Módulo 6

Programação WEB

Lição 5

Conexão com Banco de Dados: SQL e JDBC

Versão 1.0 - Nov/2007

Autor
Daniel Villafuerte

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Allan Souza Nunes
Allan Wojcik da Silva
Angelo de Oliveira
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Denis Mitsuo Nakasaki
Emanoel Tadeu da Silva Freitas
Felipe Gaúcho
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Luciana Rocha de Oliveira
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauro Cardoso Mortoni
Paulo Oliveira Sampaio Reis
Pedro Henrique Pereira de Andrade
Ronie Dotzlaw
Sergio Terzella
Thiago Magela Rodrigues Dias
Vanessa dos Santos Almeida
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

A maioria das aplicações WEB utiliza algum tipo de armazenamento externo para gerar seu conteúdo dinamicamente. Esse meio de de armazenamento externo, na maioria das vezes, é encontrado na forma de um banco de dados relacional, devido a sua simplicidade e facilidade para se extrair dados.

Essa extração de dados de um banco relacional é realizado com o uso do SQL – *Structure Query Language* (Linguagem de Pesquisa Estruturada). Esta linguagem define uma sintaxe e várias palavras chaves que podem ser entendidas por sistemas de bancos de dados. A grande maioria dos bancos relacionais provê um programa cliente que permite a entrada de comandos em SQL em que o resultado é exibido na tela. Entretanto, aplicações WEB não podem utilizar esta interface em seus programas.

A API JDBC, que faz parte da plataforma J2EE, provê aos desenvolvedores um padrão, uma maneira programática de relacionamento com os bancos de dados relacionais. Fazendo uso desta API, os desenvolvedores podem executar consultas SQL e fazer uso de seus resultados para gerar conteúdo dinâmico para seu cliente.

Ao final desta lição, o estudante será capaz de:

- Construir um objeto da classe *Connection* utilizando a classe *DriverManager* ou *DataSource*
- Construir um objeto da classe *Statement* usando o método *createStatement*() disponível na classe *Connection*
- Executar consultas SQL usando um objeto da classe *Statement* e recuperar os resultados
- Encerrar os objetos do banco de dados

2. Bancos de dados Relacionais

Bancos de dados relacionais são a escolha de armazenamento da maioria das aplicações baseadas na WEB que necessitam de um dinamismo sem seu conteúdo. A sintaxe básica necessária para recuperar e manipular dados que se encontram armazenados é de fácil aprendizado. Paralelamente, existe uma vasta indústria de suporte, várias opções de personalização disponíveis, que necessitam de pouco ou nenhum recurso técnico que podem ser encontrados na *Internet*.

Assim como o nome já diz, um banco de dados relacional armazena dados como uma série de informações relacionadas. Grupos relacionados são expressos na forma de tabelas. Cada tabela contém colunas que definem as propriedades de cada grupo de dados armazenados.

Este conceito pode ser visualizado pelo exemplo a seguir:

id	**nome**	**endereco**	**NumContato**
14627895	Duke	California	0924562788
65248987	Kimi	Finland	8687243217

Figura 1: Exemplo da estrutura de uma tabela

Neste exemplo, vemos uma tabela que é utilizada para armazenar dados do usuário. A tabela define 4 colunas: uma coluna id que armazena uma identificação que é única para cada usuário, uma coluna de nome, endereço, e uma outra para o número de contato. Cada linha da tabela representa uma única informação. Isto significa que existe um usuário chamado Duke, ao qual o endereço é na Califórnia, que possui o id 14627895 e o número de contato 0924562788.

As tabelas utilizadas num sistema de bancos de dados não são tão simples quanto a apresentada acima. As tabelas definidas num banco de dados são geralmente montadas com *constraints* lógicas que servem para preservar a consistência dos dados. Uma *constraint* é uma restrição de um tipo de dado: cada coluna é definida para ser de um tipo de dados específico. O sistema automaticamente rejeita a inserção de novos dados que não sejam compatíveis com o tipo de dado definido pela estrutura da tabela. Por exemplo, a coluna id pode ser definida internamente para ser do tipo *integer*. Tentar inserir novas linhas que contenham caracteres em seus valor para o campo id causará uma falha na inserção. Outra *constraint* geralmente imposta numa coluna é a unicidade: se uma coluna é definida como sendo "única" o sistema não aceitará a inserção de um novo dado que contenha um valor já existente no sistema.

As mecânicas de definição de uma tabela estão além do escopo desta discussão e serão deixados para posterior compreensão. Esta discussão será focada no acesso a dados e modificação numa dada tabela.

3. Sentenças SQL

Como já foi mencionado anteriormente, operações em bancos de dados relacionais são realizadas por meio de SQL. Existem vários tipos de sentenças SQL. Apesar disto, duas serão abordadas nesta lição.

3.1. Recuperação de Dados

Este tipo de sentença é focada na leitura de dados de uma ou mais tabelas num banco de dados. Consultas deste tipo podem ser configuradas para que retornem TODOS os dados relacionados em uma tabela específica (ou grupo de tabelas), ou podem ser parametrizadas para que somente algumas linhas ou colunas sejam recuperadas por meio de valores informados.

Somente um comando SQL se encaixa neste tipo: o comando SELECT.

3.1.1. Comando SELECT

O comando SELECT é utilizado para consultas ao banco de dados sobre informações que serão retornadas em conjuntos de linhas.

A forma básica de um comando SELECT é:

```
SELECT column(s) FROM tablename WHERE condition(s)
```

Na sintaxe acima, SELECT, FROM e WHERE são palavras chaves SQL, enquanto que *column*, *tablename* e *condition* são valores especificados pelo desenvolvedor.

- SELECT – marca o início do comando SELECT
- column (coluna) – o nome das colunas da qual os valores serão retornados. Se todas as colunas devem ser recuperadas, um asterisco(*) pode se utilizado no lugar do nome das colunas.
    - Recuperando múltiplas colunas -> SELECT id, nome, endereco FROM ...
    - Recuperando todas as colunas -> SELECT * FROM ...
- FROM – indica o nome da tabela de onde os dados serão recuperados. Um mecanismo para a recuperação de dados de múltiplas tabelas também está incluído na linguagem SQL, e será discutido em detalhes mais a frente.
- WHERE – é opcional e especifica condições que os registro deverão atender antes de serem incluídos no resultado. Mais de uma condição pode ser especificada; neste caso, as condições são separadas por uma palavra chave AND (exclusiva) ou OR (inclusiva) que executam as mesmas funções de seus equivalentes lógicos. Outros tipos de condições são permitidas e serão discutidas futuramente.

3.1.2. A cláusula FROM

A cláusula FROM numa sentença SELECT define a(s) tabela(s) de onde os dados serão reunidos. Se os dados vierem somente de uma única tabela, então o nome daquela tabela é simplesmente informado. Entretanto, se o dado vier de mais de uma tabela, uma operação conhecida como *table join (junção de tabelas) deve ser executada.*

Table joins podem ser executadas de inúmeros modos:

User

userid	**name**	**address**	**contactnum**
14627895	Duke	San Francisco	0924562788
65248987	Kimi	Finland	8687243217
84321874	Dante	San Jose	6365498428

UserDownloads

userid	**downloaditem**	**downloaddate**
14627895	Courseware Notes	Dec. 19, 2005
36542036	Exercises	Feb. 11, 2006
84321874	Slides	March 13, 2006

- listando todas as tabelas que serão unidas. No comando SQL poderiam ser separadas por

vírgulas. Apesar de ser mais simples, este método é também mais lento em termos de performance. O resultado obtido é um produto cartesiano das tabelas.

Exemplo: Dadas duas tabelas, *User* e *UserDownloads*, a união é executada por: SELECT * FROM user, userdownload [WHERE ...]

Se a vírgula for utilizada como delimitador, o resultado será como o mostrado a seguir:

userid	name	address	contactnum	userid	downloaditem	downloaddate
14627895	Duke	San Francisco	0924562788	14627895	Courseware Notes	Dec. 19, 2005
14627895	Duke	San Francisco	0924562788	36542036	Exercises	Feb. 11, 2006
14627895	Duke	San Francisco	0924562788	84321874	Slides	March 13, 2006
65248987	Kimi	Finland	8687243217	14627895	Courseware Notes	Dec. 19, 2005
65248987	Kimi	Finland	8687243217	36542036	Exercises	Feb. 11, 2006
65248987	Kimi	Finland	8687243217	84321874	Slides	March 13, 2006
84321874	Dante	San Jose	6365498428	14627895	Courseware Notes	Dec. 19, 2005
84321874	Dante	San Jose	6365498428	36542036	Exercises	Feb. 11, 2006
84321874	Dante	San Jose	6365498428	84321874	Slides	March 13, 2006

3.1.3. União de Tabelas

- JOIN

Fazendo uso de um dos muitos JOINs. A sintaxe é: *table1* JOIN *table2* WHERE *condition*.

Condição especifica quais linhas de ambas as tabelas serão unidas para que o resultado possa ser gerado.

- LEFT JOIN

A performance é similar ao JOIN, exceto que todas as linhas de *table1* serão retornadas a união, mesmo que não exista uma linha correspondente em *table2* que atenda a condição. Utilizando o LEFT JOIN nestas tabelas, com a condição User.userid = UserDownloads.userid:

userid	name	address	contactnum	userid	downloaditem	downloaddate
14627895	Duke	San Francisco	0924562788	14627895	Courseware Notes	Dec. 19, 2005
65248987	Kimi	Finland	8687243217			
84321874	Dante	San Jose	6365498428	84321874	Slides	March 13, 2006

- RIGHT JOIN

A performance é similar ao JOIN, exceto que todas as linhas de *table2* serão retornadas a união, mesmo que não exista uma linha correspondente em *table1* que atenda a condição. Utilizando o RIGHT JOIN nestas tabelas, com a mesma condição:

userid	name	address	contactnum	userid	downloaditem	downloaddate
14627895	Duke	San Francisco	0924562788	14627895	Courseware Notes	Dec. 19, 2005
				36542036	Exercises	Feb. 11, 2006
84321874	Dante	San Jose	6365498428	84321874	Slides	March 13, 2006

- INNER JOIN

Somente as linhas de *table1* e *table2* que atendam à condição serão consideradas no resultado. Utilizando o INNER JOIN nestas tabelas, com a mesma condição:

userid	name	address	contactnum	userid	downloaditem	downloaddate
14627895	Duke	San Francisco	0924562788	14627895	Courseware Notes	Dec. 19, 2005
84321874	Dante	San Jose	6365498428	84321874	Slides	March 13, 2006

Na maioria dos casos, a utilização de um INNER JOIN trará os resultados mais relevantes para as operações da junção de tabelas. Entretanto, em casos onde as linhas de uma determinada tabela devem aparecer no resultado não importando as condições da pesquisa, um LEFT JOIN ou RIGHT JOIN é considerado mais apropriado. Evite utilizar a vírgula como delimitador sempre que possível. Apesar da escrita ser mais fácil, a performance será muito baixa, valendo a pena gastar um pouco mais de tempo e utilizar o JOIN correta.

3.1.4. A cláusula WHERE

A cláusula WHERE numa sentença SELECT especifica uma condição que deve ser obedecida pelos registros numa tabela qualquer para que apareçam no resultado. Existem muitos operadores que podem ser utilizados para especificar uma condição:

- = - Verifica a igualdade entre dois operandos dados.
- <=, < - Verifica se o primeiro operando é menor ou igual, ou menor que o segundo operando.
- >=, > - Verifica se o primeiro operando é maior ou igual, ou maior que o segundo operando.
- like - Executa uma comparação entre os caracteres dos operandos. Utilizando este operador, dois caracteres podem ser utilizados para representar um valor desconhecido.
    - % - Aplica-se a *Strings* de qualquer tamanho. Ex: 'A%s' encontrará todas as ocorrências de palavras que se iniciem por A e terminem com s.
    - _ - Aplica-se a qualquer caractere. Ex: 'b_t' irá encontrar todas as ocorrências de palavras que se iniciem com 'b' e terminem com 't', mas que possuam **somente** um caractere entre eles.

A seguir, são dados alguns exemplos simples do uso da sentença SELECT.

- Recuperar todas as linhas e colunas da tabela de *users*.

```
SELECT * from users;
```

- Pesquisar pelo endereço dos usuários que tenham o nome de *Smith*.

```
SELECT address from users where name = 'Smith';
```

- Pesquisar todas as linhas da tabela users com o nome iniciado por 'S'.

```
SELECT * from users where name like 'S%';
```

A linguagem SQL não é case-sensitive em relação as suas palavras-chave (diferente de Java). Entretanto, é case-sensitive em relação aos parâmetros de comparação. A seguinte sentença trará uma coleção de registros diferente da apresentada anteriormente:

```
SELECT * from users where name ='sMith';
```

3.2. Manipulação de Dados

As sentenças que se encaixam neste tipo são utilizadas para modificar o estado dos dados no banco de dados. Existem muitas sentenças, cada uma sendo utilizada para uma manipulação especifica.

3.2.1. Comando INSERT

O comando INSERT é utilizado para inserir novos registros em uma determinada tabela num banco de dados.

```
INSERT INTO table-name VALUES (value1, value2, ...)
```

Acima é possível observar a estrutura básica do comando INSERT, onde *table-name* é o nome da tabela que irá armazenar os novos dados inseridos. Os valores após a palavra-chave VALUES é delimitado por vírgulas e serão adicionados à tabela. Em casos como este em que uma só tabela é

referenciada, a associação é feita com os campos passados como parâmetro baseados na ordenação das colunas da tabela.

Por exemplo, numa tabela chamada *users*, com as colunas *userid*, *name*, *address*, nesta ordem, o comando a seguir adicionará uma nova linha na tabela:

```
INSERT INTO users VALUES(199700651, 'Jedi Master', 'UP Ayala Technopark');
```

É muito importante notar que toda sentença que faz uso do comando INSERT, deve seguir todas as regras de integridade definidas pela tabela. Isto é, se um campo numa tabela é definido para ser **not null**, qualquer tentativa de inserir um valor nulo neste campo causará num erro.

3.2.2. Comando UPDATE

O comando UPDATE realiza uma modificação de registros em uma determinada tabela.

```
UPDATE table-name SET column-value(s) WHERE condition(s)
```

Acima temos a forma básica do comando UPDATE, onde *table-name* é o nome da tabela que contém as linhas as serem modificadas e *column-values* é uma lista com os novos valores das colunas na tabela. Opcionalmente, uma lista de condições pode ser adicionada para especificar quais linhas da tabela serão modificadas. Se nenhuma condição for passada, o comando UPDATE será efetuado para todas as linhas existentes numa dada tabela.

Todo comando UPDATE deve seguir as regras de integridade impostas pelo banco de dados. Por exemplo, alterar o valor de um campo marcado como **not null** para um valor nulo resultará na não execução da sentença e numa mensagem de erro vinda do banco de dados relacional.

3.2.3. Comando DELETE

O comando DELETE elimina registros de uma determinada tabela. Observe que é oposto ao comando INSERT que adiciona novos registro a tabela.

```
DELETE FROM table-name WHERE condition(s)
```

Acima é apresentada a forma básica do comando DELETE, sendo *table-name* o nome da tabela que contém os dados que se deseja eliminar. Uma lista de condições pode ser opcionalmente especificada. Se nenhuma condição for dada, o comando irá excluir todos os registros da tabela mencionada.

4. JDBC

Java fornece uma biblioteca padrão para que o acesso a bancos de dados seja efetuado, a chamada *Java Database Connectivity* ou, simplesmente, JDBC. Por meio desta os desenvolvedores podem acessar bases de dados não importando quem seja seu fabricante; os desenvolvedores de um JDBC provêem a implementação para as interfaces definidas nesta API, fornecendo o mesmo grupo de funcionalidades ao desenvolvedor do sistema.

As seguintes classes estão na API JDBC:

- *java.sql.Connection* – Representa a conexão com o banco de dados. Encapsula os detalhes de como a comunicação com o servidor é realizada.
- *java.sql.DriverManager* – Gerencia os drivers JDBC utilizados pela aplicação. Em conjunto com o endereço e a autenticação, pode fornecer objetos de conexão.
- *javax.sql.DataSource* – Abrange os detalhes (endereço, autenticação) de como a conexão com o banco de dados é obtida. É o mais recente e o preferido método de obtenção de objetos de conexão.
- *java.sql.Statement* – Fornece meios ao desenvolvedor para que se possa executar comandos SQL.
- *java.sql.ResultSet* – Representa o resultado de um comando SQL. Estes objetos normalmente são retornados por métodos.

4.1. java.sql.DriverManager

Utilizando esta classe, o desenvolvedor pode retornar um objeto de conexão que pode ser usado para executar tarefas relativas ao banco de dados. Dois passos são necessários para tal:

- Primeiro, o *driver* JDBC deve estar registrado com *DriverManager*. Isto pode ser feito utilizando o método *Class.forName* que carrega a classe do driver para a memória.
- Segundo, utilizando o método *getConnection*, mediante informação de uma URL, assim como a senha e o nome do usuários autenticados no banco de dados. A URL deve seguir a sintaxe requisitada pela implementação do banco de dados.

Abaixo vemos um exemplo de como se obtém uma conexão com um banco de dados PostgreSQL. Novamente, a URL e o *driver* específicos para a implementação são utilizados. Para outros bancos de dados, verifique a documentação fornecida.

```
String jdbcURL = "jdbc:postgresql://localhost:5432/jedi-db";
String user = "jedi";
String password = "j3d1master";
Connection conn = null;

try {
  Class.forName("org.postgresql.Driver");
  conn = DriverManager.getConnection(url, user, password);
  ...
} catch (SQLException e) {
  // caso ocorra erro de SQL
}
```

Apesar de ser um método válido para a obtenção de um objeto de conexão, necessita que o desenvolvedor faça o rastreamento de alguns detalhes, tais como: o nome da classe do *driver*, a *url* necessária para se efetuar a conexão, o nome de usuário e senhas para o acesso ao banco. Estes detalhes são os que mais estão sujeitos a mudanças a cada compilação da aplicação. Além disso, gerenciar a url e o nome do *driver* no código dificulta a aplicação a trocar a implementação de sua base de dados, se isto vier a ser necessário.

4.2. javax.sql.DataSource

Esta é uma interface definida na API JDBC desde sua versão 2. Hoje em dia, este é o modo recomendado para que se obtenham objetos de conexão. Recuperar objetos de conexão é uma tarefa bastante direta: simplesmente chame o método *getConnection()* de uma instância válida desta interface.

Sendo *DataSource* uma interface, uma instância não pode ser simplesmente criada pelo desenvolvedor utilizando o operador **new**. É recomendado que se deixe para o servidor da aplicação que gerencie a criação de objetos *DataSource*. Com isto permite-se que o servidor adicione algumas funcionalidades, tais como a *connection pooling* (reserva de conexões) de uma maneira que é transparente tanto para o desenvolvedor, quanto para o usuário final.

4.2.1. Configurando o *Datasource* no Sun Application Server

Cada servidor possui seu próprio procedimento para a configuração e gerenciamento de *DataSources*. São três passos para se configurar um *DataSource* no AppServer:

- Registrar o arquivo JAR contendo o *driver* JDBC
- Criar o *pool* de conexões para o banco de dados
- Registrar um *DataSource* que utilize este *pool*

4.2.2. Registrando o arquivo JAR

O primeiro passo é acessar o console de administração do servidor e disponibilizar o arquivo de conexão com o banco de dados para o *Sun Application Server*. Para isso, basta copiar o arquivo .jar para a pasta \lib aonde está instalado o servidor. Por exemplo, com o servidor *Glassfish* instalado em conjunto com o NetBeans 6.0 pode ser encontrado no Windows na pasta:

```
C:\Program Files\glassfish-v2\
```

4.2.3. Criando um pool de conexão

Para iniciar a criação de um pool de conexão, selecione **Resources** | **JDBC** | **Connection Pools** no painel a esquerda. Na tela exibida, pressione o botão **New** e uma tela similar a **Figura 2** será mostrada.

Em **Name** entre com o nome como este pool de conexão será conhecido (exemplo: jediPool). Em **Resource Type** selecione **javax.sql.DataSource**. No **Database Vendor** selecione o banco de dados utilizado, caso o banco que esteja utilizando não apareça na lista deixe esta opção em branco.

Pressione o botão **Next**, no campo **Datasource Classname**, caso não venha preenchido por padrão, informe o nome da classe do seu banco, para o PostGreSQL:

```
org.postgresql.jdbc3.Jdbc3PoolingDataSource
```

Observe as propriedades para associar com este *pool* de conexões. Os parâmetros seguintes necessitam ter seus valores fornecidos:

- Password – Senha do banco de dados
- ServerName – Nome do servidor do banco de dados
- PortNumber – Número da porta
- DatabaseName – Nome do Banco de dados
- User – Nome do usuário

Depois de completar os valores, pressione o botão **Finish**.

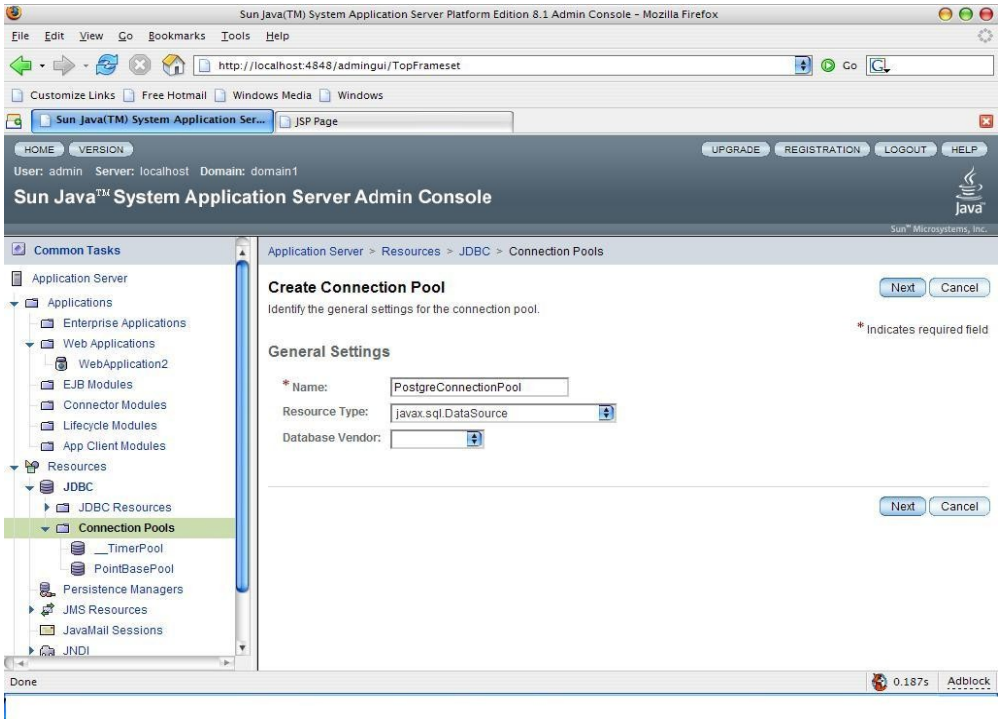

Figura 2: Criando um Connection Pool

Selecione **Resources** | **JDBC** | **Connection Pools** | **jediPool** e pressione o botão **ping** a seguinte mensagem deve ser mostrada indicando que o pool de conexão foi criado com sucesso:

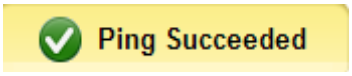

Figura 3: Conexão criada com sucesso

4.2.4. Registrando o Datasource

Para registrar um *datasource*, selecione **Resources** | **JDBC** | **JDBC Resources** no lado esquerdo do painel. Na tela que aparece, pressione o botão *New*.

Os campos devem ser preenchidos da seguinte maneira:

- **Nome do JNDI** – entre como nome lógico da sua aplicação recuperará o *DataSource*. É recomendado que este nome tenha JDBC como seu prefixo para facilitar para futuros administradores do servidor para identificar este elemento de fonte JDBC (ex. **jdbc/jediPool**)
- **Nome do Pool** – selecione o nome do pool de conexões criada anteriormente
- **Descrição** – entre com o texto descrevendo o *DataSource* (opcional)

Pressione o botão **OK** para finalizar.

Recuperando uma instância de um *DataSource* de uma aplicação servidora é simples e pode ser realizada usando unicamente algumas linhas de código da API JNDI. *Java Naming Directory* Interface (JNDI) é uma API Java padrão para acessar diretórios. Um diretório é um local centralizado onde a aplicação Java pode recuperar recursos externos usando um nome lógico.

Detalhes adicionais para JNDI e como este trabalho estão além do escopo desta lição. A única coisa que necessitamos conhecer é que o servidor de aplicação mantém um diretório para que seja publicado o *DataSource* que foi configurado anteriormente. Nossa própria aplicação pode executar um simples *lookup* de nome neste diretório para recuperar o recurso.

Para nossas finalidades, é bastante criarmos um contexto JNDI usando o construtor padrão. Este contexto JNDI abstrai os detalhes da conexão para o diretório, fazendo *lookup* do recurso numa simples chamada num método singular. Tome nota que o nome usado para o *lookup* na fonte deve ter o mesmo nome usado na configuração no *DataSource*.

```
...
Context ctxt = null;
DataSource ds = null;

try {
  // criar uma instância de um contexto JNDI
  ctxt = new InitialContext();

  // retornar o DataSource de um diretório com um nome lógico
  ds = (DataSource)ctxt.lookup("jdbc/PostgreSQLDS");
} catch (NamingException ne) {
  System.err("Specified DataSource cannot be found");
}
```

Uma vez que temos um exemplo de instância DataSource válido, pequemos um objeto Connection tão simples quanto.

```
Connection conn = ds.getConnection();
```

4.3. java.sql.Connection / java.sql.Statement

Os objetos *java.sql.Connection* representam conexões reais ao banco de dados. Uma vez que temos um exemplo de um objeto podemos criar uma instância do objeto Statement, para executar as declarações SQL.

O objeto do tipo *Statement* provê diversos métodos para executar as declarações SQL. Os métodos mais utilizados são:

- *executeQuery* – utilizado para a instruções de pesquisa no banco de dados (comando SELECT) e retorna o resultado da operação em um objeto do tipo *ResultSet*.
- *executeUpdate* – utilizado para as instruções de modificação do banco de dados (comandos CREATE, DROP, ALTER, INSERT, UPDATE ou DELETE) e retorna um tipo primitivo **int** com o número de linhas afetadas.

Abaixo está um exemplo de pedaço de um código mostrando como realizar este procedimento:

```
Context ctxt = null;
DataSource ds = null;
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
  ctxt = new InitialContext();
  ds = (DataSource)ctxt.lookup("jdbc/PostgreSQLDS");
  conn = ds.getConnection();
  stmt = conn.createStatement();
  rs = stmt.executeQuery("SELECT * FROM users");
} catch (NamingException e) {
  System.err.print("Cannot find named datasource");
} catch (SQLException se) {
  System.err.print("Error occurred while performing query");
}
```

4.4. java.sql.ResultSet

Um objeto *ResultSet* é o resultado de uma consulta em um banco de dados. Os dados dentro de um objeto *ResultSet* podem ser melhor visualizados com uma tabela. As informações podem ser recuperada uma coluna de cada vez, com o objeto *ResultSet* que mantem-se apontando para determinado registro.

Para percorrer cada linha no *ResultSet*, chamamos o método *next()*. Este método move-se a um ponto interno do objeto *ResultSet* para a próxima linha. Retorna **true** se a próxima linha for encontrada e *false* ao encontrar o final de arquivo (*EOF*).

```
while (rs.next()) {
```

```
    // ler cada coluna
  }
```

Para recuperar os campos em cada linha, o objeto *ResultSet* fornece um certo de número de métodos de recuperação. Existe um método *getString* para recuperar dados de uma String, um método *getInt* para recuperar dados do tipo inteiro, *getBoolean* para recuperar dados tipo boolean, e assim sucessivamente. Em todos os casos esses métodos recebem um parâmetro, como o número da coluna da coluna que contém o dado ou o nome da coluna. Recomenda-se, entretanto, que nomes sejam usados para especificar a coluna para ser lida em vez do número de linhas. Isto faz com que a aplicação seja mais fácil de dar manutenção, pois é possível que a ordem da coluna seja alterada ao longo do tempo após o início do desenvolvimento.

```
Context ctxt = null;
DataSource ds = null;
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
  ctxt = new InitialContext();
  ds = (DataSource)ctxt.lookup("jdbc/PostgreSQLDS");
  conn = ds.getConnection();
  stmt = conn.createStatement();
  rs = stmt.executeQuery("SELECT * FROM users");
  while (rs.next()) {
    String userName = rs.getString("name");
    String address = rs.getString("address");
    int userID = rs.getInt("userid");
    // Outras operações com os dados retornados
  }
} catch (NamingException e) {
  System.err("Cannot find named datasource");
} catch (SQLException se) {
  System.err("Error occurred while performing query");
}
```

4.5. Liberando recursos do sistema

Uma etapa muito importante, que é negligenciada freqüentemente, é a de liberar os recursos de banco de dados depois de uma operação ter sido completada. Isto deve ser feito explicitamente e é de responsabilidade do programador. Sem executar tal liberação, os recursos mencionados pela operação acima NÃO podem ser usados futuramente. Para aplicações de larga escala, isto pode rapidamente resultar em perda de disponibilidade de conexões.

A liberação de recursos pode ser feita pela chamada do método *close()* em cada um dos objetos *Connection*, *Statement* e *ResultSet*. Há uma ordem específica envolvida: o objeto do tipo *ResultSet* deve ser fechado primeiro, em seguida o objeto do tipo *Statement* e, finalmente o objeto do tipo *Connection*. Já que cada método de fechamento de cada objeto foi definido para lançar uma *SQLException*, deve-se envolver as instruções em um bloco *try-catch*.

Um erro comum que os desenvolvedores fazem é simplesmente colocar métodos de fechamento dentro do corpo do bloco *try-catch*. Aqui está um exemplo:

```
Context ctxt = null;
DataSource ds = null;
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
  ctxt = new InitialContext();
  ds = (DataSource)ctxt.lookup("jdbc/PostgreSQLDS");
  conn = ds.getConnection();
  stmt = conn.createStatement();
  rs = stmt.executeQuery("SELECT * FROM users");
  while (rs.next()) {
    String userName = rs.getString("name");
```

```
      String address = rs.getString("address");
      int userID = rs.getInt("userid");
      // Outras operações com os dados retornados
    }
    rs.close();
    stmt.close();
    conn.close();
  } catch (NamingException e) {
    System.err("Cannot find named datasource");
  } catch (SQLException se) {
    System.err("Error occurred while performing query");
  }
```

O problema com esta abordagem é que condiciona o sucesso um único endereço. Nos casos onde uma exceção ocorre no código os recursos de sistema NÃO serão liberados corretamente. Um caminho melhor será colocar a clausula *finally*, para assegurar que que os recursos sejam liberados não importando o que aconteça.

A solução para o problema é apresentada a seguir:

```
Context ctxt = null;
DataSource ds = null;
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
  ctxt = new InitialContext();
  ds = (DataSource)ctxt.lookup("jdbc/PostgreSQLDS");
  conn = ds.getConnection();
  stmt = conn.createStatement();
  rs = stmt.executeQuery("SELECT * FROM users");
  while (rs.next()) {
    String userName = rs.getString("name");
    String address = rs.getString("address");
    int userID = rs.getInt("userid");
    // Outras operações com os dados retornados
  }
} catch (NamingException e) {
  System.err("Cannot find named datasource");
} catch (SQLException se) {
  System.err("Error occurred while performing query");
} finally {
  try {
    if (rs != null) rs.close();
  } catch (SQLException e) {}
  try {
    if (stmt != null) stmt.close();
  } catch (SQLException e) {}
  try {
    if (conn != null) conn.close();
  } catch (SQLException e) {}
}
```

Os questionamentos sobre se o objeto é nulo são necessários apenas nos casos em que a condição de erro ocorra antes de um ou mais objetos terem sido apropriadamente instanciados. Cada método *close* deve também ser separado numa clausula *try-catch* para assegurar que um erro causado na tentativa de fechamento de um objeto não atrapalhe o fechamento dos outros objetos.

5. Exercício

5.1. Usuários

Considere a seguinte tabela:

USERS

userid	gender	firstname	lastname	login	password
14627895	M	Jose	Saraza	jsaraza	Asdfrewq167
65248987	M	Rosario	Antonio	rantonio	qwer4679
52317568	F	Milagros	Paguntalan	mpaguntalan	ukellllll3
72324489	M	Frank	Masipiquena	fmasipiquena	Df23efzsxf2341

1. Executar as instruções SQL necessárias para:

a) Recuperar todos os usuários masculinos.
b) Recuperar todos os usuários com o primeiro nome iniciando com a letra F.
c) Alterar o nome de login do registro com um *userid* de 65248987 para **rtonio**.
d) Eliminar todos os registros femininos.
e) Inserir o seguinte registro na tabela:

userid	gender	firstname	lastname	login	password
69257824	F	Anne	Sacramento	asacramento	k1lasdoj24f

2. Criar uma classe *LoginHandler*. Deverá conter a seguinte assinatura:

```
public boolean isUserAuthorized(String loginName, String password)
```

Dentro do corpo do método, crie e implemente uma conexão para o banco de dados exemplo, e verifique novamente a tabela usuário se existe um registro que tenha o mesmo *login* e *password* que são dados nos parâmetros. Use a classe *DriverManager* para obter a conexão com o banco de dados.

3. Criar um servlet de nome *UserEntryServlet* que forneça o seguinte formulário:

```
<HTML>
 <BODY>
  <table>
   <form action="UserEntryServlet" method="post">
    <tr>
     <td>User ID:</td>
     <td><input type="text" name="userID"/></td>
    </tr><tr>
     <td>First name</td>
     <td><input type="text" name="firstName"/></td>
    </tr><tr>
     <td>Last name</td>
     <td><input type="text" name="lastName"/></td>
    </tr><tr>
     <td>Login name</td>
     <td><input type="text" name="loginName"/></td>
    </tr><tr>
     <td>Password</td>
     <td><input type="text" name="userID"/></td>
    </tr>
   </table>
  </form>
 </BODY>
</HTML>
```

Usar os valores informados no formulário e criar um novo registro na tabela de usuários no banco de dados exemplo. Para conectar à base de dados, configure o Application Server para a fonte de dados.

4. Criar um servlet dando o nome de *UserRemovalServlet* esperando um parâmetro "userID". Eliminar o registro que corresponda a esse registro.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.