Unidade 9: Middleware JDBC para Criação de Beans

Implementando MVC Nível 1

Prof. Daniel Caetano

Objetivo: Preparar o aluno para construir classes de entidade com capacidade de persistência.

Bibliografia: QIAN, 2007; DEITEL, 2005.

INTRODUÇÃO

Até o momento criamos diversos Servlets, mas sempre trabalhando com tipos nativos ou tipos não nativos pré-existentes no Java. Se quisermos criar uma aplicação mais útil, entretanto, seremos "obrigados" a criar nossos próprios tipos não nativos.

Vimos como criar tipos não nativos logo no início do curso, mas aqueles tipos não nativos tinham uma deficiência do ponto de vista de um sistema prático real: de que adiante criar um objeto de livro ou cliente se, ao desligar o computador, todos os dados estarão perdidos?

Obviamente nenhum sistema real funciona que seja executado em um PC desta forma; sendo assim, iremos verificar nessa aula como implementar uma classe de entidade capaz de **persistir**, isto é, de fazer com que seus dados "permaneçam" mesmo que o equipamento seja desligado e, para isso, usaremos o middleware JDBC.

1. CRIANDO A WEB APPLICATION

A nossa aplicação vai conter vários elementos, incluindo servlets e JSPs. Para criá-la, façamos o seguinte:

- Clicar em Criar Projeto
- Selecionar "Java Web" e "Aplicação Web"
- Clicar em Próximo.
- Dar nome ao projeto "WProjeto5"
- Clicar em Próximo.
- Selecione o uso do GlassFish, dependendo da sua instalação
- Clicar em Finalizar.

2. CRIANDO O BEAN CLIENTE

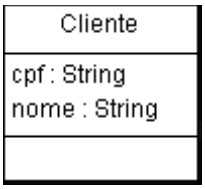
Agora que a aplicação está configurada, vamos criar a nossa classe de entidade, o nosso **bean** Cliente.

- Clique com botão direito em "Pacotes de Código Fonte"
- Selecionar **Novo > Pacote Java**
- Criar pacote com nome **entidades**, clicando depois em Finalizar.

O objeto de entidade Cliente é aquele que armazenará todos os dados de nosso cliente. Para construí-lo, comecemos assim:

PASSO 1: Clique com o botão direito no **pacote entidade**s e selecione **Novo > Classe Java** e dê o nome de **Cliente** a ela. Isso criará um novo arquivo de classe chamado **Cliente.java**, que estará automaticamente aberta no editor.

PASSO 2: Iremos agora configurar a classe para que tenha 2 atributos: **cpf** e **nome**, conforme indicado no diagrama abaixo:



Todos estes atributos serão privados. Assim, devemos inserir as seguintes linhas na classe Cliente:

Cliente.java

```
package wsiscli;

public class Cliente {

    // Atributos Privados
    private String cpf;
    private String nome;

}
```

PASSO 3: Como se trata de um objeto de entidade, teremos *getters* e *setters* para todos os atributos (cpf e nome). Vamos criá-los usando os recursos do NetBeans para acelerar o trabalho. Clique com o botão direito no código e selecione a opção **Inserir Código** que aparece no menu e, em seguida, selecione **Getter e setter**, o que irá abrir uma janela. Selecione **todos os atributos** da classe e clique em **Gerar**.

PASSO 4: Para que possamos testar, vamos criar um método **toString** que imprima nosso objeto na forma **nome (cpf)** . Para isso, clique com o botão direito no código e selecione a opção **Inserir Código** que aparece no menu e, em seguida, selecione **toString**, o

que irá abrir uma janela. Selecione **cpf** e **nome** e clique em **Gerar**. Isso irá criar um código que você deve modificar para que fique como indicado abaixo.

Cliente.java (método toString)

```
    @Override
    public String toString() {
        return getNome() + " (" + getCpf() + ")";
    }
```

PASSO 5: Vamos agora testar o que foi feito. Vá ao arquivo index.jps e modifique o conteúdo do corpo da seguinte forma:

```
<%--
    Document   : index
    Created on : 23/08/2011, 10:38:32
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8" import="entidades.*" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Teste</title>
    </head>
    <body>

            <%
                  Cliente cli = new Cliente();
                  cli.setNome("José da Silva");
                  cli.setCpf("01234567890");
                  out.println("<p>" + cli + "</p>");
            %>

    </body>
</html>
```

PASSO 6: Execute o programa e veja se, agora, o objeto é impresso corretamente.

PASSO 7: Uma das funções importantes de um objeto de entidade é que seus **getters** e **setters** realizem alguma validação. Nos getters que respondem objetos (como getCpf e getNome, que devolvem objetos **String**) é preciso definir que, caso estes objetos não existam (valor "null"), um texto vazio deve ser devolvido. Para isso, modifique os métodos getNome e getCpf, conforme indicado a seguir.

Cliente.java (método getCpf)

```
    public String getCpf() {
        if (cpf == null) return "";
        return cpf;
```

Cliente.java (método getNome)

```
    public String getNome() {
        if (nome == null) return "";
        return nome;
```

PASSO 8: Agora falta definir a validação dos **setters**. Os setters são métodos usados para **atualizar** os valores dos atributos. Essa atualização **não deve** ser realizada caso os novos valores sejam inválidos. Assim, precisamos **validar** os dados de nome e cpf, iniciando pelo nome. Como o nome é um objeto String, primeiramente vamos verificar se o valor do novo nome não é **null**, que obviamente será rejeitado. Adicionalmente, vamos verificar se o tamanho do novo nome é menor que 5 caracteres e, nesse caso, também vamos rejeitá-lo. Assim, devemos modificar o método setNome da seguinte forma:

Cliente.java (método setNome)

```
    public boolean setNome(String nome) {
        // Se nome muito curto, vai embora com false.
        if (nome == null || nome.length() < 5) return false;
        this.nome = nome;
        return true;
    }
```

Observe que o retorno do método foi **modificado** para **boolean**, para que seja possível verificar, por quem chamou o método, se o resultado da alteração teve ou não sucesso. Observe, também, que é muito importante verificar se o nome é **null** antes de executar o método **length**. A razão é simples: **null nunca poderá executar o método length** e tentar fazê-lo irá causar erro na execução do programa.

PASSO 9: Por último, algo um pouco mais complicado: validar o CPF. Não faremos uma validação completa, pois excluiremos a validação do dígito de verificação. Será seguido o seguinte procedimento: primeiro verificaremos se o novo CPF não é null; se for, será rejeitado. Depois, limparemos espaços e caracteres especiais e verificaremos se o comprimento é 11 caracteres; se não for, será rejeitado. Finalmente, verificaremos se cada um dos caracteres é um dígito (numérico); se algum deles não o for, rejeitaremos o CPF. O código que faz isso é apresentado a seguir.

Cliente.java (método setCpf)

```
    public boolean setCpf(String cpf) {
        // Se nenhum CPF fornecido, vai embora com erro.
        if (cpf == null) return false;

        // Limpa espaços, pontos e traços
        cpf = cpf.trim();
        cpf = cpf.replaceAll(" ","");
        cpf = cpf.replaceAll("[.-]","");

        // Pega o comprimento do cpf já limpo.
        int cpflen = cpf.length();
        // Se não tiver exatos 11 dígitos, rejeita.
        if (cpflen != 11) return false;
```

```
        // Precisa ser composto apenas por números
        for (int i=0; i<cpflen; i++ ) {
            // Se algum dos caracteres não for um dígito numérico,
            // vai embora com erro.
            if (Character.isDigit(cpf.charAt(i)) == false) return false;
        }
        // No caso real, é necessário testar o dígito de verificação!
        // Se chegou aqui, todas as validações foram feitas com sucesso!
        this.cpf = cpf;
        return true;
    }
```

Nas futuras disciplinas de Java SE serão apresentados mais detalhes sobre a criação de classes de entidade ainda mais completas. Por hora, trabalharemos com esta.

3. IMPLEMENTANDO A PERSISTÊNCIA

Antes de mais nada, precisamos de um banco de dados. Na aba de "serviços" do NetBeans, inicie o servidor **Java DB** e crie o banco de dados **sisclientes**, usando como usuário o nome **sisclientes** e como senha também **sisclientes**. Mude o nome de visualização para **SisClientes DB**.

Conecte ao banco de dados **sisclientes** e defina **APP** como o esquema padrão. Agora, na parte **APP > TABELAS**, crie uma tabela chamada **cliente** com os campos:

Nome	Tipo	Tamanho	Chave Primária	NULL
cpf	CHAR	11	Sim	Não
nome	VARCHAR	150	Não	Sim

Agora, vamos aproveitar o conhecimento das aulas anteriores e vamos acrescentar, em nossa classe Cliente, o seguinte método chamado **persist**:

Cliente.java (persist)

```
    /**
     * Adiciona/Atualiza um cliente no banco de dados.
     * @return true se cliente foi armazenado/atualizado com sucesso.
     */
    public boolean persist() {
        try {
            // *** CONECTA AO BANCO
            // Linka com driver
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            // Conecta ao banco
            Connection con = DriverManager.getConnection(
                   "jdbc:derby://localhost:1527/sisclientes",
                   "sisclientes", "sisclientes");
            if (con == null) throw new SQLException();
            // Cria a transação
            Statement trans = con.createStatement();
```

```
            //*** Executa UPDATE!
             String query  = "UPDATE app.cliente SET ";
             query += "nome = '" + getNome() + "'";
             query += " WHERE ";
             query += "cpf = '" + getCpf() + "'";
             int linhas = transacao.executeUpdate(query);

             // Se NÃO foi possível inserir, tenta atualizar!
             if (linhas == 0) {
                  //*** Executa INSERT
                   String query  = "INSERT INTO app.cliente VALUES(";
                   query += "'" + getCpf() + "'";
                   query += ", ";
                   query += "'" + getNome() + "'";
                   query += ")";
                   linhas = transacao.executeUpdate(query);
             }

             //*** Finaliza transação e conexão
             transacao.close();
             con.close();

             // Se foi possível alterar o banco de dados... Retorna ok.
             if (linhas != 0) return true;
             }
        // se houve algum erro nas transações de SQL...
        catch (SQLException ex) {
             System.err.println(ex); // Código apenas para debug
             }
        // Se não encontrou o driver
        catch (ClassNotFoundException ex) {
             System.err.println(ex); // Código apenas para debug
             }

        // Por padrão, retorna erro no fim.
        return false;
    }
```

4. USANDO O BEAN

PASSO 1: Modique o **index.jsp** da seguinte forma:

```
<%--
    Document   : index
    Created on : 23/08/2011, 10:38:32
    Author     : djcaetano
--%>

<%@page contentType="text/html" pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
    <head>
         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
         <title>Teste</title>
    </head>
    <body>
            <h1><a href="NovoCliente">Novo Cliente</a></h1>

    </body>
</html>
```

PASSO 2: Crie um pacote java chamado **sisclientes** e, dentro dele, crie um servlet de nome **NovoCliente**, lembrando de adicionar as informações do descritor XML.

PASSO 3: Nesse servlet, modifique o método processRequest da seguinte forma:

NovoCliente.java (processRequest)

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        try {
            Cliente cli = new Cliente();
            cli.setNome("José da Silva");
            cli.setCpf("01234567890");

            request.setAttribute("cliente", cli);
            RequestDispatcher rd;
            rd = request.getRequestDispatcher("CienteView.jsp");
            rd.forward(request.response);
            return;

        } finally {
        }
    }
```

PASSO 4: Crie, agora, na pasta de arquivos Web, o JSP **ClienteView.jsp**:

ClienteView.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8" import="entidades.*" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Mostra Cliente</title>
    </head>
    <body>

            <%
                  Cliente cli = (Cliente)request.getAttribute("cliente");
                  out.println("<p>" + cli + "</p>");
            %>

    </body>
</html>
```

PASSO 5: Experimente!

5. IMPLEMENTANDO A RESTAURAÇÃO

A restauração é o processo inverso do armazenamento, isto é, é relativo à recuperação de dados previamente armazenados. A restauração será feita com a implementação de um método chamado **restore** na classe Cliente, como indicado a seguir.

Cliente.java (restore)

```
    /** Restaura um cliente a partir do banco de dados.
     *  @return true se cliente foi armazenado/atualizado com sucesso.
     */
    public boolean restore(String umCpf) {
        try {
            // *** CONECTA AO BANCO
            // Linka com driver
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            // Conecta ao banco
            Connection con = DriverManager.getConnection(
                   "jdbc:derby://localhost:1527/sisclientes",
                   "sisclientes", "sisclientes");
            if (con == null) throw new SQLException();
            // Cria a transação
            Statement trans = con.createStatement();

            //*** Executa SELECT!
            String query  = "SELECT * FROM app.cliente WHERE ";
            query += "cpf = '" + umCpf + "'";
            ResultSet res = transacao.executeQuery(query);
            if (res.next()) {
                   setCpf( res.getString("cpf") );
                   setNome( res.setString("nome") );
                  transacao.close();
                  con.close();
                  return true;
            }
            else setCpf(umCpf);

            //*** Finaliza transação e conexão
            transacao.close();
            con.close();
           }
        // se houve algum erro nas transações de SQL...
        catch (SQLException ex) {
            System.err.println(ex); // Código apenas para debug
            }
        // Se não encontrou o driver
        catch (ClassNotFoundException ex) {
            System.err.println(ex); // Código apenas para debug
            }
        // Por padrão, retorna erro no fim.
        return false;
    }
```

Para testar, vamos tentar recuperar o cliente no servlet **NovoCliente**:

NovoCliente.java (processRequest)

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        try {
            Cliente cli = new Cliente();
            cli.restore("01234567890");

            request.setAttribute("cliente", cli);
            RequestDispatcher rd;
            rd = request.getRequestDispatcher("CienteView.jsp");
            rd.forward(request.response);
            return;

        } finally {
        }
    }
```

Experimente!

6. RESTAURAÇÃO AUTOMÁTICA

Será que não é possível criar um processo de "restauração automática" quando o objeto é criado?

Na verdade, é. Sempre que quisermos executar um código logo que um objeto é criado, devemos usar o método **construtor**. O método construtor **sempre terá o mesmo nome que a classe**. Por exemplo, na classe cliente, podemos declará-lo assim:

Cliente.java (construtor)

```
public Cliente() {

}
```

Observe que o construtor **não retorna nenhum tipo de dado**. De fato, **é proibido** usar **return** no construtor. Mas dentro desse bloco podemos escrever o que quisermos. Por exemplo:

Cliente.java (construtor)

```
public Cliente() {
    restore();
}
```

Obviamente isso não vai funcionar... afinal, o método **restore** precisa de um parâmetro: **o cpf do cliente a restaurar**!

Ora, foi dito que o construtor não pode retornar nenhum valor... **mas ele pode receber parâmetros**. Modifique-o assim:

Cliente.java (construtor)

```
public Cliente(String cpf) {
    restore(cpf);
}
```

E pronto!

Agora só falta fazer algumas modificações no Servlet **NovoCliente.** Observe no código a seguir.

NovoCliente.java (processRequest)

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        try {
            Cliente cli = new Cliente("01234567890");

            request.setAttribute("cliente", cli);
            RequestDispatcher rd;
            rd = request.getRequestDispatcher("CienteView.jsp");
            rd.forward(request.response);
            return;

        } finally {
        }
    }
```

Experimente!

7. MÚLTIPLOS CONSTRUTORES

O método que usamos anteriormente é perfeitamente legítimo; entretanto, ele sempre nos obriga a uma busca no banco de dados **mesmo que saibamos** que o cliente não existe lá. Como poderíamos evitar aquela busca ao banco?

SIMPLES... criando **outro construtor diferente**.

Nada me obriga a ter um único construtor. **A única exigência** quando se usa múltiplos construtores **é que os parâmetros sejam diferentes**.

Assim, é perfeitamente **correto** o código abaixo:

Cliente.java (construtores)

```
    public Cliente(String cpf) {
        restore(cpf);
    }

    public Cliente() {
    }
```

Com estes dois construtores, quando o objeto for criado assim:

Cliente cli = new Cliente(); // sem parâmetros

O objeto será criado com o construtor que não faz nada. Por outro lado, se o objeto for criado assim:

Cliente cli = new Cliente("01234567890"); // com um parâmetro String

O objeto será criado com o construtor que executa o **restore**.

8. BIBLIOGRAFIA

QIAN, K; ALLEN, R; GAN, M; BROWN, R. **Desenvolvimento Web Java**. Rio de Janeiro: LTC, 2007.

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.