HIBERNATE

Criando um projeto em Java + Hibernate do zero

SUMÁRIO

1 Instalação do NetBeans

Faça o download da **versão completa** do NetBeans 7 ou 8 e instale-o.

https://netbeans.org/downloads/7.1.2/

2 Instalação do Java Development Kit (JDK)

Se você ainda não tiver o Java Development Kit (JDK) instalado em sua máquina, faça o download e instale-o.

http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk7-downloads-1880260.html

3 Criar projeto no NetBeans

3.1 No NetBeans, acesse o menu **Arquivo** > **Novo projeto.**

3.2 Selecione a pasta **Java Web** e, em seguida, **Aplicação Web**.

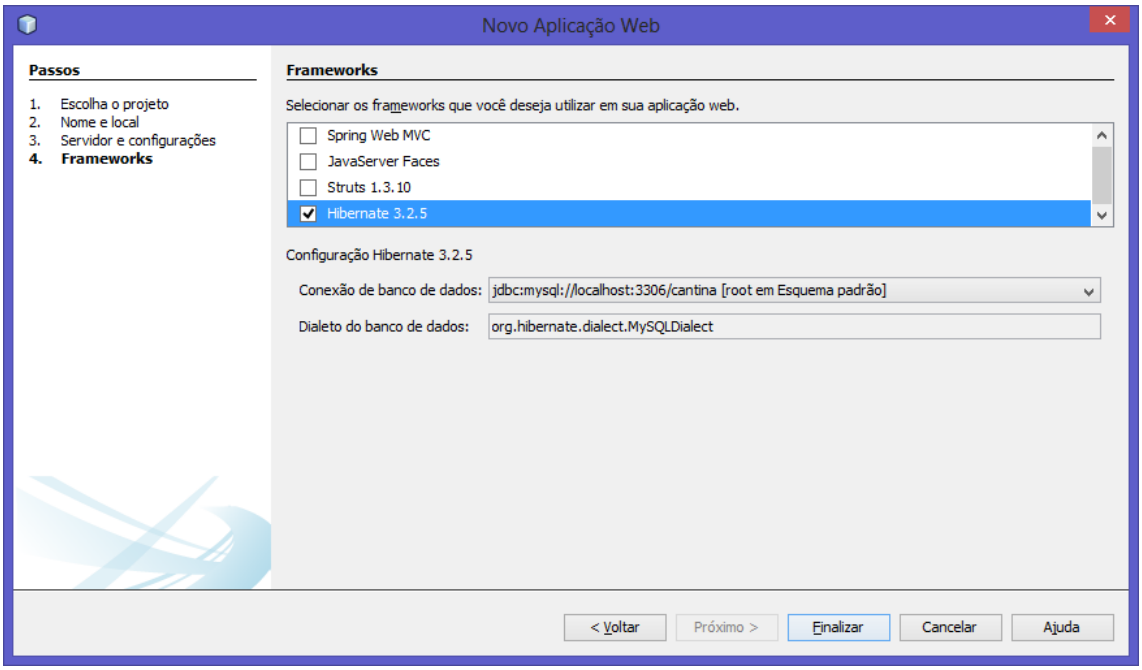
3.3 Clique em **Próximo**.

3.4 Dê o nome **ProjetoCantina** para a aplicação Web.

3.5 Clique em **Próximo**.

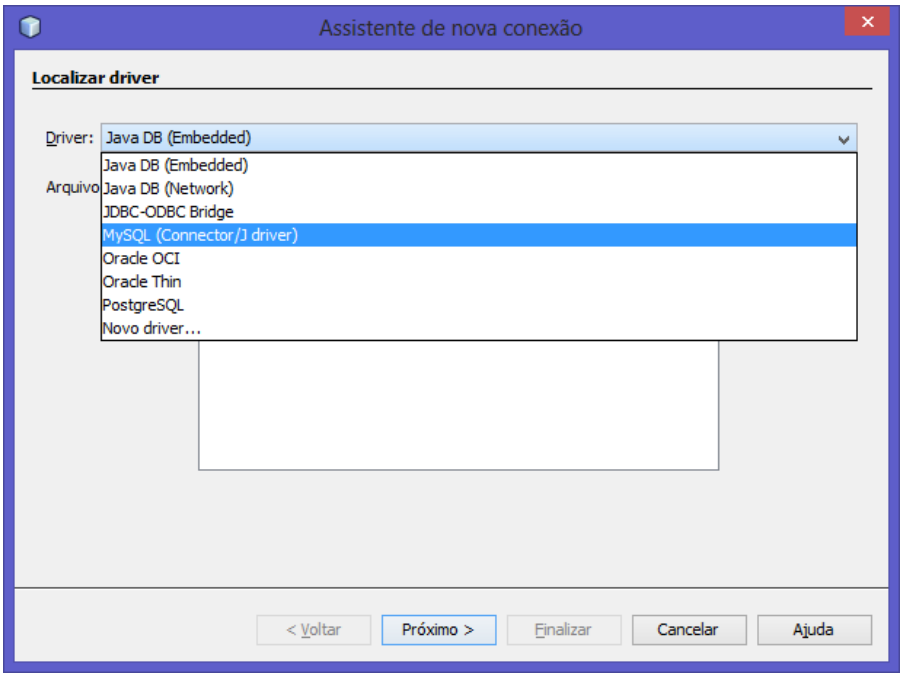
3.6 Na tela que indicará o servidor Web a ser usado, clique em **Próximo**.

3.7 Na tela dos frameworks, selecione o framework de persistência **Hibernate**.



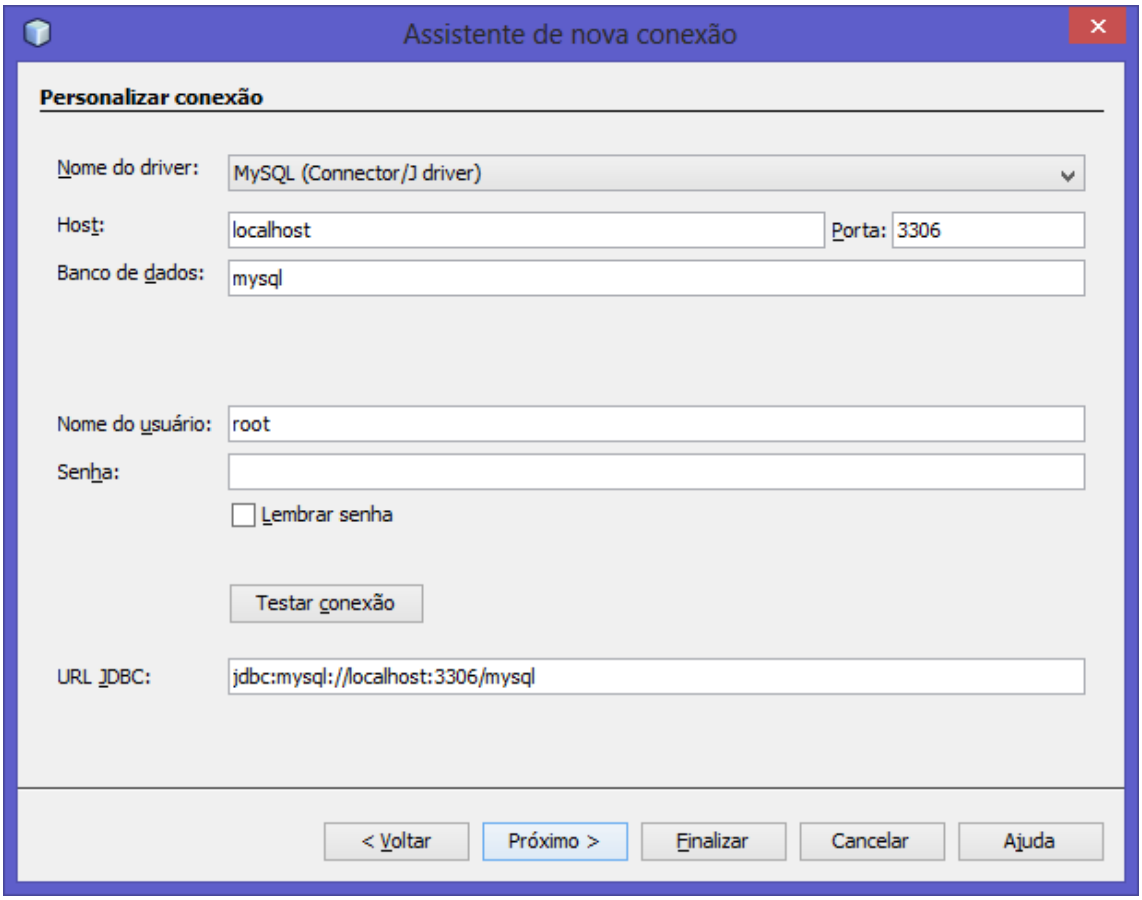
3.8 Para configurar a conexão do Hibernate com um banco de dados MySQL, selecione a opção **Nova conexão com banco de dados...** na lista de **Conexão de banco de dados**.

3.9 Na nova janela que será aberta, selecione o driver **MySQL (Connector/J driver)**.



3.10 Clique em **Próximo**.

3.11 Informe os dados de acesso ao servidor MySQL instalado em sua máquina e clique em **Próximo**. No nome do banco de dados, insira **cantina** e não se esqueça criar esse banco de dados no MySQL.



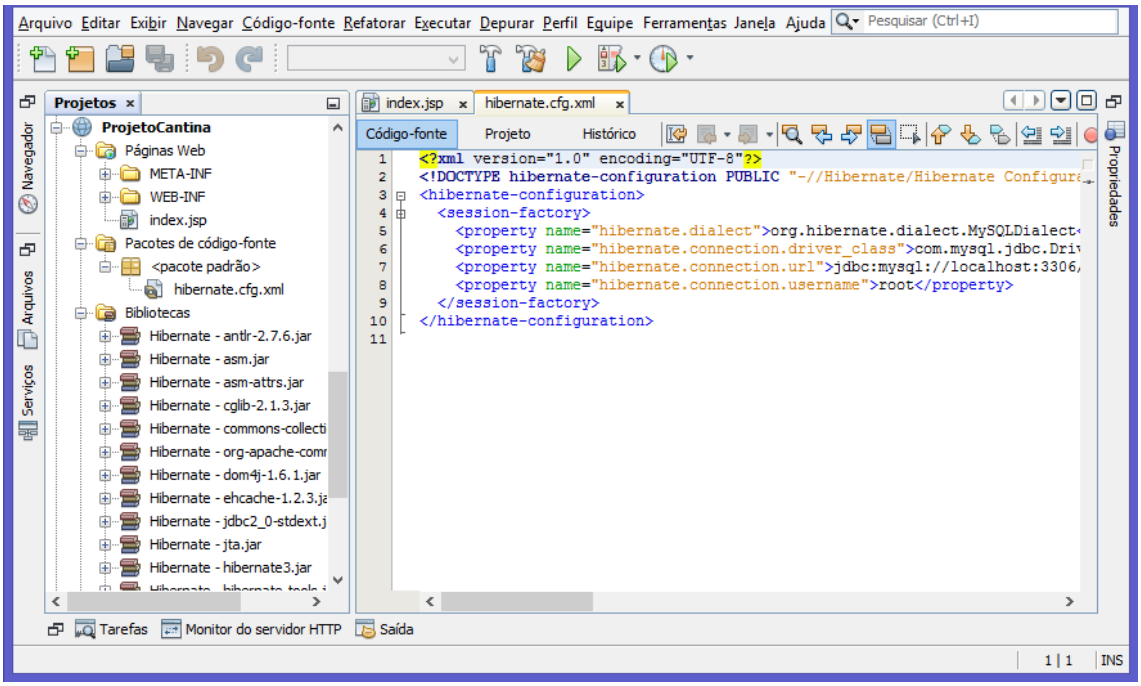
3.12 Na tela seguinte (**Escolher esquema de banco de dados**), clique em **Finalizar**.

3.13 Na tela de trás (**Frameworks**), clique em **Finalizar** para criar o projeto.

4 O arquivo hibernate.cfg.xml

4.1 Seguindo os passos anteriores, o projeto será criado e, consequentemente, será gerado o arquivo de configuração do Hibernate: **hibernate.cfg.xml**. Esse arquivo é responsável por gerenciar todas as configurações do framework Hibernate dentro da aplicação. Portanto, a configuração do banco de dados que será usado é escrita nesse arquivo.

Além disso, é também nesse arquivo que serão indicadas as classes que serão mapeadas para o banco de dados.



4.2 A cada vez que a aplicação é executada, o arquivo **hibernate.cfg.xml** é lido. Ao iniciar a aplicação, é possível definir se o banco de dados será criado/recriado ou atualizado em relação ao mapeamento das classes. Para isso, acrescente a propriedade **hibernate.hbm2dll.auto**, que define o modo como o banco de dados será manipulado pelo Hibernate.

```
<property name="hibernate.hbm2ddl.auto">update</property>
```

O valor **update** indica que o Hibernate não apagará o banco de dados toda vez que a aplicação, mas irá atualizá-lo caso tenha ocorrido alguma alteração no mapeamento das classes da aplicação. O valor **create-drop** informa que o Hibernate deverá apagar todas as tabelas do banco de dados e recriá-las sempre que a aplicação executar.

4.3 Como o Hibernate gerencia os comandos SQL que são executados no banco de dados, tornando-o transparente para a aplicação, é possível visualizar quais comandos SQL estão sendo executados. Para isso, insira a propriedade **show_sql** na configuração do Hibernate.

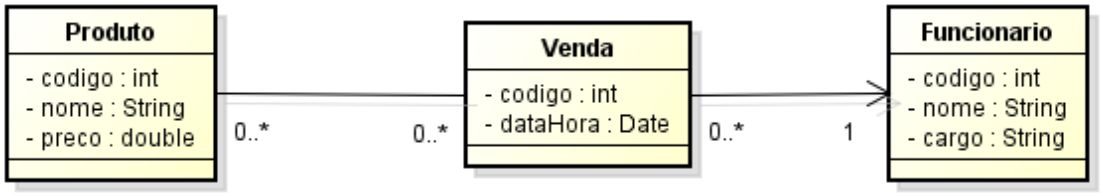
```
<property name="show_sql">true</property>
```

Portanto, o arquivo **hibernate.cfg.xml** tem esta configuração até o momento:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</proper
ty>
    <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/cantina</p
roperty>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">senha</property>

    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

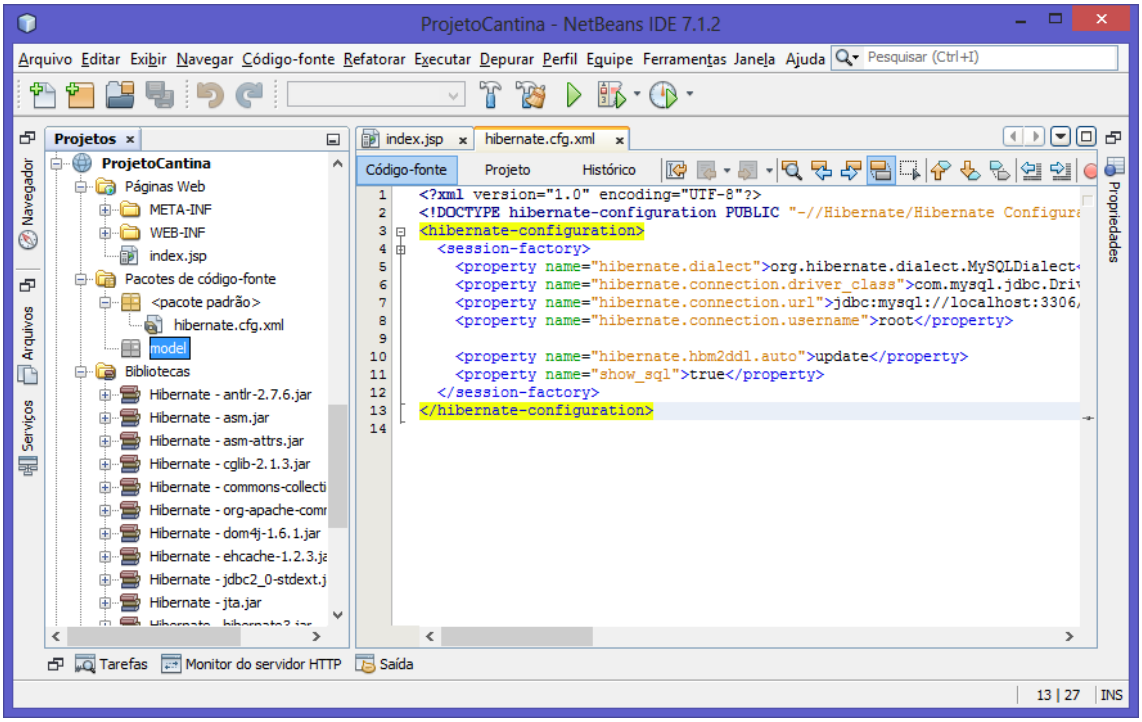
5 Criar as classes da aplicação

Para este projeto, criaremos as classes de acordo com o seguinte diagrama, a fim de exemplificar como manipular objetos em associações do tipo 1-muitos e muitos-para-muitos.



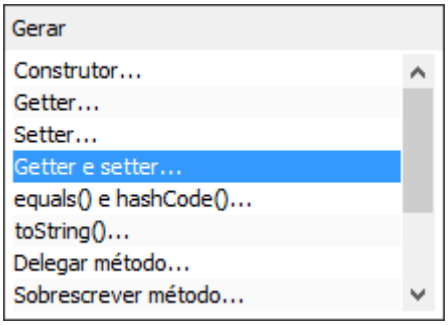
5.1 Antes de criar as classes na aplicação, crie um pacote para inseri-las posteriormente. Para isso, clique com o botão direito em **Pacotes de código-fonte** > **Novo** > **Pacote Java...**

5.2 Insira o nome **model** e o pacote será criado como exibido na seguinte figura.

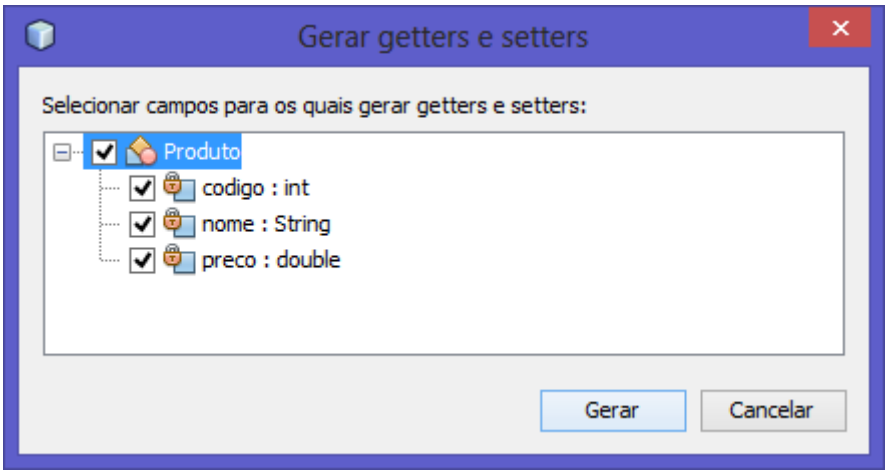


5.3 Crie as classes **Produto**, **Venda** e **Funcionario**. Para criar cada classe, clique com o botão direito em **model** e, em seguida, **Novo** > **Classe Java...**

5.4 Na classe Produto, insira os atributos conforme o diagrama de classes. Após isso, para inserir os métodos **get** e **set**, pressione **ALT+Insert**, selecione **Getter e setter...**



Marque os atributos da classe Produto e clique em **Gerar**.



5.5 Nas classes Venda e Funcionário, insira os atributos conforme o diagrama de classes. Após isso, insira os métodos **get** e **set**.

5.6 Código da classe Produto

```
public class Produto {
    private int codigo;
    private String nome;
    private double preco;

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
```

```
    public double getPreco() {
        return preco;
    }

    public void setPreco(double preco) {
        this.preco = preco;
    }
}
```

5.7 Código da classe Venda

```
import java.util.Date;

public class Venda {

    private int codigo;
    private Date dataHora;

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public Date getDataHora() {
        return dataHora;
    }

    public void setDataHora(Date dataHora) {
        this.dataHora = dataHora;
    }


}
```

5.8 Código da classe Funcionario

```
public class Funcionario {
    private int codigo;
    private String nome;
    private String cargo;

    public String getCargo() {
        return cargo;
    }

    public void setCargo(String cargo) {
        this.cargo = cargo;
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

}
```

5.9 Implemente as associações entre as classes Produto e Venda, Venda e Funcionário. Como a associação entre Produto e Venda é do tipo muitos-para-muitos, haverá uma lista de objetos Venda na classe Produto, e uma lista de objetos Produto na classe Venda. Então, insira o seguinte código na classe Produto. Não esqueça de importar as classes List e LinkedList.

```
private List<Venda> vendas = new LinkedList<Venda>();
```

```
    public List<Venda> getVendas() {
        return vendas;
    }

    public void setVendas(List<Venda> vendas) {
        this.vendas = vendas;
    }
```

Insira o seguinte código na classe Venda. Não esqueça de importar as classes List e LinkedList.

```
private List<Produto> produtos = new LinkedList<Produto>();

    public List<Produto> getProdutos() {
        return produtos;
    }

    public void setProdutos(List<Produto> produtos) {
        this.produtos = produtos;
    }
```

Insira também a associação com a classe Funcionário.

```
    Funcionario funcionario;
    public Funcionario getFuncionario() {
        return funcionario;
    }
    public void setFuncionario(Funcionario funcionario) {
        this.funcionario = funcionario;
    }
```

6 Mapear as classes com Annotations (JPA = Java Persistence API)

O Hibernate é um framework de persistência de dados que realiza o mapeamento do modelo orientado a objeto no modelo lógico relacional (e vice-versa). Por exemplo: se há um atributo **codigo** em uma classe, é necessário informar qual coluna de qual tabela do banco de dados se refere a esse **codigo**. Para realizar esse mapeamento,

podem ser usados arquivos XML para cada classe da aplicação, porém os arquivos XML duplicam o trabalho por serem arquivos externos às classes.
Uma solução mais adequada é o uso de **Annotations** através da JPA (Java Persistence API).

6.1 Mapear a classe Produto

Toda classe que deverá ser persistida no banco de dados deve ser identificada pela Annotation **@Entity**. É possível ainda indicar qual será o nome da tabela do banco de dados correspondente à classe.
Na classe Produto, insira as Annotations destacadas abaixo. Não se esqueça de importar as Annotations do pacote javax.persistence.

```
@Entity
@Table(name="pro_produtos")
public class Produto {
....
}
```

Assim como toda tabela em um banco de dados deve ter uma chave primária, toda classe mapeada no Hibernate deve ter um atributo identificador (ou um conjunto de atributos). Na classe Produto, o código é o atributo identificador, portanto insira acima dele a Annotation **@Id** e indique como os valores desse atributo serão gerados com a Annotation **@GeneratedValue**. Existem quatro tipos de estratégias para gerar valores de um atributo identificador: AUTO (identifica a estratégia automática usada pelo SGBD. No caso do MySQL, autonumeração); IDENTITY (código autonumeração); SEQUENCE (permite usar uma sequence do banco de dados para gerar os valores); TABLE (permite gerar os valores a partir de uma tabela do banco de dados que armazena códigos).

```
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private int codigo;
```

6.2 Mapear a classe Funcionário

Na classe Funcionário, serão usadas as mesmas Annotations que foram inseridas na classe Produto.

```
@Entity
@Table(name="fun_funcionarios")
public class Funcionario {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int codigo;
    ....
}
```

6.3 Mapear a classe Venda

Na classe Venda, serão usadas as mesmas Annotations das classes Produto e Funcionário.

```
@Entity
@Table(name="ven_vendas")
public class Venda {

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private int codigo;
....
}
```

Além dessas Annotations, quando há um atributo do tipo **Date**, é possível informar se os valores armazenados no banco de dados conterão apenas a data, apenas a hora ou a data/hora. Para isso é usada a Annotation **@Temporal**. Na classe Venda, será armazenada a data/hora. Então, insira a seguinte linha de código logo acima do atributo **Date dataHora**.

```
   @Temporal(TemporalType.TIMESTAMP)
    private Date dataHora;
```

6.4 Mapear associações Um-para-muitos

A associação entre as classes Venda e Funcionário é do tipo um-para-muitos, sendo que é uma associação unidirecional em que a classe Venda tem um objeto da classe Funcionário, mas a classe Funcionário não tem objetos da classe Venda. Nesse caso, o mapeamento será feito na classe Venda.

Para as associações entre classes, existem quatro Annotations: **@OneToOne**, **@OneToMany**, **@ManyToOne** e **@ManyToMany**.

Como a classe Venda tem multiplicidade MUITOS na associação e contém um objeto da classe Funcionário, será usada a Annotation **@ManyToOne**. Quando há uma associação do tipo um-para-muitos, será criada uma chave estrangeira na tabela cuja classe possui a multiplicidade MUITOS na associação. Neste caso, a tabela **ven_vendas** receberá uma chave estrangeira que faz referência à tabela **fun_funcionarios**. Com a Annotation **@JoinColumn**, é possível definir o nome do campo chave estrangeira na tabela.

Então, insira as seguintes linhas de código logo acima do atributo **funcionario** da classe Venda.

```
@ManyToOne(cascade= {CascadeType.PERSIST, CascadeType.MERGE })
@JoinColumn(name="id_funcionario")
Funcionario funcionario;
```

A opção **cascade** indica que, quando uma Venda for salva no banco de dados, o funcionário também será salvo (se ainda não existir no banco de dados).

6.5 Mapear associações Muitos-para-muitos

A associação entre as classes Produto e Venda é do tipo muitos-para-muitos. Quando isso ocorre, devido ao modelo lógico relacional do banco de dados uma nova tabela será criada com as chaves primárias de cada tabela envolvida no relacionamento.

Na classe Venda, insira as seguintes linhas de código logo acima do atributo **produtos**.

```
@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
      @JoinTable(name = "vendas_produtos", joinColumns = {
                  @JoinColumn(name = "codigo_venda") },
                  inverseJoinColumns = { @JoinColumn(name =
"codigo_produto") })
private List<Produto> produtos;
```

A Annotation **@ManyToMany** indica que a lista de produtos se refere a uma associação do tipo muitos-para-muitos. O atributo **fetch** indica que, quando for selecionada uma venda, os

produtos serão carregados no modo **lazy (preguiçoso)**. O Hibernate usa esse modo para não trazer todos os dados relacionados, para não onerar a performance. Outro valor do atributo **fetch** pode ser **EAGER**, que fará com que o Hibernate busque todos os produtos e carregue-os na memória. O atributo **cascade** com o tipo **ALL** indica que, quando uma venda for salva no banco de dados ou excluída, os produtos relacionados na tabela N-N serão salvos ou excluídos também. Essa é uma das grandes vantagens de usar o Hibernate, pois facilita o gerenciamento do registro-pai e seus respectivos filhos. A Annotation **@JoinTable** permite informar o nome da tabela N-N que será gerada pela associação muitos-para-muitos. Na Annotation **@JoinColumn**, é informado o nome que a chave estrangeira de Venda receberá na tabela N-N. No atributo **inverseJoinColumns**, é informado o nome da chave estrangeira da outra tabela que está relacionada (tabela **pro_produtos**).

O mesmo mapeamento feito na classe Venda deve ser feito na classe Produto, por ser uma associação bidirecional no diagrama de classes apresentado anteriormente. Portanto, na classe Produto, insira as seguintes linhas de código logo acima do atributo **vendas**.

```
   @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
      @JoinTable(name = "vendas_produtos", joinColumns = {
                  @JoinColumn(name = "codigo_produto") },
                  inverseJoinColumns = { @JoinColumn(name = 
"codigo_venda") })
    private List<Venda> vendas;
```

7 Criar o banco de dados pelo Hibernate

7.1 Uma vez que as classes estão mapeadas com as Annotations, é necessário informar ao Hibernate quais classes da aplicação serão persistidas no banco de dados. Para isso, vá ao arquivo **hibernate.cfg.xml** e insira as linhas de código destacadas abaixo em negrito.

```
.....
    <property name="show_sql">true</property>

    <mapping class="model.Funcionario" />
    <mapping class="model.Produto" />
    <mapping class="model.Venda" />
  </session-factory>
....
```

7.2 Em seguida, para criar o banco de dados pelo Hibernate, é necessário criar um código em Java. Para isso, clique com o botão direito em **<pacote-padrão>** e, em seguida, **Novo > Classe Java...**

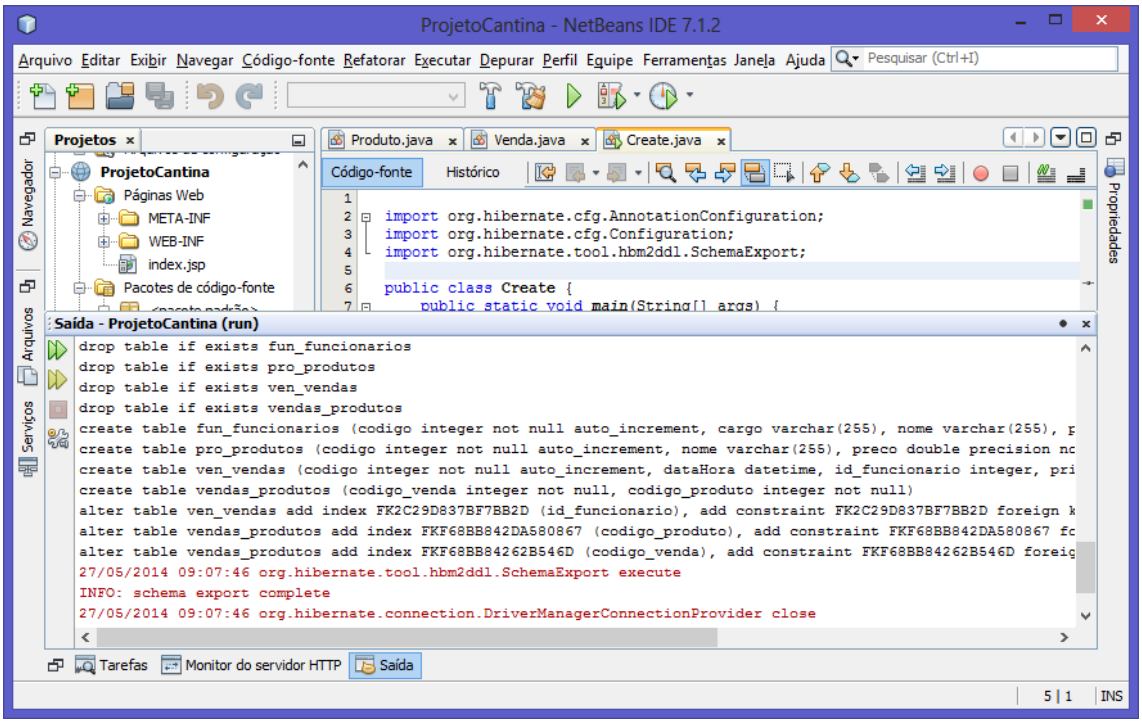
7.3 Nomeie a classe como **Create**.

7.4 Abra o arquivo **Create.java** e insira o seguinte código.

```
import org.hibernate.cfg.Configuration;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class Create {
    public static void main(String[] args) {

        Configuration conf = new AnnotationConfiguration();
        conf.configure();    //este método é responsável por ler o
arquivo hibernate.cfg.xml
        SchemaExport se = new SchemaExport(conf);
        se.create(true, true); //este método imprime o script de
criação do banco de dados no console e exporta o banco de dados de
acordo com as configurações do arquivo hibernate.cfg.xml

    }
}
```

7.5 Execute o arquivo **Create.java** pressionando **SHIFT+F6** ou através do menu **Executar > Executar arquivo.**

As tabelas serão criadas no banco de dados e o script será exibido no console.



Manipulando objetos com Hibernate

Para a manipulação de objetos com Hibernate, são necessários três passos:

1) Ler o arquivo de configuração hibernate.cfg.xml;

2) Criar uma fábrica de sessões (SessionFactory);

3) Criar uma sessão (Session).

Os itens seguintes contêm exemplos de códigos que manipulam objetos no banco de dados. Para cada exemplo, crie uma classe Java com o método main(), insira o código do exemplo e execute o arquivo.

8 Adicionar um objeto no banco de dados

```
import model.Produto;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class AdicionarObjeto {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();
```

```
        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();

        Transaction trans = session.beginTransaction();
        //cria um novo objeto da classe Produto
        Produto produto = new Produto();
        produto.setNome("Coca-cola");
        produto.setPreco(6.0);

        session.save(produto); /*salva o objeto na sessão do Hibernate
        se for usado o comando saveOrUpdate(), o Hibernate irá inserir
no banco de dados caso seja um novo objeto (new Produto())
        ou alterar se for um objeto já existente*/

        trans.commit(); //confirma a transação salvando o objeto no
banco de dados

        session.close();
    }
}
```

9 Alterar um objeto no banco de dados

Verifique se existe no banco de dados um produto com código igual a 1.

```
import model.Produto;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class AlterarObjeto {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();

        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();
```

```
        Transaction trans = session.beginTransaction();

        /* O método load carrega o objeto do banco de dados a partir
do id
         */
        Produto produto = (Produto) session.load(Produto.class, 1); /*
Carrega o produto que tem código igual a 1*/

        produto.setNome("Coca-cola Grande"); //altera o nome do
produto
        produto.setPreco(6.0);

        session.saveOrUpdate(produto); /*neste caso, o objeto será
atualizado e não inserido, porque foi carregado a partir do banco de
dados*/

        trans.commit(); /*confirma a transação salvando o objeto no
banco de dados*/

        session.close();
    }
}
```

10 Excluir um objeto do banco de dados

Verifique se existe no banco de dados um produto com código igual a 1.

```
import model.Produto;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class ExcluirObjeto {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();

        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();
```

```
        Transaction trans = session.beginTransaction();

        /* O método load carrega o objeto do banco de dados a partir
do id
         */
        Produto produto = (Produto) session.load(Produto.class, 1); /*
Carrega o produto que tem código igual a 1*/

        session.delete(produto); //exclui o objeto da sessão

        trans.commit(); //confirma a transação excluindo o objeto do
banco de dados

        session.close();
    }
}
```

11 Adicionar um objeto no banco de dados com associação um-para-muitos

```
import java.util.Date;
import model.Funcionario;
import model.Produto;
import model.Venda;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class AdicionarObjetoUmParaMuitos {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();

        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();

        Transaction trans = session.beginTransaction();

        /*Cria um objeto da classe Funcionário*/
        Funcionario funcionario = new Funcionario();
```

```
        funcionario.setNome("Jão");
        funcionario.setCargo("Balconista");

        /*Cria um objeto da classe Venda*/
        Venda venda=new Venda();
        venda.setDataHora(new Date());

        /*Associa o objeto da classe Funcionário ao objeto da classe
Venda*/
        venda.setFuncionario(funcionario);

        /*Salva o objeto da classe Venda e, devido à cascata
configurada na classe Venda, salva o objeto da classe Funcionário*/
        session.merge(venda);

        trans.commit(); //confirma a transação salvando os objetos no
banco de dados

        session.close();
    }
}
```

12 Alterar um objeto no banco de dados com associação um-para-muitos

Verifique se existe no banco de dados uma venda com código igual a 1.

```
import java.util.Date;
import model.Funcionario;
import model.Produto;
import model.Venda;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class AlterarObjetoUmParaMuitos {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();
```

```
        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();

        Transaction trans = session.beginTransaction();

        /*Busca no banco de dados um objeto da classe Venda que tenha
o id=1 */
        Venda venda = (Venda) session.load(Venda.class,1);

        /*Cria um objeto da classe Funcionário*/
        Funcionario funcionario = new Funcionario();
        funcionario.setNome("José Junqueira");
        funcionario.setCargo("Balconista");

        /*Altera a data/hora da venda*/
        venda.setDataHora(new Date());

        /*Altera o funcionário da venda*/
        venda.setFuncionario(funcionario);

        /*Salva o objeto da classe Venda e, devido à cascata
configurada na classe Venda, salva o objeto da classe Funcionário*/
        session.merge(venda);

        trans.commit(); //confirma a transação salvando os objetos no
banco de dados

        session.close();
    }
}
```

13 Excluir um objeto no banco de dados com associação um-para-muitos

Verifique se existe no banco de dados uma venda com código igual a 1. A exclusão em cascata ocorrerá somente se o tipo de cascata na classe Venda em relação aos produtos for do tipo **ALL**. Se quiser testar, altere o código na classe Venda para isso.

```
import model.Venda;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
```

```
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class ExcluirObjetoUmParaMuitos {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();

        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();

        Transaction trans = session.beginTransaction();

        /*Busca no banco de dados um objeto da classe Venda que tenha
o id=1 */
        Venda venda = (Venda) session.load(Venda.class,1);

        /*Exclui o objeto da classe Venda. Se for definido o tipo ALL
na cascata entre Venda e Funcionários,
         o objeto da classe Funcionário que estiver associado à venda
também será excluído (caso não possua outras vendas vinculadas).
         Isso não é o ideal, por isso foram configurados os tipos de
cascata PERSIST e MERGE.*/
        session.delete(venda);

        trans.commit(); //confirma a transação excluindo o objeto do
banco de dados

        session.close();
    }
}
```

14 Adicionar um objeto no banco de dados com associação muitos-para-muitos

```
import java.util.Date;
import model.Funcionario;
import model.Produto;
import model.Venda;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
```

```
import org.hibernate.cfg.Configuration;

public class AdicionarObjetoMuitosParaMuitos {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();

        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();

        Transaction trans = session.beginTransaction();

        /*Cria um objeto da classe Produto*/
        Produto produto1 = new Produto();
        produto1.setNome("Coca-cola");
        produto1.setPreco(6.0);

        /*Cria um objeto da classe Produto*/
        Produto produto2 = new Produto();
        produto2.setNome("Pizza Mussarela");
        produto2.setPreco(20.0);

        /*Cria um objeto da classe Venda*/
        Venda venda=new Venda();
        venda.setDataHora(new Date());

        /*Adiciona 2 produtos na venda*/
        venda.getProdutos().add(produto1);
        venda.getProdutos().add(produto2);

        /*Salva o objeto da classe Venda e, devido à cascata
configurada na classe Venda, salva os produtos associados*/
        session.merge(venda);

        trans.commit(); //confirma a transação salvando os objetos no
banco de dados

        session.close();
    }
}
```

15 Alterar um objeto no banco de dados com associação muitos-para-muitos

Verifique se existe uma venda com código igual a 1 no banco de dados.

```
import java.util.Date;
import model.Funcionario;
import model.Produto;
import model.Venda;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class AlterarObjetoMuitosParaMuitos {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();

        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();

        Transaction trans = session.beginTransaction();

        /*Busca no banco de dados uma venda que tenha código igual a
1*/
        Venda venda = (Venda) session.load(Venda.class, 6);

        /*Cria um objeto da classe Produto*/
        Produto produto1 = new Produto();
        produto1.setNome("Guaraná");
        produto1.setPreco(6.0);

        /*Altera a data/hora da venda*/
        venda.setDataHora(new Date());

        /*Exclui um produto da venda*/
        venda.getProdutos().remove(0);

        /*Adiciona outro produto na venda*/
        venda.getProdutos().add(produto1);
```

```
        /*Salva o objeto da classe Venda e, devido à cascata
configurada na classe Venda, salva os produtos associados e exclui da
venda o produto que foi removido*/
        session.merge(venda);

        trans.commit(); //confirma a transação salvando os objetos no
banco de dados

        session.close();
    }
}
```

Os comandos SQL executados no banco de dados são:

```
Hibernate: insert into pro_produtos (nome, preco) values (?, ?)
Hibernate: update ven_vendas set dataHora=?, id_funcionario=? where codigo=?
Hibernate: delete from vendas_produtos where codigo_venda=?
Hibernate: insert into vendas_produtos (codigo_venda, codigo_produto) values
(?, ?)
Hibernate: insert into vendas_produtos (codigo_venda, codigo_produto) values
(?, ?)
```

16 Excluir um objeto no banco de dados com associação muitos-para-muitos

Verifique se existe uma venda com código igual a 1 no banco de dados.

```
import java.util.Date;
import model.Funcionario;
import model.Produto;
import model.Venda;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class ExcluirObjetoMuitosParaMuitos {
    public static void main(String[] args) {
        Configuration conf = new AnnotationConfiguration();
        conf.configure();

        SessionFactory factory = conf.buildSessionFactory();
```

```
        Session session = factory.openSession();

        Transaction trans = session.beginTransaction();

        /*Busca no banco de dados uma venda que tenha código igual a
1*/
        Venda venda = (Venda) session.load(Venda.class, 1);

        /*Exclui o objeto da classe Venda e os produtos associados*/
        session.delete(venda);

        trans.commit(); //confirma a transação

        session.close();
    }
}
```

Atenção:

1) Com o tipo CascadeType.ALL na lista de produtos dentro da classe Venda, serão excluídos também os registros da tabela produtos além dos registros da tabela vendas_produtos.

```
Hibernate: delete from vendas_produtos where codigo_venda=?
Hibernate: delete from vendas_produtos where codigo_produto=?
Hibernate: delete from vendas_produtos where codigo_produto=?
Hibernate: delete from pro_produtos where codigo=?
Hibernate: delete from pro_produtos where codigo=?
Hibernate: delete from ven_vendas where codigo=?
```

2) Com o tipo CascadeType.MERGE, serão excluídos somente os registros de vendas_produtos (que é o ideal nessa situação).

```
Hibernate: delete from vendas_produtos where codigo_venda=?
Hibernate: delete from ven_vendas where codigo=?
```