



# Integração com banco de dados em Java

Prof. Denis Gonçalves Cople

Prof. Kleber de Aguiar

## Descrição

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.

## Propósito

Implementar os diversos processos de persistência necessários aos sistemas cadastrais, com base em modelos robustos de programação, permitindo satisfazer a uma demanda de mercado natural na construção de sistemas com esse perfil. Ao final dos estudos, você estará apto a construir sistemas com acesso a bancos de dados relacionais, utilizando tecnologia Java.

## 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

---

## Módulo 1

### Recursos para acessar o banco de dados

Descrever os recursos para acesso a banco de dados no ambiente Java.

---

## Modelo de persistência com mapeamento objeto-relacional

Descrever o modelo de persistência baseado em mapeamento objeto-relacional.

---

## Persistência em banco de dados com Java

Aplicar tecnologia Java para a viabilização da persistência em banco de dados.

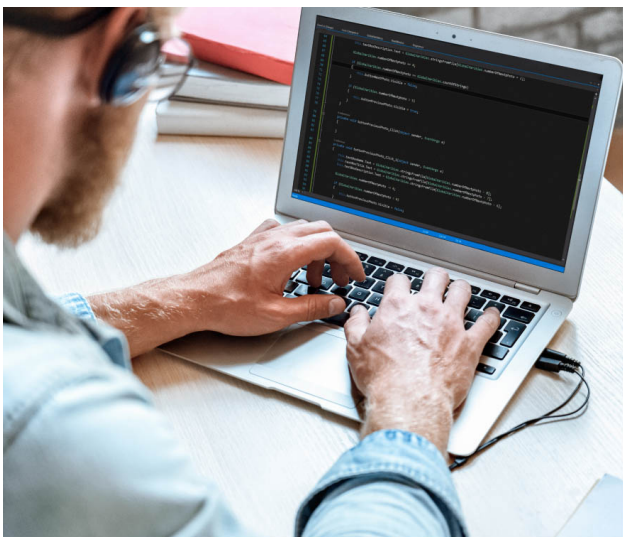
---



## 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 das tecnologias sob análise, construiremos um sistema cadastral simples e veremos como aproveitar os recursos de automatização do NetBeans para construção de componentes JPA.



## 1 - Recursos para acessar o banco de dados

Ao final deste módulo, você será capaz de descrever os recursos para acesso a banco de dados no ambiente Java.

# Conceitos

## Front-end e back-end

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



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 mensagens. Para nossos exemplos, adotaremos o **banco de dados** Derby como back-end.

### Saiba mais

As mensagens são outro 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 que deveríamos ter 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, o que também ocorria com relação aos sistemas de mensagens, em que podemos citar, como exemplos, MQ Series, JBossMQ, ActiveMQ e Microsoft MQ.

## Middleware

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



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**.

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

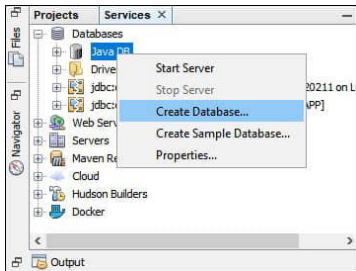
No ambiente Java, temos o **JDBC** (Java Database Connectivity) como middleware para acesso aos diferentes tipos de bancos de dados. Ele permite que utilizemos produtos de diversos fornecedores, sem modificações no código do aplicativo, sendo a consulta e manipulação de dados efetuadas por meio de comandos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

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.

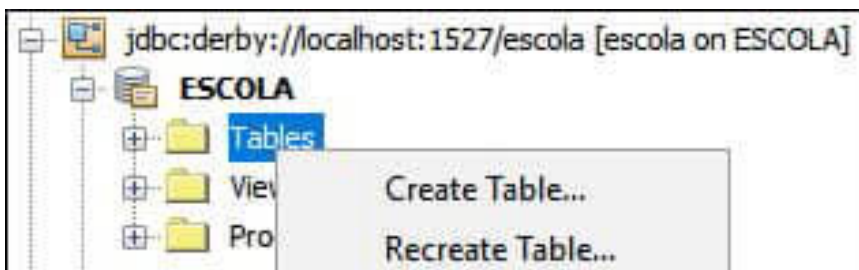
## Banco de dados Derby

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 forma 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**, como ilustrado na imagem, e preencher as informações necessárias para a configuração da nova instância do banco de dados:



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", bem como usuário e senha, onde seria 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, a qual é formada a partir do endereço de rede (localhost), porta padrão (1527) e instância que será utilizada (escola).



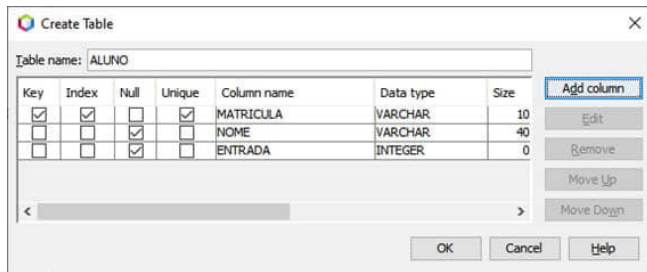
A conexão é aberta com o duplo clique sobre o identificador, ou o clique com o botão direito e escolha da opção **Connect**. Com o banco de dados aberto, podemos criar uma tabela, navegando até a divisão **Tables**, no esquema **ESCOLA**, e utilizando o clique com o botão direito, para acessar a opção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

Na janela que se abrirá, configuraremos a tabela **Aluno**, com os campos definidos de acordo com o seguinte:

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

Tabela: Campos do Aluno.  
Denis Goncalves Cople.

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**.

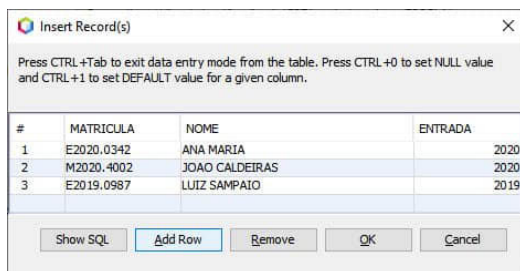


A tabela criada será acessada por meio de um novo nó, na árvore de navegação, abaixo de **Tables**, com o nome **Aluno**. Utilizando o clique com o botão direito sobre o novo nó e escolhendo a opção **View Data**, teremos a abertura de uma área para execução de **SQL** e visualização de dados no editor 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.



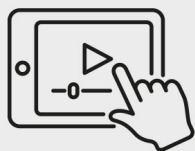
## Java database connectivity (JDBC)



## Componentes do JDBC

No vídeo a seguir apresentamos os componentes oferecidos pelo JDBC.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



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. Existem quatro tipos de **drivers** JDBC, os quais são apresentados a seguir:

#### JDBC-ODBC Bridge



Faz a conexão por meio do ODBC.

#### 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. O processo para utilizar as funcionalidades básicas do JDBC segue quatro passos simples:

1. Instanciar a classe do driver de conexão.
2. Obter uma conexão (Connection) a partir da Connection String, usuário e senha.
3. Instanciar um executor de SQL (Statement).
4. 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:

Java



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, que seria 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:

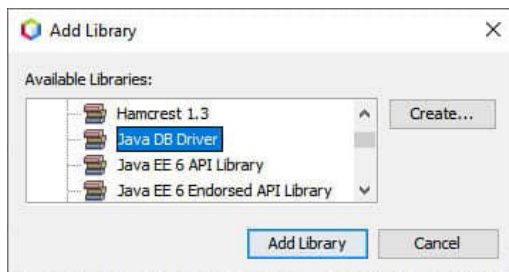
Java



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 lembrando de utilizar 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), e 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**.



Um recurso muito interessante, oferecido por meio do componente **PreparedStatement**, é a definição de comandos SQL parametrizados, os quais são particularmente úteis no tratamento de datas, já que 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. Podemos observar um exemplo da utilização do componente no trecho de código seguinte.

Java



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.

## SQL Injection

É um tipo de ataque a uma aplicação Web, baseado na injeção e execução de instruções SQL mal-intencionadas, que é a linguagem utilizada para troca de informações entre aplicações e bancos de dados relacionais. O SQL Injection visa comprometer a base de dados associada à aplicação Web atacada por ele.

### 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 removerá da base todos os alunos com entrada no referido ano.









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.

Java



Tendo definido 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



Como podemos observar, o código apresenta grande similaridade com o trecho para consulta apresentado anteriormente, mas aqui instanciamos uma coleção, e para cada registro obtido, adicionamos um objeto inicializado com seus dados. Tendo concluído o preenchimento da coleção, aqui 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 alimentar a coleção de entidades, podemos utilizar um código baseado na funcionalidade padrão das coleções do Java.

Java



## Data Access Object (DAO)



### Padrão DAO

O vídeo a seguir apresenta um resumo sobre o padrão de desenvolvimento DAO, além de mostrar como ele pode ser implementado em ambiente Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



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, pois 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 na observação anterior, foi desenvolvido 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.

**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.

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

Java



Observe que 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 a de que viabilizamos 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 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



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

Java



O código ainda não está completo, e certamente apresentará erro, devido ao fato de que não implementamos todos os métodos abstratos definidos, mas já temos o método `obterTodos` codificado nesse ponto. Ele retornará todos os registros de nossa tabela, no formato de um `ArrayList` de entidades do tipo `Aluno`, sendo inicialmente 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!

Note que, 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.

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



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 ao **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, sendo o método **incluir** relacionado ao comando **INSERT**, e o método **alterar** representando as instruções SQL do tipo **UPDATE**.

Uma regra para efetuar mapeamento objeto-relacional, e que é 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, como pode ser observado no trecho de exemplo apresentado a seguir, que permitiria imprimir o nome de todos os alunos da base de dados.

Java



## Java Persistence Architecture (JPA)



### Java Persistence Architecture I

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



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 ferramentais para a automatização de diversas tarefas referentes à persistência. Surgiram **frameworks** de persistência, para as mais diversas linguagens, como Hibernate, Entity Framework, Pony e Speedo, entre diversos outros.

## Saiba mais

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



## Java Persistence Architecture II

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



O modelo de programação **anotado** é adotado na arquitetura, simplificando muito 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



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**.

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.





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.

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:



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`, o qual já retorna uma lista de entidades, sem a necessidade de programar o preenchimento dos atributos.





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



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.

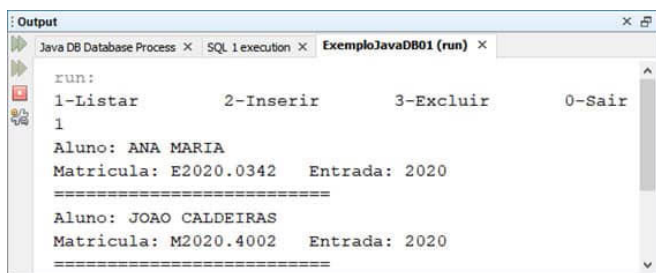
Para finalizar nosso sistema cadastral, precisamos adicionar à classe um método **main**, conforme o trecho de código apresentado a seguir:

Java



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, foi necessário apenas ativar o método correto da classe, dentre aqueles que acabamos de codificar.

Com tudo pronto, podemos utilizar as opções **Build** e **Run File** do NetBeans, causando a execução pelo painel **Output**, como pode ser observado a seguir:



```
run:
1-Listar      2-Inserir      3-Excluir      0-Sair
1
Aluno: ANA MARIA
Matricula: E2020.0342   Entrada: 2020
=====
Aluno: JOAO CALDEIRAS
Matricula: M2020.4002   Entrada: 2020
=====
```

## Gerenciamento de transações



### Gerenciamento de transações I

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



O **controle transacional** é uma funcionalidade que permite o isolamento de um conjunto de operações, garantindo a consistência na execução do processo completo. De forma 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**.

## Saiba mais

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, mas com a transação será necessário apenas emitir um comando de rollback.



## Gerenciamento de transações II

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A inclusão de transações em nosso sistema seria algo bastante simples, pois bastaria efetuar modificações pontuais na classe **AlunoDAO**. Podemos observar um dos métodos modificados a seguir.

Java



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, como pode ser observado no trecho de código a seguir:

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**, e independentemente do resultado temos a chamada para **close**, dentro de um bloco **finally**.

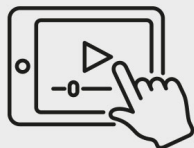
## Sistema com JPA no NetBeans



### Automatização do NetBeans

O vídeo a seguir explica como fazer para obter uma descrição das funcionalidades do NetBeans para geração de entidades a partir do banco de dados, bem como controladores JPA a partir das entidades, e demonstração do uso do ferramental na construção rápida de aplicativos cadastrais.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



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 estes passos:

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

Selecionar a conexão JDBC correta (escola).

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

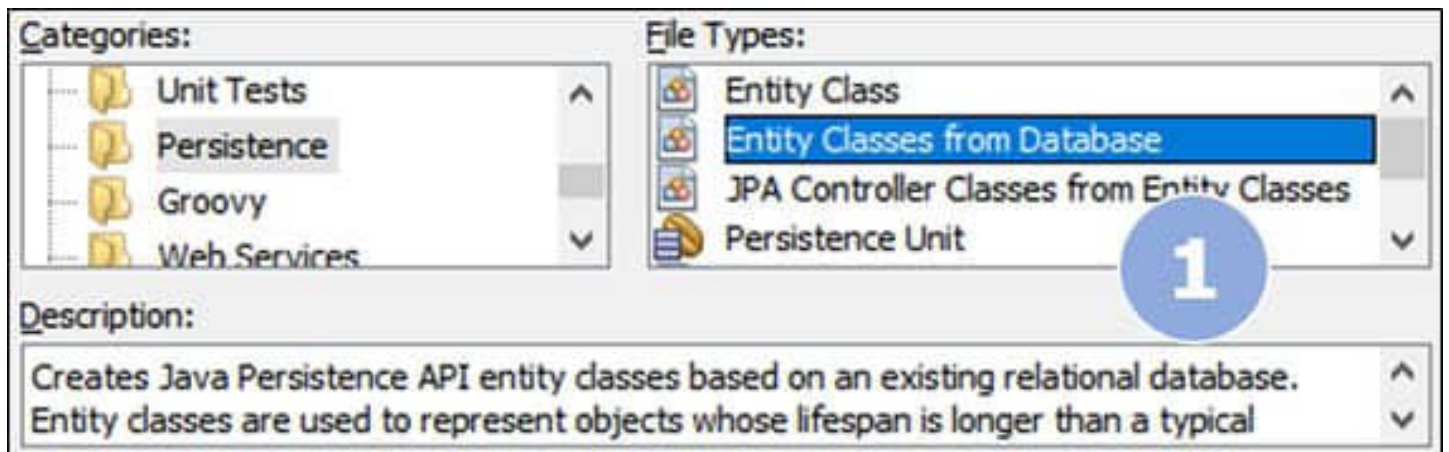
Escrever o nome do pacote (model) e deixar marcada apenas a opção de criação para a unidade de persistência.

Definir o tipo de coleção como List.

Podemos observar os passos descritos na sequência de imagens apresentadas a seguir:

## Passo 1

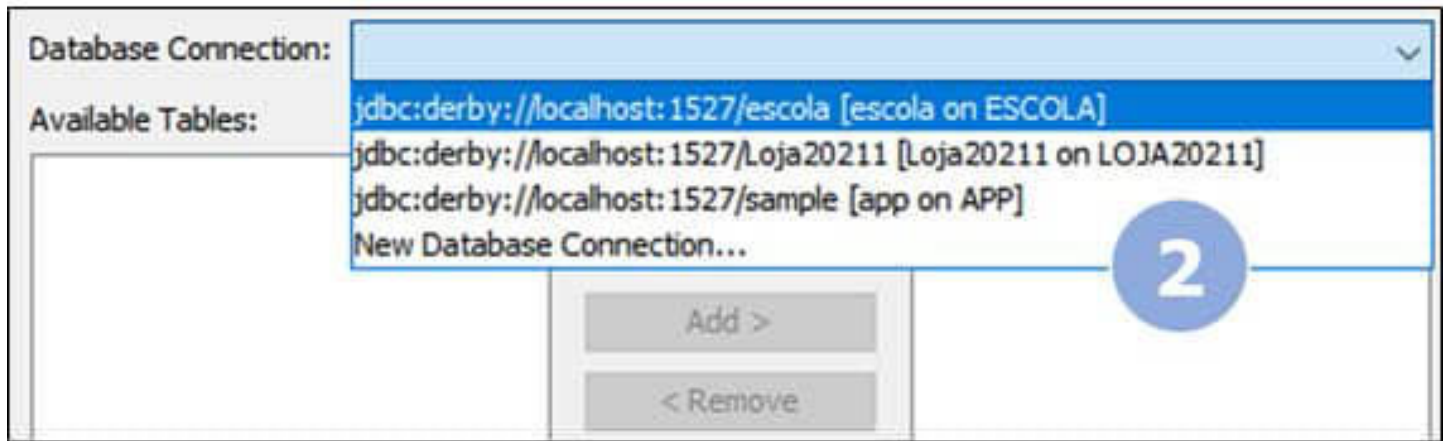
Acionar o menu New File, escolhendo a opção Entity Classes from Database.



## Passo 2

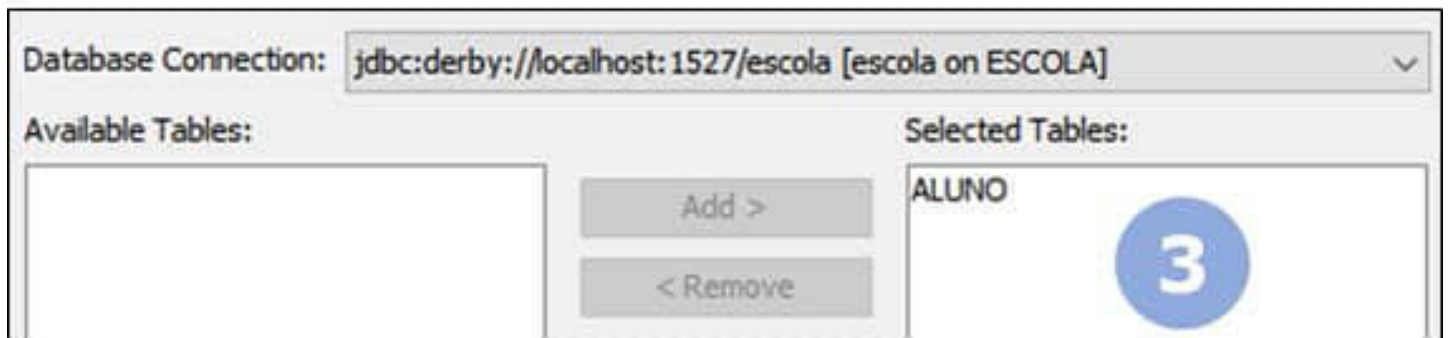
Selecionar a opção JDBC correta (escola).





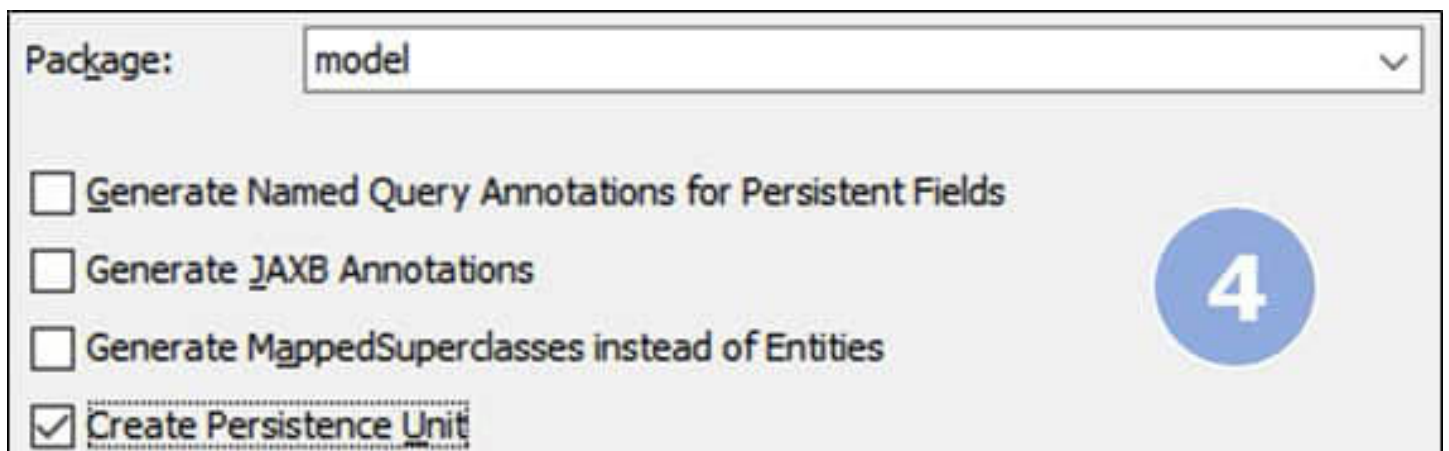
### Passo 3

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



### Passo 4

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



### Passo 5

Definir o tipo de seleção como List.

Specify the default mapping options.

Association Fetch: default

Collection Type: java.util.List

5

Após a conclusão do procedimento, teremos a criação da entidade **Aluno**, no pacote **model**, com a codificação equivalente à que foi apresentada no módulo 2, abordando o JPA, bem como o arquivo **persistence**, em **META-INF**.

Com nossa entidade gerada, podemos gerar um DAO de forma automatizada, por meio dos seguintes passos:

1

Acionar o menu New File, escolhendo JPA Controller Classes for Entity Classes.

2

Selecionar as entidades, o que no caso seria apenas Aluno.

3

Escrever o nome do pacote (manager).

Novamente, é possível observar os passos descritos por meio de uma sequência de imagens capturadas da tela do gerador automático de entidades JPA:

## Passo 1

Acionar o menu New File, escolhendo JPA Controller Classes for Entity Classes.

Filter:

**Categories:**

- Unit Tests
- Persistence
- Groovy
- Web Services

**File Types:**

- Entity Class
- Entity Classes from Database
- JPA Controller Classes from Entity Classes
- Persistence Unit

**Description:**

Creates a set of JPA controller classes and related classes from a set of entity classes.

Passo 2

Selecionar as entidades, o que no caso seria apenas Aluno.

**Entity Classes**

Available Entity Classes:

Selected Entity Classes:

model.Aluno

Add >

< Remove

Passo 3

Escrever o nome do pacote (manager).

**Project:** ExemploEntidadeJPA

**Location:** Source Packages

**Package:** manager

Ao término do processo, teremos um pacote com exceções customizadas, e outro com a classe DAO, com o nome **AlunoJpaController**, onde 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, em que os métodos adequados podem ser observados no quadro apresentado a seguir:

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

Tabela: Correlação de métodos para diferentes componentes.  
Denis Gonçalves Cople.

A análise do código completo de **AlunoJpaController** não é necessária, já que apresenta diversas partes que se repetem, mas o método de remoção é replicado aqui:

Java



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 à que foi gerada anteriormente, mas com as modificações necessárias para uso do JPA.

Java



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, sendo adotados **findAlunoEntities** para obter o conjunto de registros, **create** para a inclusão na base de dados e **destroy** para executar a remoção.

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.

## Falta pouco para atingir seus objetivos.

### Vamos praticar alguns conceitos?

#### Questão 1

Por meio do controle transacional, é possível gerenciar blocos de operações como uma ação única, que pode ser desfeita pelo banco de dados sem maiores esforços em termos de programação. Em termos de JDBC padrão, qual classe é responsável pela transação?

A Transaction

B Connection



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.



## Podcast

Para finalizar nosso conteúdo, ouça o podcast a seguir, que apresenta uma visão geral sobre integração com banco de dados em Java.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



## Referências

CORNELL, G.; HORSTMANN, C. **Core Java**. 8. ed. São Paulo: Pearson, 2010.

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

DEITEL, P.; DEITEL, H. **Java, Como Programar**. 8. ed. São Paulo: Pearson, 2010.

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

## Explore +

Para saber mais sobre os assuntos tratados neste conteúdo, leia:

Guia da Oracle sobre uso de JDBC com SWING;

Guia da Oracle sobre transações no JDBC;

Transações e concorrência no Hibernate;

Tutorial de JPA com NetBeans.