

# Construção de um CRUD Básico com JSF e Hibernate

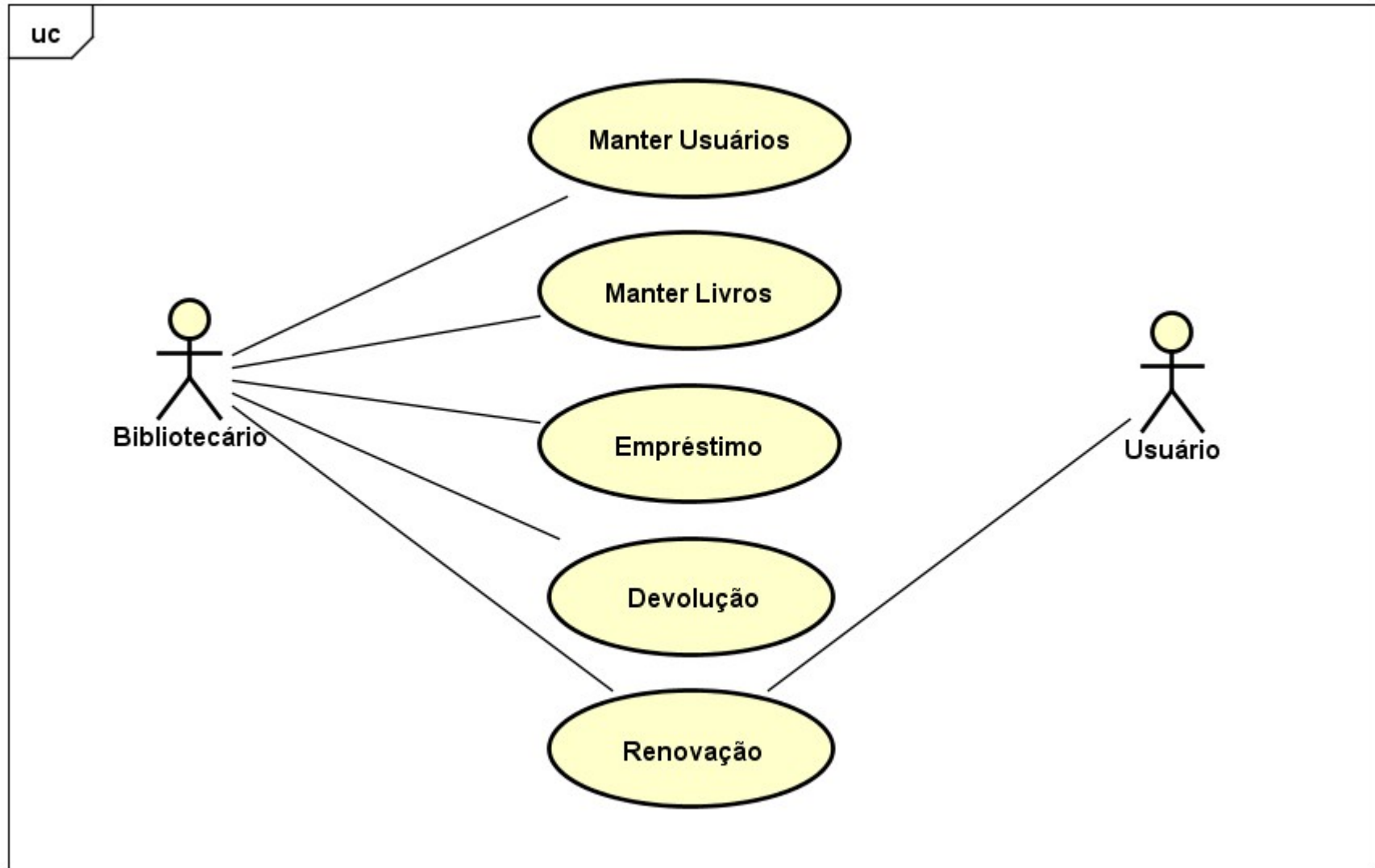
Prof. Leonardo Vianna do Nascimento  
Disciplina de Desenvolvimento de Sistemas I

# O que vamos fazer?

---

- **Projeto**
  - Sistema para gestão de uma biblioteca
- **Requisitos iniciais**
  - Manter cadastro de usuários
  - Manter cadastro de livros
  - Realizar empréstimos e devoluções
  - Realizar renovações
  - Controle de acesso de usuários

# Casos de Uso Iniciais



# Nesta Aula

---

- Implementaremos uma versão inicial do caso de uso ***Manter Usuários***
- Neste caso de uso deve-se executar as operações de inclusão, alteração, consulta e remoção de usuários
  - Um clássico CRUD (*Create, Retrieve, Update, Delete*)

# Dados de um Usuário

---

- ID
- Nome
- CPF
- Telefone para contato
- E-mail

# Implementação

---

1. Criar o projeto no Netbeans
2. Criar a classe de modelo
3. Configurar a classe para funcionar como um bean JSF
4. Implementar métodos na classe para implementar as operações do CRUD utilizando Hibernate
5. Construir as páginas de apresentação utilizando componentes JSF

# Criando o Projeto

---

- Criaremos um projeto Java Web com JSF como fizemos na última aula
- Porém, dessa vez utilizaremos também outro framework: o **Hibernate**



# Hibernate

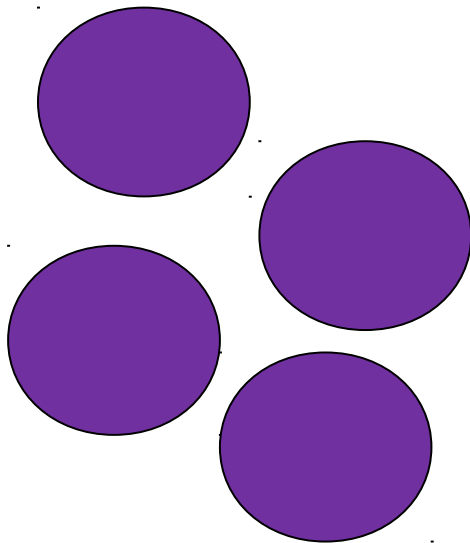
---

- Framework de mapeamento objeto-relacional (ORM, do inglês *Object Relational Mapping*)
  - Faz a ligação entre nosso modelo de classes Java para o modelo de tabelas de um banco de dados relacional
  - Nem sempre esse mapeamento é trivial e sua implementação manual tende a ser tediosa (muito código repetitivo)
  - O Hibernate permite realizarmos diversas operações de objetos Java em um banco de dados sem a necessidade de escrever muito código (muitas vezes sem usar SQL)

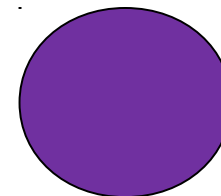


# ORM

**Objetos de modelo  
com dados  
resultantes de  
consultas**



**Objeto de modelo  
contendo dados  
para atualização**

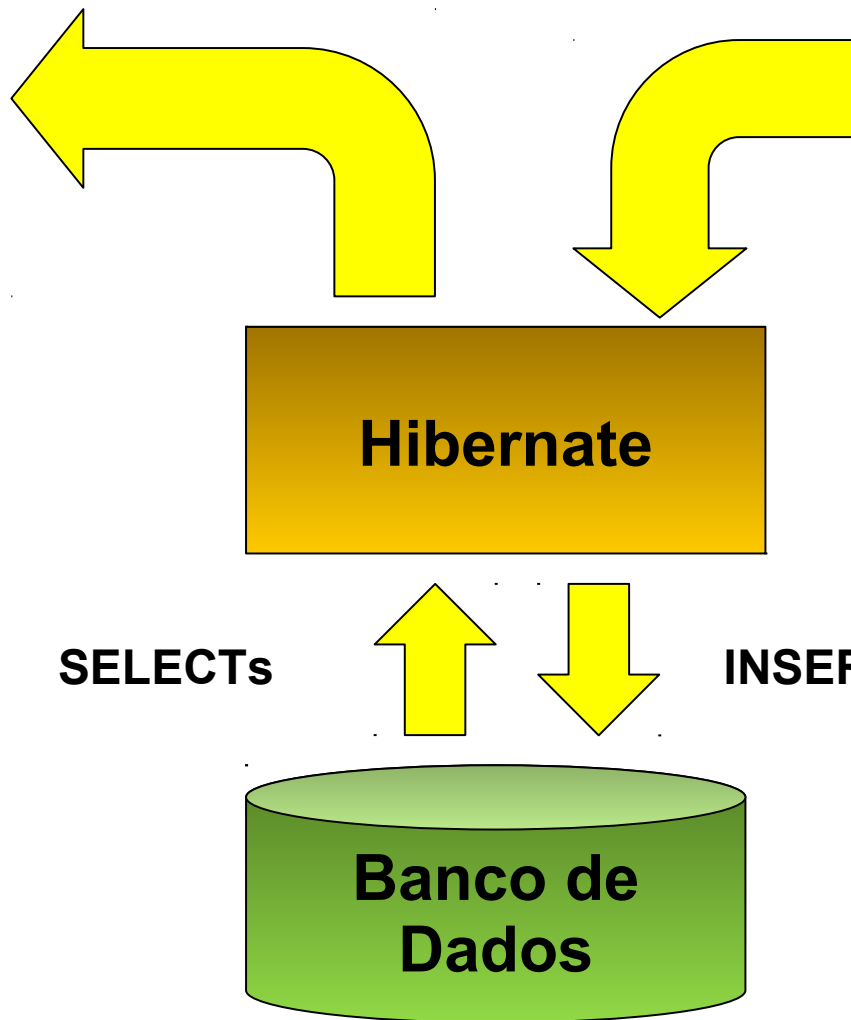


**Hibernate**

**SELECTs**

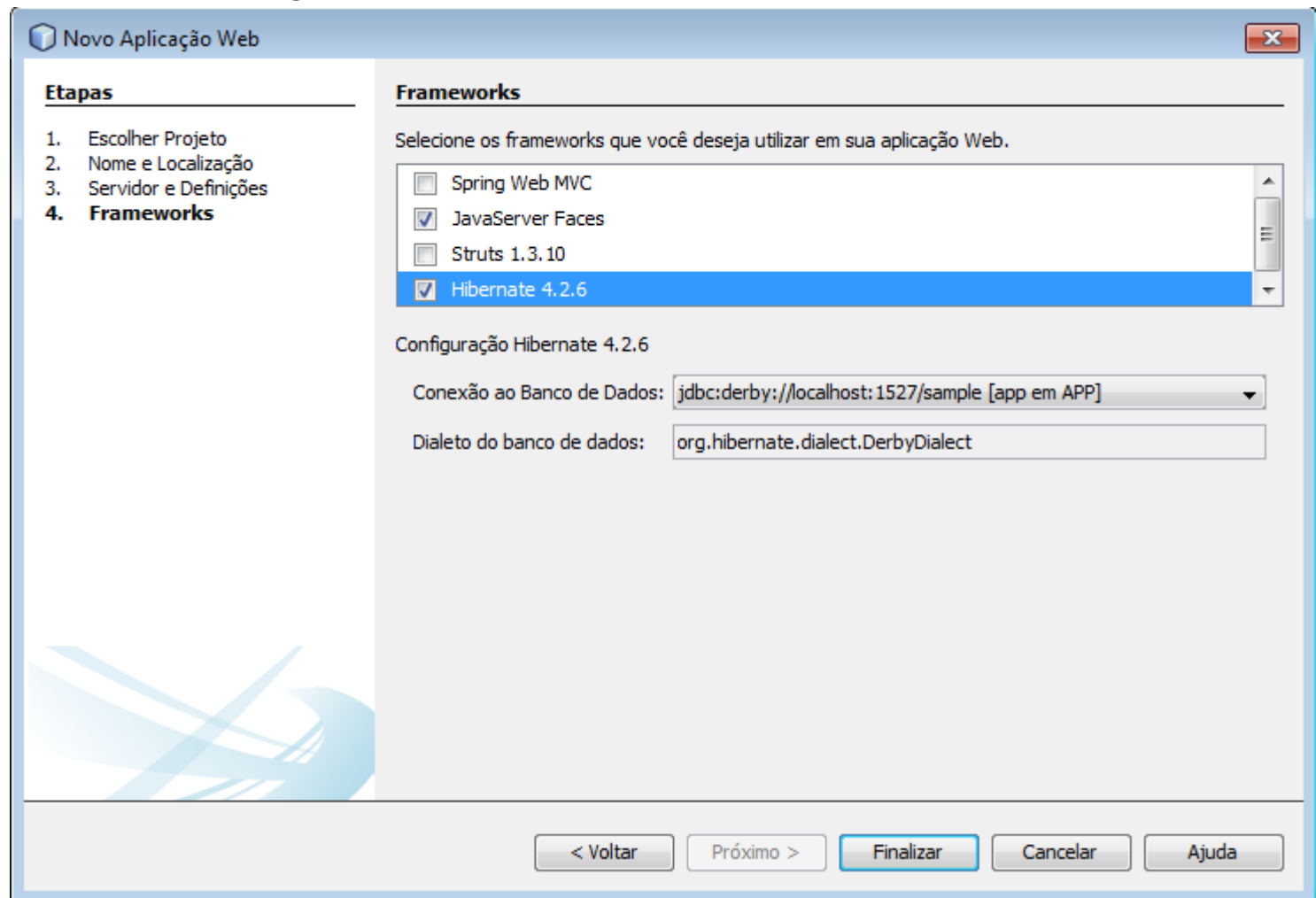
**INSERTs, UPDATES, DELETES**

**Banco de  
Dados**



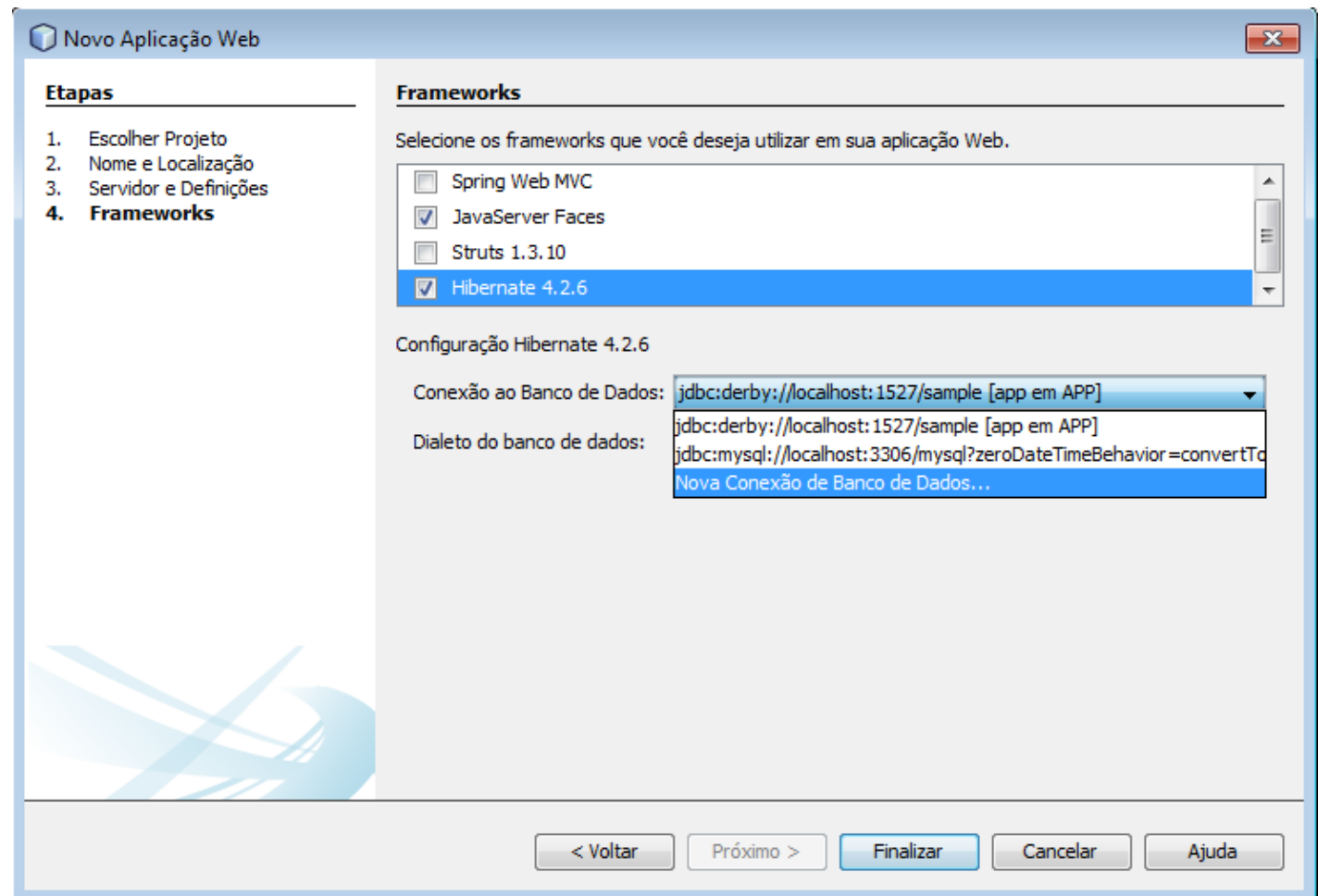
# Projeto com ORM (1)

- Ao criar um novo projeto, na última tela, selecione o Hibernate



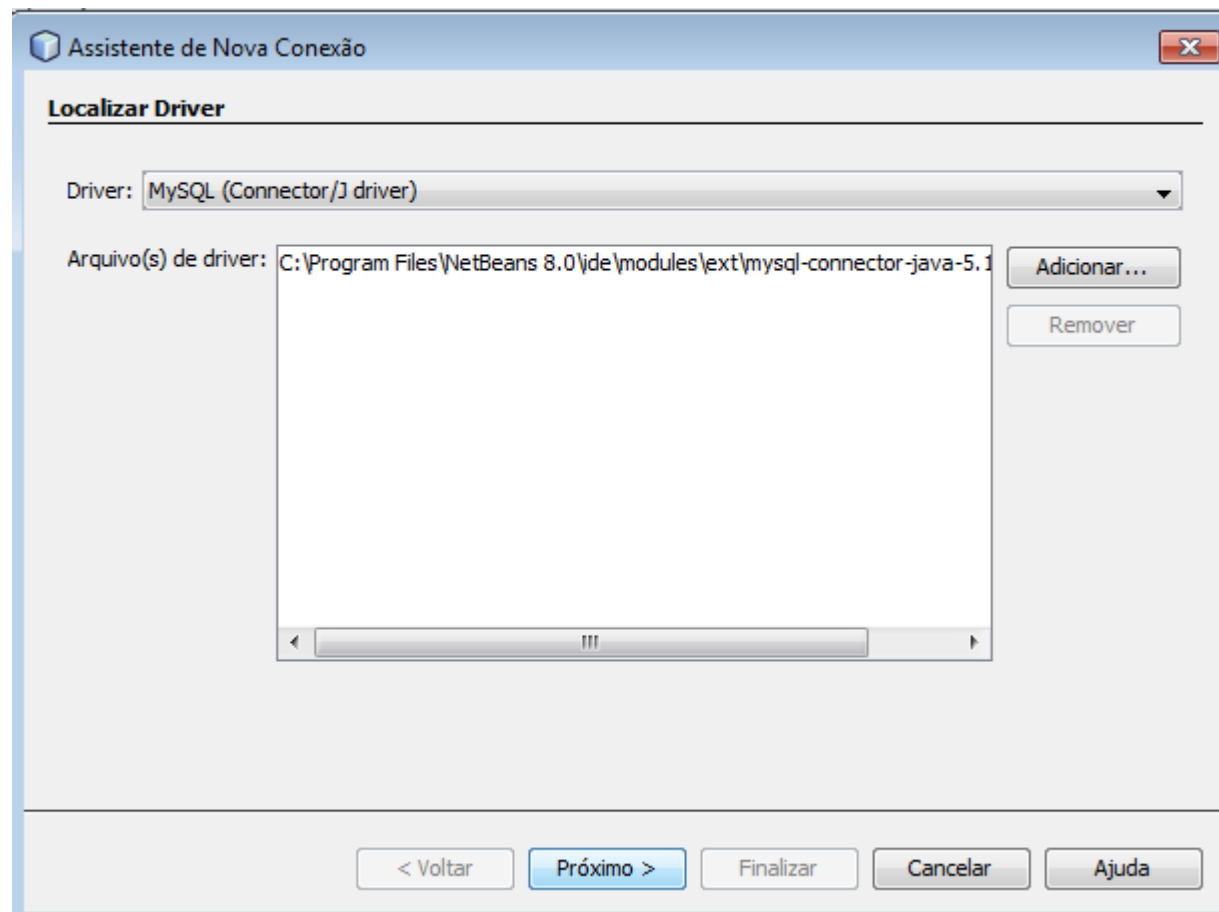
# Projeto com ORM (2)

- No campo *Conexão ao Banco de Dados* selecione *Nova Conexão de Banco de Dados...*



# Projeto com ORM (3)

- Na janela que abrir, selecione a opção relativa ao MySQL no campo *Driver* e clique em *Próximo*



# Projeto com ORM (4)

- Na próxima janela, preencha os dados para conexão e clique em *Finalizar* (finalize também o projeto)

Assistente de Nova Conexão

**Personalizar Conexão**

Nome do Driver: MySQL (Driver Connector/J) em MySQL (Connector/J driver)

Host: localhost Porta: 3306

Banco de dados: videolocadora

Nome do Usuário: root

Senha:

☐ Lembrar senha

Propriedades da Conexão Testar Conexão

JDBC URL: jdbc:mysql://localhost:3306/videolocadora?zeroDateTimeBehavior=convertToNull

Conexão Bem-sucedida.

< Voltar Próximo > Finalizar Cancelar Ajuda

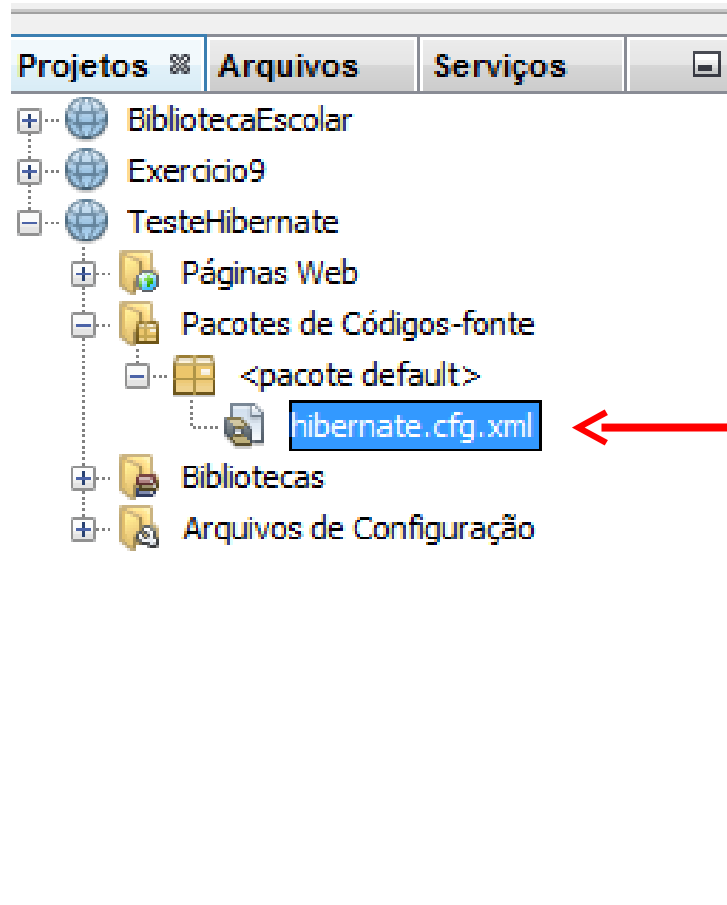
**Clique aqui para  
testar a conexão**

# Configurações do Hibernate

---

- As informações preenchidas ao criar o projeto indicam ao Hibernate como se conectar ao banco de dados
- Essas informações ficam armazenadas em um arquivo XML localizado junto com o código fonte do projeto, no pacote *default*

# Configurações do Hibernate



**Aqui está o arquivo! Ao abri-lo, é possível editar as informações de conexão ao banco de dados**

# Configurações do Hibernate

Design Código-fonte Histórico **Propriedades de JDBC**

**Fábrica de Sessão**

**Propriedades de JDBC**

Nome	Valor
hibernate.connection.driver_class	com.mysql.jdbc.Driver
hibernate.connection.url	jdbc:mysql://localhost:3306/videolocadora?zeroDateTimeBehav...
hibernate.connection.username	root

Adicionar... Editar... Remover

**Propriedades da Origem de Dados**

**Mapeamentos**

**Cache**

**Cache da Classe**

**Cache do Conjunto**

**Eventos** Adicionar

**Propriedades Opcionais**


**Wizard de edição do arquivo**



# Configurações do Hibernate

Design Código-fonte Histórico

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3 <hibernate-configuration>
4   <session-factory>
5     <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
6     <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
7     <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/videolocadora?ze
8     <property name="hibernate.connection.username">root</property>
9   </session-factory>
10 </hibernate-configuration>
11
```



Código fonte “real” do  
arquivo

# Atividade 1

---

- Vamos criar um banco de dados no MySQL para nossa aplicação chamado *biblioteca*
  - Criar uma tabela *usuarios* contendo os campos *id* (chave primário e auto incremento), *nome*, *cpf*, *telefone*, *email*
- Depois criaremos nosso projeto no Netbeans utilizando JSF e Hibernate
  - Vamos configurar o Hibernate para usar o banco de dados que acabamos de criar

# Implementação

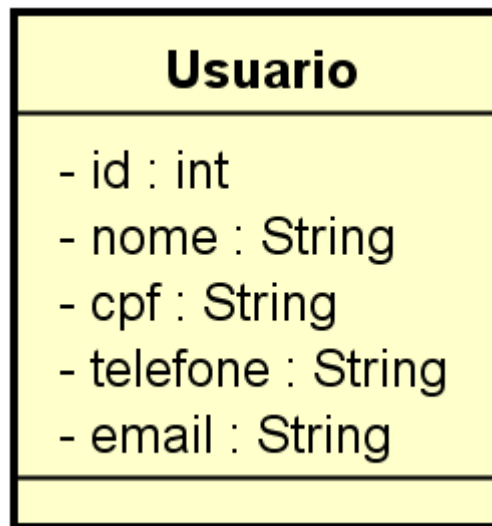
---

1. ~~Criar o projeto no Netbeans~~
2. **Criar a classe de modelo**
3. ~~Configurar a classe para funcionar como um bean JSF~~
4. ~~Implementar métodos na classe para implementar as operações do CRUD utilizando Hibernate~~
5. ~~Construir as páginas de apresentação utilizando componentes JSF~~

# Atividade 2

---

- Criar a classe de modelo chamada *Usuario*



# Implementação

---

1. ~~Criar o projeto no Netbeans~~
2. ~~Criar a classe de modelo~~
3. **Configurar a classe para funcionar como um bean JSF**
4. ~~Implementar métodos na classe para implementar as operações do CRUD utilizando Hibernate~~
5. ~~Construir as páginas de apresentação utilizando componentes JSF~~

# O que é um *bean*?

---

Objeto Java que pode ser configurado e manipulado ***sem necessidade de programação***

# Beans no JSF

---

- No contexto do JSF, *beans* armazenam dados manipulados e/ou exibidos por páginas web
- É a implementação JSF que:
  - Cria e descarta *beans*, conforme necessário
  - Lê as propriedades dos *beans* quando da exibição de uma página web
  - Atribui valores às propriedades dos *beans* quando um formulário é submetido

# Relembrando o exemplo...

---

- No exemplo da aula passada, o seguinte campo lia e atualizava o valor da propriedade *numero1* do *bean* chamado “*bean*”

```
<h:inputText value="#{bean.numero1}"/>
```

- A implementação JSF precisa localizar um objeto chamado *bean*



# Expressões de Valor

---

- Mas como o JSF sabe que deve acessar a propriedade *numero1* do bean chamado *bean*?
  - No atributo *value* do componente *inputText* usamos uma expressão de valor (*value expression*)
  - Essa expressão é escrita usando uma Linguagem de Expressão ou *Expression Language* (EL)
  - Toda expressão EL deve começar com # e estar dentro de chaves
  - Para acessar uma propriedade de um bean basta escrever o nome do bean seguido de um ponto e do nome da propriedade desejada

# Encontrando o *bean*

---

- No nosso exemplo, uma classe *SomaBean* foi definida da seguinte forma:

```
@ManagedBean (name="bean" )
```

```
@RequestScoped
```

```
public class SomaBean {
```

```
    . . .
```

```
}
```

- O JSF procura por uma classe que possua a anotação *@ManagedBean* cujo *name* seja igual ao nome do *bean* encontrado na página

# Definindo um *bean*

---

- Todo *bean* gerenciado pelo JSF é chamado de *bean gerenciado* ou *managed bean*
- Para definir uma classe como tal basta inserir a anotação `@ManagedBean` acima da definição da classe
  - O atributo *name* é opcional e permite especificar o nome que desejamos para o *bean*
  - Caso seja omitido, o nome do *bean* será obtido a partir do nome da classe, convertendo-se a primeira letra em minúscula

# Exemplos

---

```
@ManagedBean (name="soma")  
public class SomaBean { ... }
```

Nome do bean será *soma*



```
@ManagedBean  
public class SomaBean { ... }
```

Nome do bean será *somaBean*



# ATENÇÃO!!!

---

- Para usar a anotação *@ManagedBean* é necessário importar a classe *ManagedBean* definida no pacote *javax.faces.bean*

```
import javax.faces.bean.ManagedBean
```

- CUIDADO
  - O Java EE define outra anotação chamada *ManagedBean* no pacote *javax.annotation*, que não funciona com o JSF
  - Não confundir!!!!

# Propriedades dos Beans

---


- *Beans* possuem propriedades que possuem valores
  - Toda propriedade deve ter um nome e um tipo
- Para definir uma propriedade devemos escrever um método para obter seu valor (*get*) e, opcionalmente, um método para alterá-la (*set*)

# Propriedades dos Beans

```
public class SomaBean {  
    ...
```

```
    public int getNumero1() {  
        return numero1;  
    }
```

Método para obter o valor da propriedade *numero1*




```
    public void setNumero1(int numero1) {  
        this.numero1 = numero1;  
    }
```

Método para alterar o valor da propriedade *numero1*



```
    ...  
    public int getSoma() {  
        return numero1 + numero2;  
    }
```

Método para obter o valor da propriedade *soma*



```
}
```

# Propriedades dos Beans

---

- O nome da propriedade será sempre a parte do nome dos métodos que aparece após o *get* ou *set*, com a primeira letra minúscula
- Propriedades podem possuir apenas o método *get*, sendo somente leitura
- Para propriedades do tipo *boolean* é possível escolher entre o prefixo *is* ou *get*
  - Uma propriedade *boolean* chamada *ativo* pode possuir um método de consulta chamado de *getAtivo* ou *isAtivo*



# Quando um *bean* é criado?

**1**

Uma página é acessada e uma referência ao *bean* é encontrada pela primeira vez

```
...  
<h:inputText value="#{soma.numero1}"/>  
...
```



Referência ao *bean* chamado *soma*

# Quando um *bean* é criado?

---

**2**

A implementação JSF localiza que o nome *soma* está associado à classe *SomaBean* através da anotação `@ManagedBean`

```
@ManagedBean (name="soma")  
public class SomaBean { ... }
```

**3**

A implementação JSF cria um objeto da classe *SomaBean*

**4**

Toda vez que o *bean* chamado *soma* for referenciado, o objeto criado será acessado (enquanto ele existir)

# Até quando um *bean* existe?

---

- Todo *bean* tem um *escopo*
  - Um *escopo* define até quando o *bean* deve existir
- Ao se definir a classe de um *bean* devemos especificar seu *escopo*
- O *escopo* pode ser definido usando anotações presentes no pacote *javax.faces.bean*
  - *@RequestScoped*
  - *@SessionScoped*
  - *@ApplicationScoped*
  - *@ViewScoped*

# @RequestScoped

---

- Define um *bean* com escopo de requisição
- O escopo de requisição começa no momento em que uma requisição HTTP é recebido pelo servidor e termina quando a resposta é enviada para o cliente
  - Um *bean* criado neste escopo será criado toda vez que a página for acessada e destruído quando a resposta for enviada ao cliente
  - Usamos este escopo quando um *bean* precisa ser mantido apenas em uma determinada página

# @RequestScoped – Exemplo

---

- Em nosso exemplo, a classe *SomaBean* define um *bean* com escopo de requisição

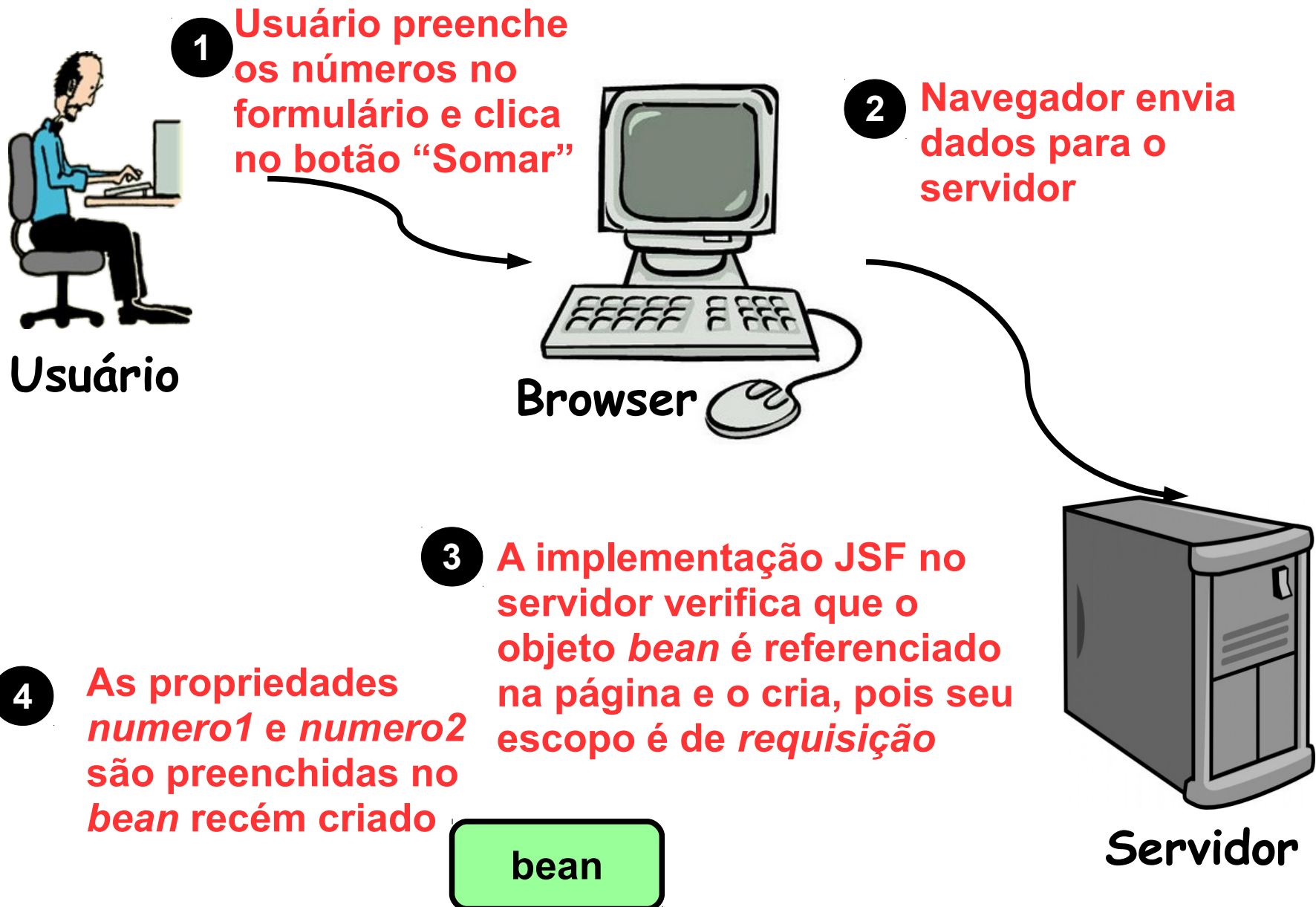
```
@ManagedBean (name="bean")
```

```
@RequestScoped
```

```
public class SomaBean { ... }
```

- Toda vez que o formulário for enviado o *bean* será criado e depois que a página com a resposta for enviada ele será destruído

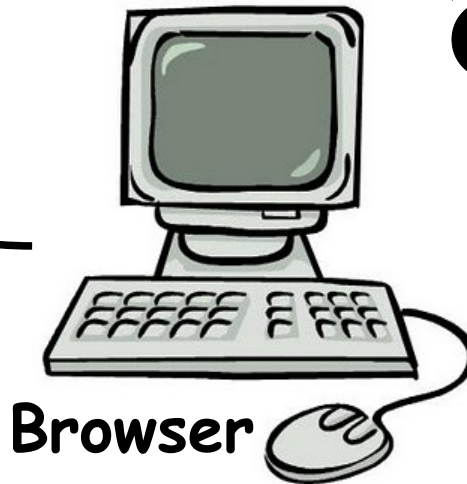
# @RequestScoped – Exemplo



# @RequestScoped – Exemplo

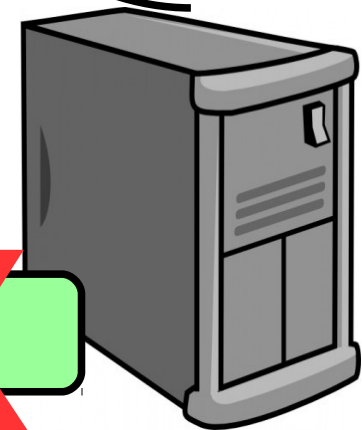
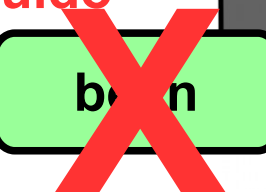


Usuário



Browser

6 A nova página é enviada ao navegador e o *bean* é destruído



Servidor

5 A implementação JSF reconstrói a página *index.xhtml* preenchendo os campos com os valores das respectivas propriedades do *bean*

7 Navegador recebe a nova página e a exibe

# Atividade 3

---

- Vamos configurar a classe *Usuario* como um bean gerenciado com nome *usuarioBean* e escopo de requisição
- Cada atributo será uma propriedade leitura/escrita
  - Devem ser criados métodos SET/GET para cada um



# Implementação

---

- ~~1. Criar o projeto no Netbeans~~
- ~~2. Criar a classe de modelo~~
- ~~3. Configurar a classe para funcionar como um bean JSF~~
- 4. Implementar métodos na classe para implementar as operações do CRUD utilizando Hibernate**
- ~~5. Construir as páginas de apresentação utilizando componentes JSF~~

# Atividade 4

---

- Vamos realizar a configuração de nossa classe *Usuario* para que o Hibernate saiba como mapeá-la para o banco de dados
  - Por padrão, o Hibernate associa à classe com uma tabela com o mesmo nome no banco (não é o nosso caso, por isso precisaremos dizer a ele qual é o nome da tabela)
  - Precisamos dizer a ele quem representa a chave primária
  - Por padrão, cada atributo da classe será associado a um campo da tabela com o mesmo nome (felizmente é o nosso caso :) )

# Anotando a Classe Modelo(1)

---

- **@Entity**
  - Serve para marcar uma classe de modelo como uma entidade persistente
- **@Table**
  - Serve para informar dados sobre a tabela que corresponde à classe no banco de dados
  - A propriedade *name* serve para especificar o nome da tabela
  - Esta anotação é desnecessária se a tabela tiver exatamente o mesmo nome da classe

# Anotando a Classe Modelo(2)

---

- @Id
  - Serve para marcar um campo como o campo de identificação da entidade (geralmente correspondendo à chave primária da tabela)
- @GeneratedValue
  - Diz que o valor do campo, ao ser inserido, será gerado automaticamente pelo banco (*auto\_increment*)
- Ambas as anotações podem ser colocadas sobre a definição do atributo ou do método *get* referente ao atributo

# Concluindo o Mapeamento

---

- Precisamos incluir uma entrada para a classe no arquivo *hibernate.cfg.xml*
  - Clique em *Mapeamentos*
  - Clique em *Adicionar*
  - Preencha o campo *Classe* com o nome completo da mesma (*modelo.Usuario*)
  - Clique em *OK*

# Atividade 5

---

- Vamos agora implementar os métodos para inclusão, alteração, consulta e remoção de usuário
- Faremos isso hoje utilizando um padrão chamado de *Active Record*
  - Os próprios objetos *Usuario* saberão salvar a si mesmos, consultar e remover
  - Acrescentaremos na classe *Usuario* três métodos: *salvar* (para incluir e alterar), *carregar* e *remover*

# Implementando os Métodos

---

- Todos os comandos são feitos a partir de um objeto ***Session***
  - Principal interface usada em aplicações Hibernate
    - Todas as operações explícitas de persistência são realizadas através de um objeto Session
  - Objeto leve
    - Fácil de criar
    - Fácil de destruir
  - Objetos Session não são threadsafe
    - Devem ser usados em um único thread (requisição)
    - Para threads adicionais, crie sessions adicionais

# Métodos de Session (1)

---

- *save*
  - Insere um objeto no banco de dados
- *update*
  - Atualiza um objeto no banco de dados
- *saveOrUpdate*
  - Insere ou altera um objeto no banco de dados, dependendo se ele já existe ou não



# Métodos de Session (2)

---

- *delete*
  - Remove um objeto do banco de dados
- *load*
  - Consulta um objeto pelo ID

# Criando um *Session*

---

- Uma aplicação obtém uma *Session* a partir de uma ***SessionFactory***
  - Objeto pesado; lento para inicializar e destruir
  - Geralmente tem-se uma apenas para toda a aplicação
  - Deve-se ter uma *SessionFactory* para cada banco de dados utilizado
- Realiza cache de comandos SQL, dados e metadados usados em tempo de execução

# Configuration

---

- É o ponto de partida para iniciar o Hibernate
  - Inicializado com propriedades de configuração do sistema
  - Especifica a localização de dados e arquivos de mapeamento, objetos, configuração do banco de dados, pool de conexões, dialeto do SQL do banco, etc.
  - Geralmente obtém a configuração via arquivos .properties, XML ou propriedades dinâmicas
    - Padrão: hibernate.cfg.xml
- Cria a SessionFactory

# Criando SessionFactory

---

- Usaremos um recurso do Netbeans para isso
  - Menu *Novo*, opção *HibernateUtil.java*
  - Criaremos essa classe em um pacote chamado *persistencia*
- Também adicionaremos uma configuração ao arquivo *hibernate.cfg.xml*
  - Vá em *Propriedades Diversas*
  - Clique em *Adicionar*
  - Na janela que abrir, selecione a propriedade *hibernate.current\_session\_context\_class* e altere-a para *thread*

# Implementar Métodos

---

- Agora podemos implementar os métodos de persistência
- Em cada um precisaremos:
  - Obter a sessão corrente do Hibernate
  - Iniciar uma transação
  - Realizar a operação no banco
  - Finalizar a transação

# Interface com o Usuário

---

- Hoje implementaremos as operações de inclusão, consulta e remoção
  - A operação de alteração ficará para outra aula
- Precisaremos implementar três páginas:
  - Uma página inicial de entrada (*index.xhtml*) contendo um menu da aplicação
  - Uma página de inclusão
  - Uma página de consulta, com botão para remover o usuário

# Tag *link*

---

- Representa um link que, quando clicado, abre outra página da aplicação
- Atributo *value*
  - *Texto do link*
- Atributo *outcome*
  - *Nome da página a ser aberta (pode ser sem extensão)*

# Tag *panelGrid*

---

- Organiza componentes em um formulário em uma tabela
- Cada componente dentro do *panelGrid* é colocado em uma célula da tabela
  - O preenchimento é feito da esquerda para direita e de cima para baixo
    - Por exemplo, um *panelGrid* com três colunas e nove componentes terá três linhas, cada uma com três componentes



# Tag *panelGrid*

---

- Atributo *columns*
  - Permite especificar o número de colunas da tabela a ser gerada

# Tag *outputLabel*

---

- Cria um rótulo associado a um componente de entrada de um formulário
  - Recomendado seu uso em razão de acessibilidade (leitores de tela leem o texto do *label* associado a um campo de entrada)
- *Atributos value*
  - Texto mostrado no componente
- *Atributo for*
  - ID do componente associado ao label

# Tag *inputText*

---

- Campo de texto de uma linha
- Atributos *id*
  - *Permite especificar o ID HTML do componente*
- Atributos *maxlength* e *size*
  - *Mesma função do HTML*
  - *Permite especificar, respectivamente, a quantidade máxima de caracteres que pode ser digitada e o tamanho visual do componente (em caracteres)*

# Tag *inputText*

---

- Atributo *value*
  - Valor do componente, geralmente associado a uma propriedade de um *bean*
- Atributo *label*
  - Uma descrição do componente para uso em mensagens de erro
- Atributo *required*
  - *true* indica que o campo é obrigatório
  - *false* indica que o campo não é obrigatório (valor padrão para o atributo)

# Tag *commandButton*

---

- Um botão que executa uma ação
- Atributo *value*
  - *Texto do botão*
- Atributo *actionListener*
  - *Permite especificar um método de um bean a ser executado ao se clicar no botão*
  - *O método deve ser void*

# Tag *button*

---

- Faz o mesmo que *link*, mas na forma de um botão

# Mensagens

---

- Qualquer objeto da aplicação pode criar uma mensagem e adicioná-la a uma fila de mensagens
- Mensagens são classificadas em quatro categorias
  - Informação
  - Aviso
  - Erro
  - Erro Fatal

# Mensagens

---

- Todas as mensagens podem conter um **resumo** e um **detalhamento**
  - Exemplo de resumo: *Entrada inválida*
  - Exemplo de detalhamento: *O número informado é maior do que o valor máximo permitido*
- Tag *messages*
  - Permite exibir uma lista de todas as mensagens vindas do controle



# Criando Mensagens

---

- Exemplo de definição de grau de severidade:

```
public void salvar() {  
    FacesContext context = FacesContext.getCurrentInstance();  
    FacesMessage msg;  
    try {  
        imovel.alterar();  
  
        msg = new FacesMessage(FacesMessage.SEVERITY_INFO,  
                               "Imóvel alterado com sucesso!", "");  
  
    } catch (Exception e) {  
  
        msg = new FacesMessage(FacesMessage.SEVERITY_ERROR,  
                               e.getMessage(), "");  
    }  
    context.addMessage(null, msg);  
    return "formConsultaAltImovel";  
}
```