRADA")
    private Integer ano;

    public Aluno() {
    }

    public Aluno(String matricula) {
        this.matricula = matricula;
    }
    // getters e setters para os atributos internos
}

```

As anotações utilizadas são bastante intuitivas, como Entity transformando a classe em uma entidade, Table selecionando a tabela na qual os dados serão escritos, e Column associando o atributo a um campo da tabela. As características específicas dos campos podem ser mapeadas por meio de anotações como Id, que determina a chave primária, e Basic, na qual o parâmetro optional permite definir a obrigatoriedade ou não do campo.

Também é possível definir consultas em sintaxe JPQL, uma linguagem de consulta do JPA que retorna objetos, ao invés de registros. As consultas em JPQL podem ser criadas em meio ao código do aplicativo ou associadas à classe com anotações NamedQuery.

Configuração da conexão com banco

Toda a configuração da conexão com banco é efetuada em um arquivo no formato XML com o nome persistence. No mesmo arquivo, deve ser escolhido o driver de persistência. Confira!

```
java
```

```
org.eclipse.persistence.jpa.PersistenceProvider  
  
modelJPA.Aluno
```

Analisando o arquivo persistence do exemplo, temos uma unidade de persistência com o nome **ExemploJavaDB01PU**, sendo a conexão com o banco de dados definida por meio das propriedades url, user, driver e password, com valores equivalentes aos que são adotados na utilização padrão do JDBC. Também temos a escolha das entidades que serão incluídas no esquema de persistência, no caso apenas Aluno, e do provedor de persistência, em que foi escolhido Eclipse Link, mas que poderia ser trocado por Hibernate ou Oracle Top Link, entre outras opções.

Aplicação de JPA

Com os elementos do projeto devidamente configurados, poderíamos utilizá-los para listar o nome dos alunos, por meio do fragmento de código apresentado a seguir:

```
java
```

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("ExemploJavaDB01PU");  
EntityManager em = emf.createEntityManager();  
Query query = em.createNamedQuery("Aluno.findAll",Aluno.class);  
List lista = query.getResultList();  
lista.forEach(aluno->{  
    System.out.println(aluno.getNome());  
});
```

Observe como o uso de JPA diminui muito a necessidade de programação nas tarefas relacionadas ao mapeamento objeto-relacional. Tudo que precisamos fazer é instanciar um EntityManager a partir da unidade de persistência, recuperar o objeto Query e, a partir dele, efetuar a consulta por meio do método getResultList, que já retorna uma lista de entidades, sem a necessidade de programar o preenchimento dos atributos.

Atividade 3

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A

EntityManager.

B Query.

C Transaction.

D Persistence.

E Factory.



A alternativa A está correta.

Os componentes do tipo EntityManager gerenciam as operações sobre as entidades do JPA, trazendo métodos como persist, para incluir um registro na base de dados, ou createQuery, para a obtenção de objetos Query, capazes de recuperar as entidades a partir da base. A função de EntityManagerFactory gera objetos EntityManager, enquanto Persistence faz a relação com as unidades de persistência.

Padrão DAO e JPA na prática

Na aprendizagem de qualquer linguagem de programação, os exercícios práticos são fundamentais para a fixação do conteúdo e entendimento das particularidades, sintaxe e demais recursos.

Acompanhe este vídeo e entenda como implementar um exemplo prático de DAO e JPA.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Nesta prática, utilizaremos como ponto de partida os códigos vistos nos tópicos “Orientação a objetos e o modelo relacional”, “O padrão DAO” e “Java Persistence Architecture (JPA)”. Você deverá implementar os passos a seguir:

- Criar a classe Aluno.
- Realizar a leitura dos dados da tabela, alimentando uma coleção que representará os dados para o restante do programa.
- Listar o conteúdo da tabela.
- Criar a classe de base DAO.
- Criar a classe AlunoDAO.
- Implementar os métodos de manipulação de dados da classe AlunoDAO.
- Exemplificar a manipulação de dados (INSERT/UPDATE/DELETE/READ).

Atividade 4

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A II e III, apenas.

B II e IV, apenas.

C I, II e IV, apenas.

D I, III e IV, apenas.

E I e IV, apenas.



A alternativa C está correta.

As boas práticas indicam que se deve criar uma classe DAO para cada entidade relevante. O objetivo é concentrar as instruções SQL em um único tipo de classe, viabilizando a mudança de fornecedor de banco de dados de forma simples, já que o processo de conexão pode ser encontrado em apenas um método, reutilizado por todo o restante do código.

Nesse exemplo, E representa a classe da entidade e K representa a classe da chave primária. A afirmativa III está errada. A classe é genérica. Os descendentes de GenericDAO deverão implementar métodos abstratos, preocupando-se apenas com os aspectos gerais do mapeamento objeto-relacional e fazendo ampla utilização dos métodos utilitários.

Sistema cadastral simples

Podemos utilizar a classe AlunoDAO na construção de um sistema cadastral simples, em modo texto, com pouquíssimo esforço. Como as operações sobre o banco já estão todas configuradas, nossa preocupação será apenas com a entrada e saída do sistema.

Confira neste vídeo a criação de um sistema cadastral com as funcionalidades básicas, incluindo o gerenciamento das transações.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Classe SistemaEscola

Vamos criar uma classe com o nome SistemaEscola. Observe a seguir sua codificação inicial, incluindo a definição da instância de AlunoDAO e os métodos para exibição de valores, tanto para um aluno quanto para o conjunto completo deles.

```
java
public class SistemaEscola {
    private final AlunoDAO dao = new AlunoDAO();
    private static final BufferedReader entrada =
        new BufferedReader(
            new InputStreamReader(System.in));

    private void exibir(Aluno aluno){
        System.out.println("Aluno: "+aluno.nome+
                           "\nMatricula:
"+aluno.matricula+
                           "\nEntrada:
"+aluno.ano+
                           "\n=====");
    }

    public void exibirTodos(){
        dao.obterTodos().forEach(aluno->exibir(aluno));
    }
}
```

Note como o método exibirTodos utiliza notação lambda para percorrer toda a coleção de alunos, com chamadas sucessivas para o método exibir, o qual recebe uma instância de Aluno e imprime seus dados no console.

Também temos uma instância estática de BufferedReader encapsulando o teclado, com o nome **entrada**, que será utilizada para receber as respostas do usuário nos demais métodos da classe. Vamos verificar sua utilização nos métodos apresentados a seguir, que deverão ser adicionados ao código de SistemaEscola.

java

```
public void inserirAluno() throws IOException{
    Aluno aluno = new Aluno();
    System.out.println("Nome:");
    aluno.nome = entrada.readLine();
    System.out.println("Matricula:");
    aluno.matricula = entrada.readLine();
    System.out.println("Entrada:");
    aluno.ano = Integer.parseInt(entrada.readLine());
    dao.incluir(aluno);
}

public void excluirAluno() throws IOException{
    System.out.println("Matricula:");
    String matricula = entrada.readLine();
    dao.excluir(matricula);
}
```

No método `inserirAluno`, instanciamos um objeto `Aluno` e preenchemos seus atributos com os valores informados pelo usuário, sendo a inclusão na base efetivada ao final, enquanto `excluirAluno` solicita a matrícula e efetua a chamada ao método `excluir`, para que ocorra a remoção na base de dados.

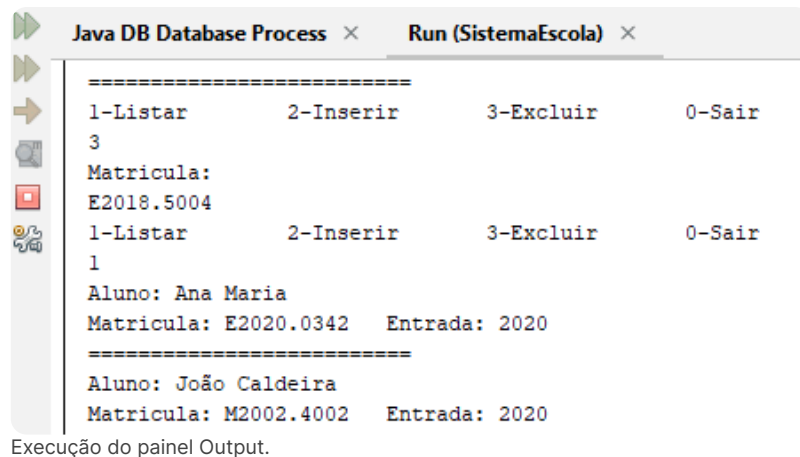
Para finalizar nosso sistema cadastral, precisamos adicionar à classe um método **main**, conforme o trecho de código seguinte.

java

```
public static void main(String args[]) throws IOException{
    SistemaEscola sistema = new SistemaEscola();
    while(true){
        System.out.println(
            "1-Listar\t2-Inserir\t3-Excluir\t0-Sair");
        int opcao = Integer.parseInt(entrada.readLine());
        if(opcao==0)
            break;
        switch(opcao){
            case 1:
                sistema.exibirTodos();
                break;
            case 2:
                sistema.inserirAluno();
                break;
            case 3:
                sistema.excluirAluno();
                break;
        }
    }
}
```

O método `main` permite executar o exemplo, que na prática é bem simples, oferecendo as opções de **listagem**, **inclusão**, **exclusão** e **término**, a partir da digitação do número correto pelo usuário. Feita a escolha da opção, é necessário apenas ativar o método correto da classe, entre aqueles que acabamos de codificar.

Com tudo pronto, podemos utilizar as opções **Build** e **Run File** do NetBeans, causando a execução pelo painel Output, veja!



```
=====
1-Listar      2-Inserir      3-Excluir      0-Sair
3
Matricula:
E2018.5004
1-Listar      2-Inserir      3-Excluir      0-Sair
1
Aluno: Ana Maria
Matricula: E2020.0342  Entrada: 2020
=====
Aluno: João Caldeira
Matricula: M2002.4002  Entrada: 2020
```

Execução do painel Output.

Gerenciamento de transações

O **controle transacional** é uma funcionalidade que permite o isolamento de um conjunto de operações, garantindo a consistência na execução do processo completo. De modo geral, uma transação é iniciada, as operações são executadas, e temos ao final uma confirmação do bloco, com o uso de **commit**, ou a reversão das operações, com **rollback**.



Comentário

Com o uso de transações, temos a garantia de que o banco irá desfazer as operações anteriores à ocorrência de um erro, como na inclusão dos itens de uma nota fiscal. Sem o uso de uma transação, caso ocorresse um erro na inclusão de algum item, seríamos obrigados a desfazer as inclusões que ocorreram antes do erro, de forma programática. Com a transação, será necessário apenas emitir um comando de rollback.

A inclusão de transações em nosso sistema é algo bastante simples, pois basta efetuar modificações pontuais na classe `AlunoDAO`. Acompanhe um dos métodos modificados!

```

java

@Override
public void excluir(String chave) {
    Connection c1 = null;
    try {
        c1 = getConnection();
        c1.setAutoCommit(false);
        PreparedStatement ps = getConnection().prepareStatement(
            "DELETE FROM ALUNO WHERE MATRICULA = ?");
        ps.setString(1, chave);
        ps.executeUpdate();
        c1.commit();
        closeStatement(ps);
    } catch (Exception e) {

        if(c1!=null)
            try {
                c1.rollback();
                c1.close();
            } catch (SQLException e2){}

    }
}

```

Aqui temos a modificação do método excluir para utilizar transações, o que exige que a conexão seja modificada.

Por padrão, cada alteração é confirmada automaticamente, mas utilizando setAutoCommit com valor false, a sequência de operações efetuadas deve ser confirmada com o uso de commit.

Podemos observar, no código, que após desligar a confirmação automática, temos o mesmo processo utilizado antes para geração e execução do SQL parametrizado, mas com a diferença do uso de commit antes de closeStatement. Caso ocorra um erro, será acionado o bloco catch, com a chamada para rollback e o fechamento da conexão.

A forma de lidar com transações no JPA segue um processo muito similar, veja no código.

```

java

public void incluir(Aluno aluno) {
    EntityManagerFactory emf = Persistence.
createEntityManagerFactory("ExemploJavaDB01PU");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    try {
        em.persist(aluno);
        em.getTransaction().commit();
    } catch (Exception e) {
        em.getTransaction().rollback();
    } finally {
        em.close();
    }
}

```

Como já era esperado, o uso de JPA permite um código muito mais simples, embora os princípios relacionados ao controle transacional sejam similares. O controle deve ser obtido com getTransaction, a partir do qual uma transação é iniciada com begin, sendo confirmada com o uso de commit, ou cancelada com rollback.

No fluxo de execução, temos a obtenção do EntityManager, início da transação, inclusão no banco com o uso de persist e confirmação com commit. Caso ocorra um erro, temos a reversão das operações da transação como rollback. Independentemente do resultado, temos a chamada para close, dentro de um bloco finally.

Atividade 1

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A Transaction

B Connection

C EntityManager

D ResultSet

E Rollback



A alternativa B está correta.

No uso do JDBC padrão, temos a gerência de conexões por meio de Connection. Iniciada com o desligamento do commit automático, a transação deve ser confirmada por meio do método commit de Connection, enquanto o uso de rollback desfaz as operações.

Implementação do sistema cadastral simples

Na aprendizagem de qualquer linguagem de programação, os exercícios práticos são fundamentais para a fixação do conteúdo e o entendimento de particularidades, sintaxes e demais recursos.

Assista a este vídeo e entenda como se dá a implementação do sistema cadastral simples, como criar a classe SistemaEscola e como implementar os métodos inserirAluno e excluirAluno, entre outras ações.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Nesta prática, utilizaremos como ponto de partida os códigos vistos no tópico “Sistema cadastral simples”. Você deverá implementar os passos a seguir:

- Criar a classe SistemaEscola.

- Implementar os métodos `inserirAluno` e `excluirAluno`.
- Implementar o método `main`.
- Executar o sistema.
- Incluir as transações na classe `AlunoDAO`.

Atividade 2

Questão 1

HOUE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A I e III, apenas.

B I e II, apenas.

C II e III, apenas.

D II, apenas.

E III, apenas.



A alternativa A está correta.

O método `exibirTodos` utiliza notação `lambda` para percorrer toda a coleção de alunos. O sistema escola deverá ter uma instância da classe `AlunoDAO` para fazer o mapeamento objeto-relacional. A instância `BufferedReader` encapsula o teclado e seu nome é "entrada".

Sistema com JPA no NetBeans

O NetBeans IDE permite o desenvolvimento rápido e fácil de aplicações desktop Java, móveis e web, oferecendo excelentes ferramentas para desenvolvedores de PHP e C/C++. É gratuito e tem código-fonte aberto, além de uma grande comunidade de usuários e desenvolvedores em todo o mundo.

Compreenda neste vídeo como a se dá a implementação do JPA no netbens e conheça o gerador automático de entidades JPA, a geração da Classe `AlunoJpaController` e a adição da biblioteca Java DB Driver.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Gerador automático de entidades JPA

Diversas ferramentas de produtividade são oferecidas pelo NetBeans, e talvez uma das mais interessantes seja o gerador automático de entidades JPA. Para iniciar o processo, vamos criar projeto comum Java, adotando o nome **ExemploEntidadeJPA**, e seguir alguns passos, acompanhe!

Acionar o menu New File, escolhendo a opção Entity Classes from Database.

Selecionar a conexão JDBC correta (escola).

Adicionar as tabelas de interesse, que no caso seria apenas Aluno.

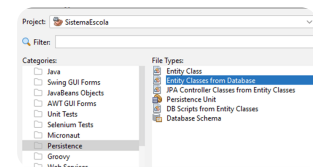
Escrever o nome do pacote (model) e deixar marcada apenas a opção de criação para a unidade de persistência.

Definir o tipo de coleção como List.

Podemos observar os passos descritos na sequência de imagens apresentadas a seguir, confira!

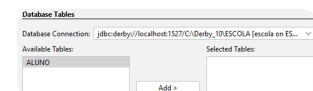
Passo 1

Acionar o menu New File, escolhendo a opção Entity Classes from Database.
Opção Entity Classes from Database.



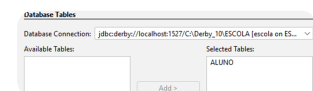
Passo 2

Selecionar a opção JDBC correta (escola).
Opção JDBC correta.



Passo 3

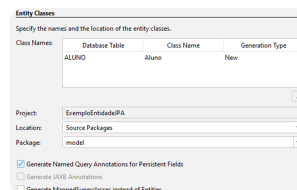
Adicionar as tabelas de interesse, no caso seria Aluno.
Adição da tabela Aluno (tabela de interesse).



Passo 4

Escrever o nome do pacote (model) e deixar marcada apenas a opção de criação para a unidade de persistência.

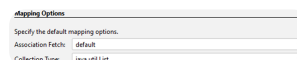
Nome do pacote (model) e seleção da opção de criação para unidade de persistência.



Passo 5

Definir o tipo de seleção como List.

Definição do tipo de seleção.



Após a conclusão do procedimento, teremos a criação da entidade Aluno, no pacote model, com a codificação equivalente àquela apresentada anteriormente, abordando o JPA, bem como o arquivo persistence, em META-INF.

Geração da Classe AlunoJpaController

Com nossa entidade gerada, podemos gerar um DAO de maneira automatizada, por meio dos passos a seguir, confira!

Ao término do processo, teremos um pacote com exceções customizadas, e outro com a classe DAO, com o nome **AlunoJpaController**, no qual temos todas as operações básicas sobre a tabela Aluno, utilizando tecnologia JPA.

Todas as operações para manipulação de dados são feitas a partir do EntityManager. Veja os métodos adequados no próximo quadro.

SQL	DAO	Jpacontroller	EntityManager
INSERT	Incluir	create	persist
UPDATE	Alterar	edit	merge
DELETE	Excluir	destroy	remove

Tabela: Operações para manipulação de dados partir do EntityManager.
Tomás de Aquino

A análise do código completo de AlunoJpaController não é necessária, pois apresenta diversas partes que se repetem, mas o método de remoção é replicado a seguir.

java

```
public void destroy(String id) throws NonexistentEntityException{
    EntityManager em = null;
    try {
        em = getEntityManager();
        em.getTransaction().begin();
        Aluno aluno;
        try {
            aluno = em.getReference(Aluno.class, id);
            aluno.getMatricula();
        } catch (EntityNotFoundException enfe) {
            throw new
NonexistentEntityException("....", enfe);
        }
        em.remove(aluno);
        em.getTransaction().commit();
    } finally {
        if (em != null)
            em.close();
    }
}
```

Inicialmente, é obtida uma instância de EntityManager, permitindo efetuar as operações que se seguem. A transação é iniciada, uma referência para a entidade a ser removida é obtida com getReference e testada em seguida com getMatricula, ocorrendo erro para uma referência nula, ou sendo efetivada a exclusão com remove, no caso contrário.

O método getEntityManager retorna uma instância do gerenciador de entidades, com base no método createEntityManager da fábrica, do tipo EntityManagerFactory, que deve ser fornecida por meio do construtor.

Após utilizar os recursos de automatização do NetBeans, podemos criar uma classe Java com o nome SistemaEscola, muito similar àquela gerada anteriormente, mas com as modificações necessárias para uso do JPA.

```

java

public class SistemaEscola {
    private final AlunoJpaController dao = new AlunoJpaController(
        Persistence.createEntityManagerFactory(
            "ExemploEntidadeJPAPU"));
    private static final BufferedReader entrada =
        new BufferedReader(new InputStreamReader(System.in));

    private void exibir(Aluno aluno){
        System.out.println("Aluno: "+aluno.getNome()+
            "\nMatricula:
"+aluno.getMatricula()+
            "\tEntrada: "+aluno.getEntrada()+
            "\n=====");
    }
    public void exibirTodos(){
        dao.findAlunoEntities().forEach(aluno->exibir(aluno));
    }
    public void inserirAluno() throws Exception{
        Aluno aluno = new Aluno();
        System.out.println("Nome:");
        aluno.setNome(entrada.readLine());
        System.out.println("Matricula:");
        aluno.setMatricula(entrada.readLine());
        System.out.println("Entrada:");
        aluno.setEntrada(Integer.parseInt(entrada.readLine()));
        dao.create(aluno);
    }
    public void excluirAluno() throws Exception{
        System.out.println("Matricula:");
        String matricula = entrada.readLine();
        dao.destroy(matricula);
    }
    public static void main(String args[]) throws Exception{
        SistemaEscola sistema = new SistemaEscola();
        while(true){
            System.out.println(
                "1-Listar\t2-
Inserir\t3-Excluir\t0-Sair");
            int opcao =
Integer.parseInt(entrada.readLine());
            if(opcao==0)
                break;
            switch(opcao){
                case 1: sistema.exibirTodos(); break;
                case 2: sistema.inserirAluno(); break;
                case 3: sistema.excluirAluno(); break;
            }
        }
    }
}

```

As modificações que devem ser efetuadas incluem a utilização de `AlunoJpaController`, inicializado a partir da unidade de persistência `ExemploEntidadeJPAPU`, no lugar de `AlunoDAO`, além de getters e setters no acesso aos atributos de `Aluno`.

Também temos modificações nos nomes dos métodos utilizados para efetuar consultas e modificações no banco, devido à nomenclatura própria do JPA. São adotados **`findAlunoEntities`** para obter o conjunto de registros, **`create`** para a inclusão na base de dados e **`destroy`** para executar a remoção.

Adição da biblioteca Java DB Driver

Enquanto as bibliotecas JPA são adicionadas de forma automática no projeto, teremos de adicionar manualmente a biblioteca Java DB Driver, conforme processo já descrito. Após adicionar a biblioteca, podemos executar o projeto, obtendo o mesmo comportamento do exemplo criado anteriormente, com uso de DAO.

Atividade 3

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A trabalhar com dados no formato JSON.

B importar o modelo ER do banco de dados.

C utilizar bases de dados criadas no Derby.

D conhecer a conexão JDBC com o banco de dados.

E exportar o modelo ER do banco de dados.



A alternativa D está correta.

Utilizando o NetBeans, temos a opção de criação Entity Classes from Database, em que podemos selecionar as tabelas de determinada conexão e deixar que a IDE gere todo o código necessário para as entidades. Tudo que precisamos para recuperar as tabelas é conhecer a conexão JDBC com o banco de dados.

Aplicação JPA no NetBeans

Na aprendizagem de qualquer linguagem de programação, os exercícios práticos são fundamentais para a fixação do conteúdo e entendimento das particularidades, sintaxe e demais recursos.

Confira neste vídeo como aplicar o JPA no NetBeans, como criar o projeto ExemploEntidadeJPA e como acionar o menu New File, escolhendo a opção Entiny Classes from Database.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Nesta prática, utilizaremos como ponto de partida os códigos vistos no tópico “Sistema com JPA no NetBeans”. Você deverá implementar os passos a seguir:

- Criar o projeto ExemploEntidadeJPA.
- Acionar o menu New File, escolhendo a opção Entity Classes from Database.
- Selecionar a conexão JDBC correta (escola).
- Adicionar as tabelas de interesse, no caso apenas Aluno.
- Escrever o nome do pacote (model) e deixar marcada apenas a opção de criação para a unidade de persistência.
- Definir o tipo de coleção como List.
- Acionar o menu New File, escolhendo JPA Controller Classes for Entity Classes.
- Selecionar as entidades, no caso apenas Aluno.
- Escrever o nome do pacote (manager).
- Criar a classe Java SistemaEscola.
- Adicionar da biblioteca Java DB Driver.
- Executar o projeto.

Atividade 4

Questão 1

HOUE UM ERRO AO MIGRAR ESTA ATIVIDADE:

QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO

A II e III, apenas.

B II e IV, apenas.

C I, II e IV, apenas.

D I, III e IV apenas.

E I e IV, apenas.



A alternativa C está correta.

Persistence Unit é um XML que possui o driver para a conexão com o banco de dados. O EntityManagerFactory é o iniciador do JpaController. Você deve usar um EntityManagerFactory para criar instâncias de um EntityManager. Uma das ferramentas mais interessantes do NetBeans é o gerador automático de entidades JPA. O EntityManager é o serviço central para todas as ações de persistência.

Você deve obter acesso de um EntityManager para poder inserir, atualizar, remover e consultar uma entidade no seu banco de dados.

Considerações finais

Analizamos neste conteúdo duas tecnologias para acesso e manipulação de dados no ambiente Java, que são JDBC e JPA. O JDBC é o middleware de acesso a bancos de dados do Java, e JPA é uma arquitetura de persistência baseada em anotações.

Vimos que é possível trabalhar apenas com JDBC, utilizando os componentes ao longo do código, mas a propagação de comandos SQL ao longo dos códigos nos levou à adoção do padrão DAO, organizando nossos códigos e trazendo o nível de padronização exigido para o surgimento de ferramentas como o JPA.

Finalmente, utilizamos os conhecimentos adquiridos na construção de um sistema cadastral simples, criado com programação pura inicialmente, mas que foi refeito com o ferramental de geração do NetBeans, demonstrando como obter maior produtividade.

Explore +

Para saber mais sobre os assuntos tratados neste conteúdo, veja as nossas recomendações!

Referências

CORNELL, G.; HORSTMANN, C. **Core Java**. São Paulo: Pearson, 2010.

DEITEL, P.; DEITEL, H. **Ajax, Rich Internet Applications e Desenvolvimento Web para Programadores**. São Paulo: Pearson, 2009.

DEITEL, P.; DEITEL, H. **Java, Como Programar**. São Paulo: Pearson, 2010.

MONSON-HAEFEL, R.; BURKE, B. **Enterprise Java Beans 3.0.5**. USA: O'Reilly, 2006.