

## Sumário

<b>1. Spring MVC</b>	<b>3</b>
1.1 Porque utilizar um Framework MVC?	3
1.2 Como surgiram os Frameworks	4
1.3 Configurando o Spring no projeto	4
1.4 Configurando o XML do Spring	5
1.5 Entendendo o Controller do Spring MVC	6
1.6 Atividade – Criando um projeto com Spring MVC	7
1.7 Adicionando despesas	10
1.8 Organizando as views	11
1.9 Atividade – Cadastrando despesas	13
1.10 Utilizando o Bean Validation	15
1.11 Personalizando mensagens do Bean Validation	17
1.12 Atividade - Bean Validation	17
1.13 Listando despesas e enviando objetos para view	19
1.14 Atividade – Exibindo despesas cadastradas	21
1.15 Redirecionando para outra ação	22
1.16 Atividade – Removendo e alterando despesas	23
1.17 AJAX	26
1.18 Quitando despesas com AJAX	28
1.19 Atividade – Quitando uma despesa com AJAX	29
<b>2. Spring MVC: Autenticação e Autorização</b>	<b>33</b>
2.1 COOKIES	33
2.2 SESSÃO	33
2.3 Registrando o usuário na sessão	34
2.4 Atividade – Tela de login	34
2.5 Validar usuários com interceptadores	36
2.6 Atividade – Usando interceptador para validar usuário	38
<b>3. Sprint IoC</b>	<b>40</b>
3.1 Removendo acoplamento com inversão de controle	40
3.2 Contexto de Injeção de Dependência (CDI)	43
3.3 Contexto de Injeção de Dependência (CDI)	43
3.4 Atividade – IoC com Spring	45

3.5	Aplicando CSS à aplicação .....	47
3.6	Atividade - Aplicando CSS à aplicação .....	49
4.	Spring com JPA .....	50
4.1	Mapeamento objeto relacional .....	50
4.2	JPA.....	50
4.3	Mapeando uma classe com JPA.....	50
4.4	Integração do Hibernate com o Spring.....	52
4.5	DAO do JPA e Injeção do EntityManager .....	53
4.6	Atividade – Configurando o projeto para trabalhar com JPA.....	55
5.	Apêndice A – Convertendo o projeto em MAVEN .....	61
5.1	MAVEN PROJECT .....	61
5.2	Atividade – Convertendo o projeto em Maven Project.....	63

# 1. Spring MVC

## 1.1 Porque utilizar um Framework MVC?

Ao usar JDBC puro, precisamos conhecer os comandos SQL. Os IDEs que trabalham com Java não dão suporte a erros de compilação e nem auto completar para essa linguagem. A curva de aprendizagem e a probabilidade de cometer algum erro é grande. Temos que nos preocupar apenas com o código Java não acham?

A API do Servlet encapsula muitos detalhes técnicos do protocolo HTTP, o que nos dá certas vantagens, porém, tem alguns problemas. Um exemplo é o fato de tudo que vem no request é String e nos obriga a fazer a conversões.

Resumindo, além de nos preocuparmos com a regra de negócio e a implementação dos recursos, tivemos que ficar atentos aos requisitos não funcionais e a pequenos detalhes necessários ao funcionamento ideal do projeto.

## 1.2 Como surgiram os Frameworks

Diante das repetidas dificuldades enfrentadas pelas empresas, essas começaram a criar conjuntos de bibliotecas que encapsulavam detalhes técnicos e deixavam o programador livre para trabalhar com montagem da tela e as regras de negócio. Começava a surgir então pequenos Frameworks.

Por muito tempo, o framework mais conhecido era o **Struts**. Porém sofreu poucas atualizações e abou não acompanhando as necessidades do mercado. Ficou obsoleto.

Hoje um dos frameworks mais famoso é o **Spring** que vem sofrendo constantes atualizações e se tornou uma forte ferramenta para desenvolvimento MVC.

Para podermos aprender como tirar projeto desse poderoso framework, vamos criar um módulo separado para nossa aplicação utilizando seus recursos do **Spring MVC**.

## 1.3 Configurando o Spring no projeto

O **Spring MVC** vem junto com as bibliotecas do framework Spring e pode ser baixado do site: <http://springsource.org>.

Depois de baixado, adicionamos os jars do Spring MVC no nosso projeto dentro do diretório /WEB-INF/lib, precisamos declarar um Servlet que fará o papel de *Front Controller* da aplicação. Para isso, editamos o arquivo **web.xml**.

```
<!-- Declaracao do servlet do Spring MVC abaixo -->
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/spring-context.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

É uma declaração comum de Servlet, como vimos no início do curso. A única novidade é a presença do elemento **<init-param>** que adiciona um parâmetro **contextConfigLocation** com um caminho para o xml do Spring: **/WEB-INF/spring-context.xml**.

## 1.4 Configurando o XML do Spring

Porque o Spring tem seu próprio XML? O Spring é muito mais que um framework MVC, por isso, o **web.xml** não é suficiente para suprir todas as suas necessidades. Por esse motivo anotamos no web.xml apenas o apontamento para o XML do Spring.

O próximo passo então, é configurar esse XML. A primeira coisa que iremos fazer nesse arquivo é habilitar o uso de anotações do Spring MVC e dizer ao Spring onde encontrar nossas classe:

```
<context:component-scan base-package="br.com.igordev.despesas" />
<mvc:annotation-driven />
```

Temos também que avisar ao Spring, onde colocaremos nossas views (arquivos .jsp). Para esse novo projeto iremos usar a pasta /WEB-INF/views. O Spring tem uma classe responsável por essa tarefa, basta informarmos os parâmetros:

```
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

Isso já é suficiente para começarmos a trabalhar com o Spring MVC.

## 1.5 Entendendo o Controller do Spring MVC

Para que o Spring consiga encontrar nossas classes **Controller**, devemos salvá-las na raiz ou em algum subpacote de **br.com.igordev.despesas** como foi configurado no spring-context.xml. Outro ponto importante é anotarmos essa classe com **@Controller**.

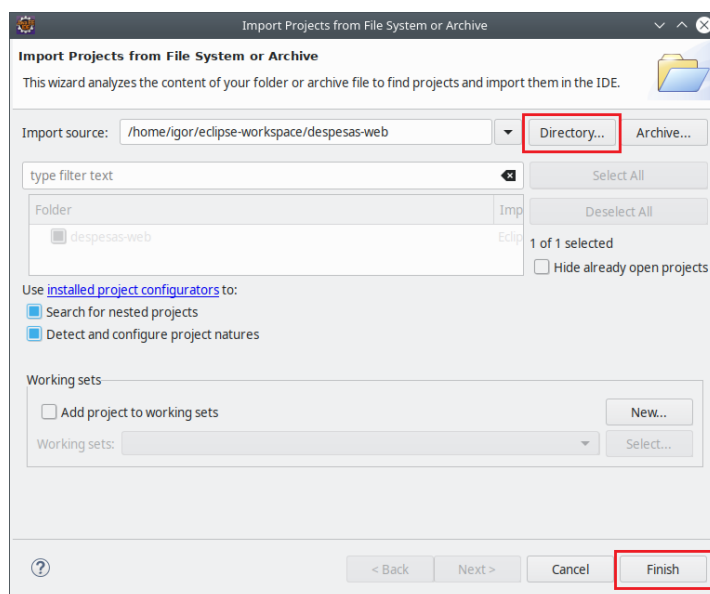
Depois de criarmos e anotarmos a classe, o Spring identifica os métodos dessa classe com ações (Action) e podemos associá-los diretamente à uma URL por meio da anotação **@RequestMapping("/url")**. Outro detalhe é que esses métodos devem retornar uma String indicando qual página deverá ser aberta ao final da execução (como foi feito na nossa interface no capítulo 9 lembra?).

Na atividade a seguir vamos criar o novo projeto, configurar o Spring MVC e criar a lógica do HelloWorld.

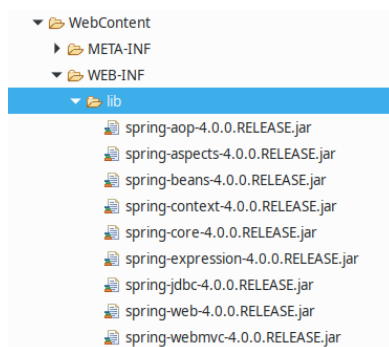
## 1.6 Atividade – Criando um projeto com Spring MVC

**1.6.1** Procure na pasta **material de apoio** o projeto **despesas-web.zip** e descompacte-o no workspace do eclipse.

**1.6.2** No eclipse, acesse o menu **File > Open Project from File System**. Clique no botão **Directory...**, localize o projeto despesas-web descompactado no item **1.6.1** e clique em **OK** e em seguida **Finish**.



**1.6.3** Procure na pasta **material de apoio** a subpasta **jars-spring4**. Copia todo conteúdo dessa pasta para /WEB-INF/lib do projeto despesas-web:



**1.6.4** Abra o **web.xml** do projeto e insira o mapeamento para o *Front Controller* do Spring dentro da tag `<web-app></web-app>`:

```
<!-- Declaracao do servlet do Spring MVC abaixo -->
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/spring-context.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

**1.6.5** Vamos fazer um simples Olá Mundo, para testarmos nossa configuração:

- Crie uma nova classe chamada `OlaMundoController` no pacote `br.com.igordev.despesas.controller`:
- Adicione nessa classe o seguinte conteúdo:



```
@Controller
public class OlaMundoController {
    @RequestMapping("/olaMundoSpring")
    public String execute() {
        System.out.println("Executando um serviço com o Spring MVC");
        return "ok";
    }
}
```

Use CTRL + SHIFT + O para fazer as importações:

- Crie uma pasta **views** para nossos **JSPs** que deve ficar dentro da pasta WebContent/WEB-INF
- Falta o JSP que será exibido após a execução da nossa lógica. Crie o JSP **ok.jsp** no diretório WebContent/WEB-INF/views do projeto com o conteúdo:

```
<html>
<body>
    <h2>Olá mundo com o Spring MVC!</h2>
</body>
</html>
```

**1.6.8** Adicione o projeto no servidor do Tomcat. Entre na aba **servers** clique sobre o Tomcat com o botão direito, mova o projeto **despesas-web** para **configured** e clique **Finish**.

**1.6.7** Reinicie o TomCat e acesse: <http://localhost:8080/despesas-web/olaMundoSpring>.

## 1.7 Adicionando despesas

Para podermos nos concentrar apenas nas funcionalidades do Spring, o projeto despesas-web veio com o **modelo** e o **dao** já implementados. Vamos então precisar de um formulário HTML para fazer a adição de novas despesas.

```
<html>
<body>
  <h3>Adiciona despesas</h3>
  <br />
  <br />
  <form action="adicionaDespesa" method="post">
    Descrição: <br />
    <textarea rows="5" cols="100" name="descricao"></textarea><br />
    <input type="submit" value="Adicionar" />
  </form>
</body>
</html>
```

O formulário submete para a ação **adicionaDespesa**. Devemos então criar um controller e um método mapeado para essa ação:

```
@Controller
public class DespesasController {

    @RequestMapping("adicionaDespesa")
    public String adiciona() {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.adiciona(despesa);
        return "despesa-adicionada";
    }
}
```

Mas, como montaremos o objeto **despesa** para passarmos ao nosso DAO se não temos um `HttpServletRequest`? Uma das grandes vantagens de frameworks modernos é que eles conseguem popular os objetos para nós. Basta que de alguma forma, nós façamos uma ligação entre o campo que está na tela com o objeto que queremos popular. Essa ligação é feita através da criação de um parâmetro do método `adiciona`. Esse parâmetro é o objeto que deverá ser populado pelo Spring MVC:

```
@Controller
public class DespesasController {

    @RequestMapping("adicionaDespesa")
    public String adiciona(Despesa despesa) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.adiciona(despesa);
        return "despesa-adicionada";
    }
}
```

O Spring irá instanciar nosso POJO Despesa e popular com os dados vindos do formulário HTML, bastando para isso, colocar os **inputs** do formulário com os mesmos nomes dos campos da classe.

## 1.8 Organizando as views

Assim como na pasta **src** do **Model**, vamos organizar os JSPs relacionados em subpastas. Para as despesas, vamos criar a subpasta /WEB-INF/views/despesa.

Por padrão, o Spring MVC não procura em subpastas, procura apenas na pasta views. Vamos mudar o retorno do método adiciona e devolver o nome da subpasta e o nome da página JSP. Nesse caso o retorno fica como despesa/adicionada.

```
@Controller
public class DespesasController {

    @RequestMapping("adicionaDespesa")
    public String adiciona(Despesa despesa) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.adiciona(despesa);
        return "despesa/adicionada";
    }
}
```

Porém, temos um problema, o formulário também deverá ficar na pasta despesa e o acesso direto a pasta WEB-INF é proibido pelo servlet-container. Vamos então criar outro método na classe DespesaController que retorna o formulário e fazer o mapeamento para URL “novaDespesa”:

```
@Controller
public class DespesasController {

    @RequestMapping("novaDespesa")
    public String form() {
        return "despesa/formulario";
    }

    @RequestMapping("adicionaDespesa")
    public String adiciona(Despesa despesa) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.adiciona(despesa);
        return "despesa/adicionada";
    }
}
```

A estrutura de diretórios fica assim:

```
WEB-INF
-views
  -despesa
    -formulario.jsp
    -adicionada.jsp
```

## 1.9 Atividade – Cadastrando despesas

**1.9.1** O primeiro passo é criar nosso formulário para adicionar uma despesa. Para isso crie uma pasta **despesa** dentro da pasta WebContent/WEB-INF/views. Dentro da pasta despesa adicione um novo arquivo **formulario.jsp**.

```
<html>
<body>
    <h3>Adiciona despesas</h3>
    <br />
    <br />
    <form action="adicionaDespesa" method="post">
        Descrição: <br />
        <textarea rows="5" cols="100" name="descricao"></textarea><br />
        <input type="submit" value="Adicionar" />
    </form>
</body>
</html>
```

**1.9.2** No pacote **br.com.igordev.despesas.controller** adicione um novo controller: DespesasController. Nele vamos adicionar o método para chamada do form e também a ação para adicionar uma nova despesa (CTRL + SHIFT + O para importações):

**Nota:** não esqueça das anotações @Controller e @RequestMapping

```
@Controller
public class DespesasController {

    @RequestMapping("novaDespesa")
    public String form() {
        return "despesa/formulario";
    }
}
```

```
@RequestMapping("adicionaDespesa")
public String adiciona(Despesa despesa) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.adiciona(despesa);
    return "despesa/adicionada";
}
```

**1.9.3** Finalmente, vamos criar o retorno da ação. Crie o arquivo adicionada.jsp na pasta WEB-INF/views/despesa:

```
<html>
<body>
    <h1>Nova despesa adicionada com sucesso!</h1>
</body>
</html>
```

**1.9.4** Vamos precisar criar a tabela despesa no BD **imovel**. Acesse o DBEaver, conecte ao BD imovel e rode o script abaixo:

```
use imovel;
create table despesa (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    descricao VARCHAR(255),
    pago BOOLEAN,
    dataPagamento DATE);
```

**1.9.5** Reinicie o Tomcat e acesse: <http://localhost:8080/despesas-web/novaDespesa>

## 1.10 Utilizando o Bean Validation

Na atividade anterior construímos um formulário e conseguimos adicionar um despesa. Porém, o que impediria o usuário de cadastrar uma despesa sem descrição?

A partir do Java EE 6 temos uma especificação que resolve este problema. A JSR 303, também conhecida como Bean Validation.

Com o Bean Validation declaramos através de anotações as regras de validação dentro do nosso modelo, por exemplo, na nossa despesa:

```
public class Despesa {  
  
    private Long id;  
    @Size(min=5)  
    private String descricao;  
    private boolean pago;  
    private Calendar dataPagamento;  
    ...  
}
```

Essa validação indica que o campo deve conter no mínimo 5 caracteres. O próximo passo é avisar ao Spring que é necessário processar essa validação. Para isso, anotamos o parâmetro da ação com @Valid:

```
@RequestMapping("adicionaDespesa")  
public String adiciona(@Valid Despesa despesa) {  
    JdbcDespesaDao dao = new JdbcDespesaDao();  
    dao.adiciona(despesa);  
    return "despesa/adicionada";  
}
```

Antes da execução do método o Spring MVC executa a validação, ou seja, será verificado se a descrição da despesa é menor que 5. Se sim, será lançada uma exceção do tipo `ConstraintViolationException` que possui a descrição do erro. Não queremos mostrar a exceção para o usuário, portanto, o Spring MVC pode guardar o resultado (os erros de validação) em um objeto do tipo `BindingResult`, basta adicionar esse objeto também como parâmetro para o método:

```
@RequestMapping("adicionaDespesa")
public String adiciona(@Valid Despesa despesa, BindingResult result) {

    if (result.hasFieldErrors()) {
        return "despesa/formulario";
    }

    ....
}
```

Se quisermos mostrar a mensagem de erros no formulário, podemos utilizar uma tag especial do Spring MVC: `<form:errors />`. Essa tag possui um atributo *path* que indica a qual atributo da classe a mensagem está relacionada.

Veja a importação da taglib e também a adição da tag próxima ao campo no formulário:

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<body>
    <h3>Adiciona despesas</h3>
    <form:errors path="despesa.descricao" cssStyle="color:red"/>
    <br />
    <br />
    <form action="adicionaDespesa" method="post">
```



```
Descrição: <br />
<textarea rows="5" cols="100" name="descricao"></textarea><br />
<input type="submit" value="Adicionar" />
</form>
</body>
</html>
```

## 1.11 Personalizando mensagens do Bean Validation

Existem 2 maneiras para mostrar uma mensagem mais amigável para o usuário ao invés de exibirmos o texto que veio da exceção. A primeira e mais simples, é colocar logo após a anotação, A String de validação a mensagem:

```
private Long id;
@Size(min=5, message="A descrição deve conter no mínimo 5 caracteres")
private String descricao;
```

A outra maneira é isolar as mensagens em um arquivo ValidationMessages.properties. Esse arquivo deve ficar na pasta **src** e pode usar alguns coringas para formatação da mensagem:

```
despesa.descricao.msg=A descrição deve conter no mínimo {min} caracteres
```

E no atributo *message*, colocar o valor da chave:

```
private Long id;
@Size(min=5, message="{despesa.descricao.msg}")
private String descricao;
```

## 1.12 Atividade - Bean Validation

**1.12.1** Para configurar o framework Bean Validation é preciso copiar quatro jars. Primeiro, localize no **material de apoio** a pasta **jars-hibernate-validator**. Haverá quatro JARs, copie-os (CTRL+C) e cole-os (CTRL+V) dentro da /WebContent/WEB-INF/lib do projeto despesas-web.

**1.12.2** Abra a classe Despesa. Nela é preciso definir as regras de validação através das anotações do framework Bean validation. A atributo descricao deve ter pelo menos 5 caracteres:

```
private Long id;  
@Size(min=5, message="A descrição deve conter no mínimo 5 caracteres")  
private String descricao;
```

**1.12.2** Precisamos também colocar as anotações no método adiciona do controller. Abre a classe DespesasController e faça as alterações:

```
@RequestMapping("adicionaDespesa")  
public String adiciona(@Valid Despesa despesa, BindingResult result) {  
    if (result.hasFieldErrors()) {  
        return "despesa/formulario";  
    }  
  
    JdbcDespesaDao dao = new JdbcDespesaDao();  
    dao.adiciona(despesa);  
    return "despesa/adicionada";  
}
```

**1.12.3** Modifique também o formulario.jsp para exibir a mensagem de erro:

- Primeiro faça a importação da taglib:

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
```

E em seguida, insira a tag `<form:error />` logo abaixo do subtítulo:

```
<body>
  <h3>Adiciona despesas</h3>
  <form:errors path="despesa.descricao" cssStyle="color:red"/>
  <br />
  <br />
  <form action="adicionaDespesa" method="post">
  ...
```

**1.12.4** Reinicie o Tomcat, acesse o formulário e tente gravar uma despesa vazia:

<http://localhost:8080/despesas-web/novaDespesa>

**1.12.5** Opcional – Crie o arquivo **ValidationMessages.properties** e isole a mensagem de erro.

## 1.13 Listando despesas e enviando objetos para view

Agora que já conseguimos incluir despesas, podemos montar uma ação para exibir a lista de despesas cadastradas:

```
@RequestMapping("listaDespesas")
public String lista() {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    List<Despesa> despesas = dao.lista();
    return "despesa/lista";
}
```

Essa lista de despesas deverá ser disponibilizada para o JSP fazer sua exibição, ou seja, o Spring MVC não só recebe o nome da página JSP (despesa/lista) quando chama o método lista, o Spring MVC também recebe os dados para o JSP. Para dar suporte a essa estrutura, o Spring MVC encapsula os dados para exibição e nome da tela em uma classe especial chamada **ModelAndView**. Vamos modificar o método para retornar um objeto do tipo ModelAndView com os dados que precisamos:

```
@RequestMapping("listaDespesas")
public ModelAndView lista() {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    List<Despesa> despesas = dao.lista();

    ModelAndView mv = new ModelAndView("despesa/lista");
    mv.addObject("despesas", despesas);
    return mv;
}
```

O Spring também fornece outra opção que se assemelha bastante ao setAttribute do HttpServletRequest. Podemos receber um parâmetro no método do tipo **Model** e dessa forma não é necessário instanciar um objeto ModelAndView dentro do método e ele permanece retornando uma String:

```
@RequestMapping("listaDespesas")
public String lista(Model model) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    List<Despesa> despesas = dao.lista();
    model.addAttribute("despesas", despesas);
    return "despesa/lista";
}
```

## 1.14 Atividade – Exibindo despesas cadastradas

**1.14.1** Crie um novo método na classe `DespesasController` para retornar a lista de despesas:

```
@RequestMapping("listaDespesas")
public String lista(Model model) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    List<Despesa> despesas = dao.lista();
    model.addAttribute("despesas", despesas);
    return "despesa/lista";
}
```

**1.14.2** Para fazer a listagem, vamos precisar da **JSTL** (iremos fazer um `forEach`), portanto precisamos importá-la. Procure no **material de apoio** o diretório **jars-jstl**. Copie o conteúdo do diretório (CTRL+C) e cole (CTRL+V) na pasta `WebContent/WEB-INF/lib` do projeto `despesas-web`.

**1.14.3** Crie o arquivo **lista.jsp** no diretório `WebContent/WEB-INF/views/despesa` e preencha com o seguinte conteúdo (inclua a importação para as taglibs):

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<html>
<body>
    <a href="novaDespesa">Criar nova despesa</a>
    <br />
    <br />
    <table cellpadding="2" cellspacing="2">
        <tr>
            <th>Id</th>
            <th>Descrição</th>
            <th>Pago?</th>
            <th>Data pagamento</th>
        </tr>
    </table>

```

```
</tr>
<c:forEach items="${despesas}" var="despesa">
    <tr>
        <td>${despesa.id}</td>
        <td>${despesa.descricao}</td>
        <c:if test="${despesa.pago eq true }">
            <td>Pago</td>
        </c:if>
        <c:if test="${despesa.pago eq false }">
            <td>A Pagar</td>
        </c:if>
        <td><fmt:formatDate value="${despesa.dataPagamento.time}"
            pattern="dd/MM/yyyy" /></td>
    </tr>
</c:forEach>
</table>
</body>
</html>
```

**1.14.4** Reinicie o Tomcat e acesse: <http://localhost:8080/despesas-web/listaDespesas>

## 1.15 Redirecionando para outra ação

Assim como fizemos no Capítulo 9, vamos adicionar uma coluna para permitir remover e outra para alterar uma despesa:

```
<td><a href="removeDespesa?id=${despesa.id}">Remover</a>
```

Vamos ter uma também uma ação associada a esse link:

```
@RequestMapping("removeDespesa")
public String remove(Despesa despesa) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.remove(despesa);
    return " ??? ";
}
```

Depois da execução temos que voltar para lista, assim como fizemos com a lista de imóveis. Relembre a estratégia:

```
return new ListaImoveisServico().executa(request, response); //chamada direta dentro do Servlet
```

ou

```
return "mvc?servico=ListaImoveisServico"; //chamada para ou serviço via URL
```

Usando o Spring MVC também temos as duas opções, só precisamos modificar no retorno do método, adicionando as um prefixo:

Para fazer um redirecionamento no lado do servidor basta usar o prefixo *forward* no retorno:

```
@RequestMapping("removeDespesa")
public String remove(Despesa despesa) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.remove(despesa);
    return "forward:listaDespesas";
}
```

Para fazer um redirecionamento no lado do cliente usamos o prefixo *redirect*:

```
@RequestMapping("removeDespesa")
public String remove(Despesa despesa) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.remove(despesa);
    return "redirect:listaDespesas";
}
```

## 1.16 Atividade – Removendo e alterando despesas

**1.16.1** Adicione no arquivo lista.jsp uma coluna com um link que ao ser clicado invocará a Action para remover uma despesa:

```
<td><a href="removeDespesa?id=${despesa.id}">Remover</a>
```

**1.16.2** Adicione também em DespesasController a ação para executar a remoção. Use o prefixo *redirect* para exibir a lista:

```
@RequestMapping("removeDespesa")
public String remove(Despesa despesa) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.remove(despesa);
    return "redirect:listaDespesas";
}
```

**1.16.3** Acesse a lista de despesas e teste a remoção: <http://localhost:8080/despesas-web/listaDespesas>

**1.16.4** Agora, vamos adicionar o link para alterar uma despesa. Logo abaixo do link inserido no passo anterior, inclua:

```
<td><a href="alteraDespesa?id=${despesa.id}">Alterar</a>
```

**1.16.5** Inclua uma ação na classe DespesasController, que exibirá a despesa e seus respectivos campos para permitir a alteração:

```
@RequestMapping("mostraDespesa")
public String mostra(Long id, Model model) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    model.addAttribute("despesa", dao.buscaPorId(id));
    return "despesa/mostra";
}
```



**1.16.6** A ação do item anterior invoca o jsp "despesa/mostra", portando, vamos criar o arquivo mostra.jsp na pasta views/despesa. Modifique o conteúdo do arquivo para ficar como abaixo:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<body>
  <h3>Alterar despesa - ${despesa.id}</h3>
  <form action="alteraDespesa" method="post">
    <input type="hidden" name="id" value="${despesa.id}" />
    Descrição: <br />
    <textarea name="descricao" cols="100" rows="5">${despesa.descricao}</textarea>
    <br /> Pago?
    <input type="checkbox" name="pago" value="true" ${despesa.pago ? 'checked' : ''} />
    <br /> Data de pagamento: <br />
    <input type="text" name="dataPagamento"
      value="<fmt:formatDate value="${despesa.dataPagamento.time}" pattern="dd/MM/yyyy" /> " />
    <br />
    <br />
    <input type="submit" value="Alterar" />
  </form>
</body>
</html>
```

**1.16.7** Para o Spring MVC saber converter automaticamente a data no formato brasileiro para um Calendar é preciso usar a anotação **@DateTimeFormat**. Abra a classe Despesa e adicione a anotação acima do atributo dataPagamento:

```
@DateTimeFormat(pattern="dd/MM/yyyy")
private Calendar dataPagamento;
```

**1.16.8** O formulário da tela **mostra.jsp** chama a ação **alteraDespesa** (action="alteraDespesa"). Vamos então incluir essa ação na classe DespesasController:

```
@RequestMapping("alteraDespesa")
public String altera(Despesa despesa) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.altera(despesa);
    return "redirect:listaDespesas";
}
```

**1.16.9** Acesse a lista de despesas e teste a alteração: <http://localhost:8080/despesas-web/listaDespesas>

**1.16.10** Opcional - Você consegue colocar um calendário no campo Data Pagamento usando o JQuery?

## 1.17 AJAX

Queremos melhorar a usabilidade do nosso projeto fazendo com que o operador, ao clicar em “Pagar” o sistema automaticamente liquide a despesa e acrescente a data atual como data de pagamento.

A questão é que não queremos navegar para lugar algum ao clicarmos nesse link. Queremos permanecer na mesma tela, sem que nada aconteça, nem seja recarregado, ou seja, de alguma forma precisamos mandar a requisição para a ação, mas ainda assim precisamos manter a página do jeito que ela estava ao clicar no link. Podemos fazer isso através de uma técnica chamada AJAX, que significa Asynchronous Javascript and XML.

Para realizarmos uma requisição AJAX, precisamos utilizar Javascript. E no curso vamos utilizar o suporte que o JQuery nos fornece para trabalhar com AJAX.

Para fazermos uma requisição para um determinado endereço com o JQuery, basta definirmos qual método utilizaremos para enviar essa requisição (POST ou GET) e passar o endereço que queremos invocar:

```
$.get("paginaInvocada.jsp")
```

Sendo assim, vamos criar um link na nossa página que irá chamar função `quitarAgora()`:

```
<td><a href="#" onClick="quitarAgora(${despesa.id})">Pagar</a></td>
```

Depois, criamos a função correspondente que aciona a ação de quitar a despesa passando o `id`. Vamos utilizar o método `$.get` do JQuery:

```
<script type="text/javascript">  
    function quitarAgora(id) {  
        $.get("quitarDespesa?id="+ id);  
    }  
</script>
```

No entanto, a requisição que estamos fazendo não gerará resposta nenhuma e nós sempre retornamos uma String o resultado que determina qual JSP será exibido. Dessa vez, não exibiremos nem um JSP e nem invocaremos outra Action. O protocolo HTTP sempre retorna um código indicando qual é o estado dessa resposta: se foi executado com sucesso, se a página não foi encontrada, se algum erro aconteceu e assim por diante.

O código **200** indica que a execução ocorreu com sucesso. Para podermos devolver esse código para a página, podemos receber um objeto `HttpServletResponse` em qualquer ação do nosso Controller, logo, vamos utilizá-lo para passar essa resposta.

```
@RequestMapping("quitarDespesa")
public void quitarDespesa(Long id, HttpServletResponse response) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.quita(id);
    response.setStatus(200);
}
```

Para facilitar ainda mais nosso trabalho e não termos que lidar diretamente com o protocolo HTTP, o Spring MVC fornece uma anotação que responde automaticamente um código **200** para métodos que não tem retorno (void) caso nenhum erro tenha ocorrido. Sendo assim, vamos remover o parâmetro `HttpServletResponse` e incluir a anotação `@ResponseBody` no método:

```
@ResponseBody
@RequestMapping("quitarDespesa")
public void quitarDespesa(Long id, HttpServletResponse response) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.quita(id);
}
```

## 1.18 Quitando despesas com AJAX

No retorno do método, precisamos notificar o usuário que a despesa foi quitada. Existe uma maneira bem simples de resolver isso com JQuery. Podemos adicionar na função de chamada (`$.get`) outra função que é acionada automaticamente mediante um status **200**:

```
<script type="text/javascript">
    function quitarAgora(id) {
        $.get("quitaDespesa?id="+id, function(resposta) {
            alert("Despesa quitada na data de hoje!");
        });
    }
</script>
```

No exemplo acima, apresentamos um simples **Alert**, no entanto, o JQuery provê recursos que permitem alterar qualquer elemento do HTML, basta para isso, usar a função a seguir:

```
$("#idDoElementoHTML").html("Novo conteúdo HTML desse elemento");
```

**Nota:** O controlador do Spring MVC, ou seja o servlet no web.xml, foi configurado para receber todas as requisições incluindo essas que foram enviadas para receber o conteúdo de arquivos comuns como imagens, css ou scripts. Queremos que o controlador não atenda essas requisições que não são para ações. Para isso é preciso adicionar no arquivo spring-context.xml um mapeamento que informa ao Spring MVC que ele deve ignorar todo acesso a conteúdo estático:

```
<mvc:default-servlet-handler />
```

## 1.19 Atividade – Quitando uma despesa com AJAX

**1.19.1** Abra o arquivo spring-context.xml e acrescente:

```
<mvc:default-servlet-handler />
```

Na pasta WebContent crie uma nova pasta resources, vamos colocar nela tudo relativo a conteúdo estático do nosso projeto.

**1.19.2** Acrescente o JQuery ao projeto, procure na pasta **material de apoio** pela pasta **js**. Copie-a e cola na pasta resources criada no item anterior.

**1.19.3** Importe o JQuery na página **lista.jsp**. Logo após a tag HTML, acrescente o código abaixo:

```
<head>
  <script type="text/javascript" src="resources/js/jquery.js"></script>
</head>
```

Ainda no arquivo, modifique a coluna que exibe a despesa **paga ou a pagar**, para quando a despesa estiver em aberto, automaticamente virar um link que aciona a quitação:

```
<!-- codigo omitido -->
<c:if test="${despesa.pago eq true }">
  <td>Pago</td>
</c:if>
<c:if test="${despesa.pago eq false }">
  <td><a href="#" onClick="quitarAgora(${despesa.id})">Pagar</a></td>
</c:if>
<!-- codigo omitido -->
```

Vamos precisar que o JQuery modifique dinamicamente toda linha correspondente a despesa que foi quitada (status e data de pagamento). Para conseguir isso, vamos criar um jsp apenas com esse conteúdo. Crie o arquivo **quitada.jsp** na pasta /WEB-INF/view/despesa. Acrescente nele o seguinte conteúdo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<td>${despesa.id}</td>
<td>${despesa.descricao}</td>
<td>Pago</td>
<td><fmt:formatDate value="${despesa.dataPagamento.time}" pattern="dd/MM/yyyy" /></td>
<td><a href="removeDespesa?id=${despesa.id}">Remover</a>
<td><a href="mostraDespesa?id=${despesa.id}">Alterar</a>
```

Agora no **lista.jsp**, para que o JQuery consiga encontrar a linha que será modificada, vamos precisar dar um **id** para cada linha gerada no `forEach`. Localize a tag `<tr>` dentro do `forEach` e faça a seguinte modificação:

```
<!-- codigo omitido -->
<c:forEach items="${despesas}" var="despesa">
    <tr id="despesa_${despesa.id}" >
        <td>${despesa.id}</td>
    <!-- restante da página omitido -->
```

Crie logo abaixo da tag **<body>** a função JavaScript responsável pela quitação. A função receberá uma resposta da ação do `DespesaController` e modificará o conteúdo da linha quitada `<tr>`:

```
<body>
    <script type="text/javascript">
        function quitarAgora(id) {
            $.post("quitarDespesa", {'id' : id}, function(resposta) {
                //muda a linha da tabela pelo id
                $("#despesa_"+id).html(resposta);
            });
        }
    </script>
    <!-- restante da página omitido -->
```

**1.19.4** Acrescente o método para fazer a quitação da despesa e em seguida devolver a jsp de resposta para ser substituído na tag `<tr>` configurada acima:

```
@RequestMapping("quitarDespesa")
public String quitarDespesa(Long id, Model model) {
    JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.quita(id);
    model.addAttribute("despesa", dao.buscaPorId(id));
    return "despesa/quitada";
}
```

**1.19.5** Reinicie o Tomcat e acesse a lista: <http://localhost:8080/despesas-web/listaDespesas>. Teste a quitação com AJAX.



## 2. Spring MVC: Autenticação e Autorização

### 2.1 COOKIES

Um cookie é normalmente um par de strings guardado no cliente, assim como um mapa de strings. Por não ser seguros, normalmente são utilizados para manter informações como o nome do usuário logado por exemplo.

Quando um cookie é salvo no cliente, ele é enviado de volta ao servidor toda vez que o cliente efetuar uma nova requisição. Desta forma, o servidor consegue identificar aquele cliente sempre com os dados que o cookie enviar.

### 2.2 SESSÃO

Na sessão, diferentemente dos cookies, podemos salvar objetos. Dessa forma, podemos atrelar objetos de qualquer tipo a um cliente, não sendo limitada somente à strings e é independente de cliente.

A sessão nada mais é que um tempo que o usuário permanece ativo no sistema. A cada página visitada, o tempo de sessão é zerado. Quando o tempo ultrapassa um limite demarcado no arquivo web.xml, o cliente perde sua sessão. Para configurar esse tempo limite de 3 minutos para sessão, basta acrescentar a seguinte informação:

```
<session-config>  
  <session-timeout>3</session-timeout>  
</session-config>
```

## 2.3 Registrando o usuário na sessão

Para aumentar a segurança da nossa aplicação, vamos criar uma tela de login para os usuários. Se o usuário informado não estiver cadastrado, não conseguirá acessar o sistema, caso contrário, vamos armazenar seus dados na sessão e liberar as funcionalidades da aplicação.

O Spring MVC nos possibilita receber a sessão em qualquer método, só é preciso colocar o objeto **HttpSession** como parâmetro.

A sessão é parecida com um objeto do tipo `Map<String, Object>`, podemos guardar nela qualquer objeto que quisermos dando-lhes uma chave que é uma `String`. Portanto, para logarmos o usuário na aplicação poderíamos criar uma ação que recebe os dados do formulário de login e a sessão HTTP, guardando o usuário logado dentro da mesma. Na atividade a seguir, vamos testar a funcionalidade de validar usuários e armazenar seus dados em um sessão.

## 2.4 Atividade – Tela de login

**2.4.1** Crie a página **formulario-login.jsp** dentro de `WebContent/WEB-INF/views` com o conteúdo:

```
<html>
<body>
  <h2>Página de Login das Despesas</h2>
  <form action="efetuaLogin" method="post">
    Login:<input type="text" name="login" /><br />
    Senha:<input type="password" name="senha" /><br />
    <input type="submit" value="Entrar nas despesas" />
  </form>
</body>
</html>
```

```
</form>
</body>
</html>
```

**2.4.2** Crie a classe **LoginController** no pacote **br.com.igordev.despesas.controller**. Acrescente um método para exibir o formulario-login.jsp (**Não esqueça de anotar a classe com @Controller**):

```
@RequestMapping("loginForm")
public String loginForm() {
    return "formulario-login";
}
```

**2.4.3** Ainda na classe **LoginController** coloque o método que verifica a existência do usuário. Acrescente o método **efetuaLogin**:

```
@RequestMapping("efetuaLogin")
public String efetuaLogin(Usuario usuario, HttpSession session) {
    if (new JdbcUsuarioDao().existeUsuario(usuario)) {
        session.setAttribute("usuarioLogado", usuario);
        return "menu";
    }
    return "redirect:loginForm";
}
```

**2.4.4** Após o usuário se logar, ele será redirecionado para uma página que conterá links para as outras páginas do sistema e uma mensagem de boas vindas.

Crie a página **menu.jsp** em WebContent/WEB-INF/views com o código:

```
<html>
<body>
  <h2>Página inicial da Lista de Despesas</h2>
  <p>Bem vindo, ${usuarioLogado.login}</p>
  <a href="listaDespesas">Clique aqui</a> para acessar a lista de despesas
</body>
</html>
```

**2.4.5** Acesse o DBEaver e crie a tabela de usuários. Certifique-se que está logado no BD **imovel**:

```
use imovel;
create table usuarios (
  login VARCHAR(255),
  senha VARCHAR(255)
);
```

**2.4.6** Insira seu usuário para testar a tela de login. Digite o seguinte comando no DBEaver:

```
insert into usuarios (login, senha) values ('seu_nome', 'sua_senha');
```

**2.4.7** Teste a tela de login acessando: <http://localhost:8080/despesas-web/loginForm>.

## 2.5 Validar usuários com interceptadores

No Spring MVC, podemos utilizar o conceito dos Interceptadores, que funcionam como Filtros que aprendemos anteriormente, mas com algumas funcionalidades a mais que estão relacionadas ao framework.

Para criarmos um Interceptador basta criarmos uma classe que implemente a interface **org.springframework.web.servlet.HandlerInterceptor**. Ao implementar essa interface, precisamos implementar 3 métodos: **preHandle**, **postHandle** e **afterCompletion**.

Para nossa aplicação queremos verificar o acesso antes de executar uma ação, por isso vamos usar apenas o método **preHandle** da interface **HandlerInterceptor**. O Spring MVC oferece uma classe auxiliar (**HandlerInterceptorAdapter**) que já vem com uma implementação padrão para cada método da interface. Então para facilitar o trabalho vamos estender essa classe e sobrescrever apenas o método que é do nosso interesse.

O método **preHandle** recebe a requisição e a resposta, além do controlador que está sendo interceptado. O retorno é um booleano que indica se queremos continuar com a requisição ou não. Portanto, a classe **AutorizadorInterceptor** só deve devolver true se o usuário está logado. Caso o usuário não esteja autorizado vamos redirecionar para o formulário de login.

Para pegar o usuário logado é preciso acessar a sessão HTTP. O objeto **request** possui um método que devolve a sessão do usuário atual:

```
HttpSession session = request.getSession();
```

Existem duas ações na nossa aplicação que não necessitam de autorização. São as ações do **LoginController**, necessárias para permitir a autenticação do usuário. Além disso, vamos garantir também que a pasta de resources pode ser acessada mesmo sem login. Essa pasta possui as imagens, css e arquivos JavaScript. Portanto a classe **AutorizadorInterceptor** fica:

```
public class AutorizadorInterceptor extends HandlerInterceptorAdapter {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
        throws Exception {
        // liberar páginas e recursos que não necessitem de login
        String uri = request.getRequestURI();

        if (uri.endsWith("loginForm") || uri.endsWith("efetuaLogin") || uri.contains("resources")) {
            return true;
        }
    }
}
```

```
        if (request.getSession().getAttribute("usuarioLogado") != null) {  
            return true;  
        }  
  
        response.sendRedirect("loginForm");  
        return false;  
    }  
}
```

Perceba que essa classe não tem uma anotação como fizemos com as outras. Portanto, para usarmos um interceptor, precisaremos registrá-lo no arquivo **spring-context.xml**:

```
<mvc:interceptors>  
    <bean class="br.com.igordev.despesas.interceptor.AutorizadorInterceptor" />  
</mvc:interceptors>
```

## 2.6 Atividade – Usando interceptor para validar usuário

**2.6.1** Vamos criar o interceptor do nosso projeto. No pacote **br.com.igordev.despesas.interceptor**, criar a classe **AutorizadorInterceptor**. Estenda a classe **org.springframework.web.servlet.handler.HandlerInterceptorAdapter**. Utilize o CTRL + 3 [override] e sobrescreva o método **preHandle**.

```
public class AutorizadorInterceptor extends HandlerInterceptorAdapter {  
    @Override  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)  
        throws Exception {  
        // liberar páginas e recursos que não necessitem de login  
        String uri = request.getRequestURI();  
  
        if (uri.endsWith("loginForm") || uri.endsWith("efetuaLogin") || uri.contains("resources")) {  
            return true;  
        }  
        if (request.getSession().getAttribute("usuarioLogado") != null) {  
            return true;  
        }  
        response.sendRedirect("loginForm");  
        return false;  
    }  
}
```

**2.6.2** Registro o interceptor criado no item anterior no arquivo spring-context.xml. Insira o código abaixo dentro da tag <beans></beans>

```
<mvc:interceptors>
    <bean class="br.com.igordev.despesas.interceptor.AutorizadorInterceptor" />
</mvc:interceptors>
```

**2.6.3** Adicione um link para logout na tela inicial. Abre o arquivo menu.jsp e modifique para que fique conforme abaixo:

```
<html>
<body>
    <h2>Página inicial da Lista de Despesas</h2>
    <p>Bem vindo, ${usuarioLogado.login}</p>
    <a href="listaDespesas">Clique aqui</a> para acessar a lista de despesas
    <br />
    <br />
    <a href="logout">Sair do sistema</a>
</body>
</html>
```

**2.6.4** Crie a ação de logout na classe **LoginController**:

```
@RequestMapping("logout")
public String logout(HttpSession session) {
    session.invalidate();
    return "redirect:loginForm";
}
```

**2.6.5** Reinicie o Tomcat e faça o teste de login: <http://localhost:8080/despesas-web/listaDespesas>.

## 3. Sprint IoC

### 3.1 Removendo acoplamento com inversão de controle

No capítulo 10, vimos como fazer inversão de controle (IoC) no projeto **jimovel-web**. Na nossa classe **DespesasController** temos um problema semelhante. Veja quantas vezes instanciamos a classe **JdbcDespesaDao**:

```
@Controller
public class DespesasController {

    @RequestMapping("novaDespesa")
    public String form() {
        return "despesa/formulario";
    }

    @RequestMapping("adicionaDespesa")
    public String adiciona(@Valid Despesa despesa, BindingResult result) {

        if (result.hasFieldErrors()) {
            return "despesa/formulario";
        }

        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.adiciona(despesa);
        return "despesa/adicionada";
    }

    @RequestMapping("listaDespesas")
    public String lista(Model model) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        List<Despesa> despesas = dao.lista();
        model.addAttribute("despesas", despesas);
        return "despesa/lista";
    }

    @RequestMapping("removeDespesa")
    public String remove(Despesa despesa) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.remove(despesa);
        return "redirect:listaDespesas";
    }

    @RequestMapping("mostraDespesa")
    public String mostra(Long id, Model model) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        model.addAttribute("despesa", dao.buscaPorId(id));
        return "despesa/mostra";
    }

    @RequestMapping("alteraDespesa")
    public String altera(Despesa despesa) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.altera(despesa);
        return "redirect:listaDespesas";
    }

    // @ResponseBody
    @RequestMapping("quitarDespesa")
    public String quitarDespesa(Long id, Model model) {
        JdbcDespesaDao dao = new JdbcDespesaDao();
        dao.quita(id);
        model.addAttribute("despesa", dao.buscaPorId(id));
        return "despesa/quitada";
    }
}
```



Precisamos então, de alguma forma, eliminar esse volume excessivo de instanciação. Vamos então transformar o **dao** em um atributo da classe e instanciá-lo no construtor, em seguida, apenas utilizamos esse atributo nas ações:

```
private JdbcDespesaDao dao;

public DespesasController() {
    this.dao = new JdbcDespesaDao();
}

@RequestMapping("novaDespesa")
public String form() {
    return "despesa/formulario";
}

@RequestMapping("adicionaDespesa")
public String adiciona(@Valid Despesa despesa, BindingResult result) {

    if (result.hasFieldErrors()) {
        return "despesa/formulario";
    }
    //JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.adiciona(despesa);
    return "despesa/adicionada";
}

@RequestMapping("listaDespesas")
public String lista(Model model) {
    //JdbcDespesaDao dao = new JdbcDespesaDao();
    List<Despesa> despesas = dao.lista();
    model.addAttribute("despesas", despesas);
    return "despesa/lista";
}
//restante da classe ...
```

Melhoramos a classe pois diminuimos a quantidade de código para dar manutenção, porém, ainda temos a responsabilidade de criar o **dao** no construtor, gerando um forte acoplamento entre essas classes.

Podemos então fazer a inversão de controle e passar essa responsabilidade para cima, apenas recebendo a conexão no construtor:

```
private JdbcDespesaDao dao;

public DespesasController(JdbcDespesaDao dao) {
    this.dao = dao;
}
```

Agora, vamos passar para a classe JdbcDespesaDao:

```
public class JdbcDespesaDao {

    private final Connection connection;

    public JdbcDespesaDao() {
        try {
            this.connection = new ConnectionFactory().getConnection();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    //restante do código ...
}
```

No construtor do **JdbcDespesasDao**, vemos também uma situação de dependência, ou seja, o JdbcDespesaDao depende de uma connection que está sendo fornecida pelo **ConnectionFactory**. Vamos mais uma vez repassar essa tarefa, apenas recebendo a connection no construtor:

```
private final Connection connection;

public JdbcDespesaDao(Connection connection) {
    this.connection = connection;
}
//restante do código ...
```

Mas e agora, quem irá criar a conexão e o dao?

### 3.2 Contexto de Injeção de Dependência (CDI)

O padrão de projetos Dependency Injection (DI) (Injeção de dependências), procura resolver esses problemas. A ideia é que a classe não mais resolva as suas dependências por conta própria mas apenas declare que depende de alguma outra classe.

Repare que com **@Controller** já definimos que a nossa classe faz parte do Spring MVC, mas a anotação vai além disso. Também definimos que queremos que o Spring controle o objeto. Spring é no fundo um container que dá new para nós e também sabe resolver e ligar as dependências. Por isso, o Spring também é chamado **Container IoC** (Inversion of Control) ou **Container DI**.

### 3.3 Contexto de Injeção de Dependência (CDI)

Para receber o DAO em nosso controlador, usaremos a anotação **@Autowired** (amarrar) acima do construtor. Isso indica ao Spring que ele precisa resolver e injetar a dependência:

```
@Controller
public class DespesasController {

    private JdbcDespesaDao dao;

    @Autowired
    public DespesasController(JdbcDespesaDao dao) {
        this.dao = dao;
    }
}
```

Para o Spring conseguir criar o `JdbcDespesaDao` vamos declarar a classe como componente. O Spring possui a anotação **@Repository** que deve ser utilizada nas classes como o **DAO**. Além disso, vamos também amarrar a conexão com **@Autowired**:

```
@Repository
public class JdbcDespesaDao {

    private final Connection connection;

    @Autowired
    public JdbcDespesaDao(Connection connection) {
        this.connection = connection;
    }
    //restante do código ...
}
```

Utilizando a anotação **@Autowired** no construtor, avisamos ao Spring que a classe `JdbcDespesaDao` precisa de uma conexão, porém, o Container não faz ideia com qual banco queremos nos conectar. Vamos precisar declarar um **bean** no XML do `spring-context.xml` passando as informações da conexão e do banco. No JavaEE esse bean é conhecido como **DataSource**.

```
<bean id="mysqlDataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
      value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost/imovel" />
    <property name="username" value="root" />
    <property name="password" value="" />
  </bean>
```

Com a **mysqlDataSource** definida, podemos injetar ela na `JdbcDespesaDao` para recuperar a conexão JDBC:

```
@Repository
public class JdbcDespesaDao {

    private final Connection connection;

    @Autowired
    public JdbcDespesaDao(DataSource dataSource) {
        this.connection = DataSource.getConnection();
    }
    //restante do código ...
}
```

Dessa forma resolvemos todas as dependências: DespesasController → JdbcDespesaDao → DataSource.

### 3.4 Atividade – IoC com Spring

**3.4.1** Vamos instalar os **jars** do DataSource no projeto. Procure em **material de apoio** a pasta **jars-datasource**. Copie os arquivos commons-dbcp-x.x.jar e commons-pool-x.x.jar para a pasta /WebContent/WEB-INF/lib do projeto **despesas-web**.

**3.4.2** Faça a declaração do **bean** do datasource no xml do spring (spring-context.xml):

```
<bean id="mysqlDataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
        value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost/imovel" />
    <property name="username" value="root" />
    <property name="password" value="" />
</bean>
```

**3.4.3** Abra o arquivo `JdbcDespesasDao` e modifique o construtor para receber o `DataSource` (importe o `datasource` de **javax.sql**). Não esqueça da anotação **@Repository** no início da classe:

```
@Repository
public class JdbcDespesaDao {

    private final Connection connection;

    @Autowired
    public JdbcDespesaDao(DataSource dataSource) {
        try {
            this.connection = dataSource.getConnection();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    //restante do código ...
}
```

**3.4.4** Abra agora a classe `DespesasController`. Coloque a dependência para o `JdbcDespesaDao` no construtor da classe:

```
@Controller
public class DespesasController {

    private JdbcDespesaDao dao;

    @Autowired
    public DespesasController(JdbcDespesaDao dao) {
        this.dao = dao;
    }

    //restante do código ...
}
```

**3.4.5** Ainda na classe `DespesasController`, remova todas as linhas que instanciam a classe `JdbcDespesasDao`, já que essa operação não é mais necessária:

```
@RequestMapping("adicionaDespesa")
public String adiciona(@Valid Despesa despesa, BindingResult result) {

    if (result.hasFieldErrors()) {
        return "despesa/formulario";
    }

    //JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.adiciona(despesa);
    return "despesa/adicionada";
}

@RequestMapping("listaDespesas")
public String lista(Model model) {
    //JdbcDespesaDao dao = new JdbcDespesaDao();
    List<Despesa> despesas = dao.lista();
    model.addAttribute("despesas", despesas);
    return "despesa/lista";
}

@RequestMapping("removeDespesa")
public String remove(Despesa despesa) {
    //JdbcDespesaDao dao = new JdbcDespesaDao();
    dao.remove(despesa);
    return "redirect:listaDespesas";
}
```

**3.4.6** Reinicie o Tomcat e acesse a aplicação: <http://localhost:8080/despesas-web/loginForm>. Tudo deve estar funcionando.

## 3.5 Aplicando CSS à aplicação

Para aprimorar o visual, vamos aplicar **CSS** nas nossas páginas. CSS é um acrônimo para *Cascading Style Sheets*, que podemos traduzir para **Folhas de Estilo em Cascata**. Os estilos definem o layout, por exemplo, a cor, o tamanho, o posicionamento e são declarados para um determinado elemento HTML. O CSS altera as propriedades visuais daquele elemento e, por cascata, todos os seus elementos filhos. Em geral, o HTML é usado para estruturar conteúdo da página (tabelas, parágrafos, div, etc.) e o CSS formata e define a aparência.

A sintaxe do CSS tem uma estrutura simples: é uma declaração de propriedades e valores separados por um sinal de dois pontos(:), e cada propriedade é separada por um sinal de ponto e vírgula(;), da seguinte maneira:

```
background-color: yellow;  
color: blue;
```

Como estamos declarando as propriedades visuais de um elemento em outro lugar do nosso documento, precisamos indicar de alguma maneira a qual elemento nos referimos. Fazemos isso utilizando um **seletor CSS**.

No exemplo a seguir, usaremos o seletor `p`, que alterará todos os parágrafos do documento:

```
p {  
    color: blue;  
    background-color: yellow;  
}
```

Existem várias lugares que podemos declarar os estilos, mas o mais comum é usar um arquivo externo, com a extensão **.css**. Para que seja possível declarar nosso CSS em um arquivo à parte, precisamos indicar em nosso documento HTML uma ligação entre ele e a folha de estilo.

A indicação de uso de uma folha de estilos externa deve ser feita dentro da tag **<head>** do nosso documento HTML.



### 3.6 Atividade - Aplicando CSS à aplicação

**3.6.1** No projeto **despesas-web**, crie o diretório **css** dentro da pasta **WebContent/resources**. Procure no **material de apoio** a pasta **css** e copie o arquivo **imovel.css** para a pasta criada (/WebContent/resources/css).

**3.6.2** Abra a view **lista.jsp** e dentro da tag <head> e crie o link para a folha de estilo:

```
<head>
  <link type="text/css" href="resources/css/imovel.css" rel="stylesheet" />
  <script type="text/javascript" src="resources/js/jquery.js"></script>
</head>
```

**3.6.3** Reinicie o Tomcat e acesse a aplicação: <http://localhost:8080/despesas-web/loginForm>. Faça login e clique para exibir a lista.

**3.6.4** Opcional – Aplique a página de estilo nas outras páginas do projeto.

## 4. Spring com JPA

### 4.1 Mapeamento objeto relacional

Até o momento, utilizamos a API do JDBC para fazer a integração da nossa aplicação com o banco de dados. Apesar de estar funcionando, é muito trabalhoso termos que lidar com a construção de scripts para serem executados no banco de dados. Outro problema é precisarmos constantemente transformar os registro do banco de dados em objetos e também o contrário, ou seja, desmembrar os atributos da classe para fazer operações de insert, update e delete.

### 4.2 JPA

JPA é uma especificação que surgiu com o intuito de fazer esse mapeamento objeto relacional (ORM) de forma automática, transformando objeto em dados e dados de volta par objeto. O Hibernate é a implementação mais famosa do JPA (na verdade, o Hibernate surgiu antes do JPA e serviu como base para a criação da especificação) e podemos integrá-lo perfeitamente ao Spring MVC.

### 4.3 Mapeando uma classe com JPA

Para que o Hibernate entenda nossa classe **Despesa** como um tabela do banco de dados, precisamos inserir nela algumas anotações:

```
@Entity
public class Despesa {

    @Id
    @GeneratedValue
    private Long id;
    @Size(min=5, message="{despesa.descricao.msg}" )
    private String descricao;
    private boolean pago;
    @DateTimeFormat(pattern="dd/MM/yyyy")
    @Temporal(TemporalType.DATE)
    private Calendar dataPagamento;
    //restante do código ...
}
```

**@Entity:** indica que a classe é uma entidade do BD (tabela)

**@Id/@GeneratedValue:** O campo que recebe essa anotação é a **Primary Key** da tabela. A segunda anotação (@GeneratedValue) indica que o valor será gerado automaticamente (auto incremental)

**@Temporal(TemporalType.DATE):** usado para um campo data. Indica que apenas a data (e não a hora) será armazenada no BD.

Fazendo dessa forma, o JPA irá procurar no banco uma tabela com o nome **despesa** e as colunas terão o mesmo nome dos atributos da classe. Caso seja necessário utilizar um nome diferente, podemos anotar o atributo com:

```
@Column(name="outro_nome_para_coluna")
```

Existe uma anotação semelhante para mudar o nome da tabela:

```
@Table(name="outro_nome_para_tabela")
```

Para utilizar o Hibernate no projeto, além da importação dos jars para a pasta WEB-INF/lib, precisamos do seu arquivo de configuração: **persistence.xml** que fica do lado do Model:

```
<persistence-unit name="despesas">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <!-- classes que representam as entidades do projeto -->
  <class>br.com.igordev.despesas.modelo.Despesa</class>
  <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
    <property name="hibernate.show_sql" value="true"/>
    <property name="hibernate.format_sql" value="true"/>
    <property name="hibernate.hbm2ddl.auto" value="update"/>
  </properties>
</persistence-unit>
```

As propriedades mais importantes são:

**hibernate.show\_sql (true/false):** Se configurada para true, exibe no console o código SQL gerado para executar o comando;

**hibernate.format\_sql (true/false):** Se configurada para true, faz a indentação do código SQL de forma que ele fique mais legível;

**hibernate.hbm2ddl.auto:** Se configurada para update criar a tabela se essa não existir. Ele também modifica a estrutura da tabela baseando-se nas alterações da classe.

**<persistence-unit name="despesas">** representa nossa unidade de persistência, também conhecida como **EntityManager**.

## 4.4 Integração do Hibernate com o Spring

Dentre as diversas características do Spring uma das que mais chama a atenção é a integração nativa com diversas tecnologias importantes do mercado. Podemos citar o Hibernate e o JPA. Com o Hibernate configurado, precisamos avisar ao Spring como iniciar

nossa unidade de persistência, para isso, adicionamos outro **bean** no arquivo spring-context.xml:

```
<!-- gerenciamento de jpa pelo spring -->
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="mysqlDataSource" />
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
</bean>
```

Para dar também ao Spring o poder de gerenciar as transações (commit/rollback) precisaremos de outro **bean**:

```
<!-- gerenciamento de transações pelo spring -->
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Pronto, configuração concluída. Como o Spring já sabe qual o banco, usuário e senha, ele agora também consegue subir o EntityManager e controlar as transações do **DAO**. Mas e o DAO? Como fica.

## 4.5 DAO do JPA e Injeção do EntityManager

O DAO do JPA depende do EntityManager, porém, como já temos o **bean** responsável por localizar e criar a unidade de persistência (persistence-unit), podemos simplesmente

injetar o EntityManager no DAO. Depois disso, basta chamar os métodos correspondentes para cada tarefa. A forma de injetar o EntityManager não é com o @Autowired do Spring, devemos utilizar ao invés disso **@PersistenceContext**:

```
@Repository
public class JpaDespesaDao {

    @PersistenceContext
    EntityManager manager;

    @Override
    public Despesa buscaPorId(Long id) {
        return manager.find(Despesa.class, id);
    }

    @Override
    public List<Despesa> lista() {
        return manager.createQuery("select d from despesa d", Despesa.class).getResultList();
    }

    @Override
    public void adiciona(Despesa despesa) {
        manager.persist(despesa);
    }

    @Override
    public void altera(Despesa despesa) {
        manager.merge(despesa);
    }

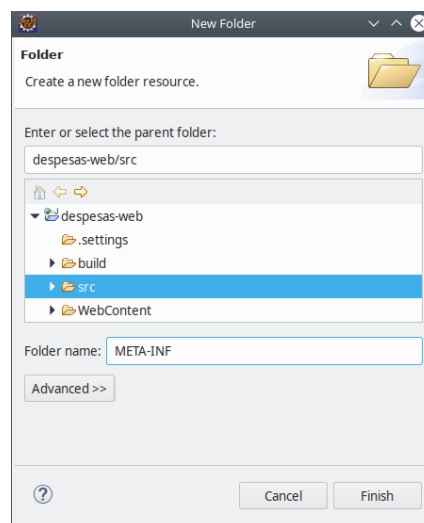
    @Override
    public void remove(Despesa despesa) {
        Despesa despesaRemover = buscaPorId(despesa.getId());
        manager.remove(despesaRemover);
    }

    @Override
    public void quita(Long id) {
        Despesa despesa = buscaPorId(id);
        despesa.setPago(true);
        despesa.setDataPagamento(Calendar.getInstance());
        manager.merge(despesa);
    }
}
```

## 4.6 Atividade – Configurando o projeto para trabalhar com JPA

**4.6.1** Inicialmente, vamos integrar o Hibernate ao projeto. Localiza no **material de apoio** a pasta **jars-jpa/hibernate** e copie todo seu conteúdo para **WebContent/WEB-INF/lib** do projeto **despesas-web**.

**4.6.2** Clique com direito na pasta **src > New > Folder** e crie o diretório META-INF:



**4.6.3** Volte no **material de apoio/jars-jpa** e copie o arquivo **persistence.xml** para a pasta criada no item anterior.

**4.6.4** Abra o arquivo **persistence.xml** e adicione a unidade de persistência:

```
<persistence-unit name="despesas">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <class>br.com.igordev.despesas.modelo.Despesa</class>
  <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
    <property name="hibernate.show_sql" value="true"/>
    <property name="hibernate.format_sql" value="true"/>
    <property name="hibernate.hbm2ddl.auto" value="update"/>
  </properties>
</persistence-unit>
```

**4.6.5** Agora, vamos fazer a integração do Spring com o Hibernate. Acesse em **material de apoio** a pasta **jars-jpa/spring4**. Copie os arquivos **aopalliance.jar**, **spring-orm-4.x.x.RELEASE.jar**, **spring-tx-4.x.x.RELEASE.jar** para a pasta **WebContent/WEB-INF/lib** do projeto.

**4.6.6** Adicione os **beans** do "entityManagerFactory" e do "transactionManager" dentro do arquivo **spring-context.xml** dentro da tag **<beans></beans>**:

```
<!-- gerenciamento de jpa pelo spring -->
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="mysqlDataSource" />
  <property name="jpaVendorAdapter">
    <bean
      class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
  </property>
</bean>

<!-- gerenciamento de transações pelo spring -->
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory"
    ref="entityManagerFactory" />
</bean>
```

Dessa forma, o Spring cria o EntityManager com o EntityManagerFactory e gerencia as transações.

**4.6.7** Vamos agora fazer as anotações na nossa classe **Despesa.class** para que o Hibernate entenda ela com uma entidade do BD. Abra a classe e faça as seguintes alterações (faça as importações necessárias do pacote **javax** – CTRL + SHIFT + O):



```
@Entity(name="despesa")
public class Despesa {

    @Id
    @GeneratedValue
    private Long id;
    @Size(min=5, message="{despesa.descricao.msg}" )
    private String descricao;
    private boolean pago;
    @DateTimeFormat(pattern="dd/MM/yyyy")
    @Temporal(TemporalType.DATE)
    private Calendar dataPagamento;
    //restante do código ...
}
```

**4.6.8** Agora, nosso projeto terá um segundo DAO para gerenciar a classe Despesa. Vamos então criar uma interface e obrigar os 2 a implementá-la. Crie a interface **DespesaDao** no pacote **br.com.igordev.despesas.dao**. Acrescente nela os métodos necessários para as operações:

```
public interface DespesaDao {
    Despesa buscaPorId(Long id);
    List<Despesa> lista();
    void adiciona(Despesa despesa);
    void altera(Despesa despesa);
    void remove(Despesa despesa);
    void quita(Long id);
}
```

Utilize CTRL + SHIT + O para fazer as importações.

**4.6.8** Faça com que a classe **JdbcDespesaDao** implemente a interface **DespesaDao**. Nenhum erro deve ocorrer:

```
@Repository
public class JdbcDespesaDao implements DespesaDao {

    private final Connection connection;

    @Autowired
    public JdbcDespesaDao(DataSource dataSource) {
        try {
            this.connection = dataSource.getConnection();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    //restante do código ...
}
```

**4.6.9** Vamos agora criar a implementação para o JPA. Crie a classe **JpaDespesaDao.java** no pacote **br.com.igordev.despesas.dao**. Não esqueça de anotar a classe com **@Repository** para o Spring saber que é um DAO e também de implementar a interface **DespesaDao**.

```
@Repository
public class JpaDespesaDao implements DespesaDao {

    @PersistenceContext
    EntityManager manager;

    @Override
    public Despesa buscaPorId(Long id) {
        return manager.find(Despesa.class, id);
    }

    @Override
    public List<Despesa> lista() {
        return manager.createQuery("select d from despesa d", Despesa.class).getResultList();
    }

    @Override
    public void adiciona(Despesa despesa) {
        manager.persist(despesa);
    }

    @Override
    public void altera(Despesa despesa) {
        manager.merge(despesa);
    }

    @Override
    public void remove(Despesa despesa) {
        Despesa despesaRemover = buscaPorId(despesa.getId());
        manager.remove(despesaRemover);
    }

    @Override
    public void quita(Long id) {
        Despesa despesa = buscaPorId(id);
        despesa.setPago(true);
        despesa.setDataPagamento(Calendar.getInstance());
        manager.merge(despesa);
    }
}
```

**4.6.10** Agora, devemos modificar nosso **DespesasController** para que ele consiga usar qualquer uma das implementações. Para isso, mude a declaração de classe **JdbcDespesaDao** para interface **DespesaDao**:

```
@Controller
public class DespesasController {

    DespesaDao dao;
    //JdbcDespesaDao dao;
```

**4.6.11** Vamos injetar a implementação do DAO diretamente na declaração que fizemos, para isso adicione a anotação **@Autowired** logo acima da declaração da variável. Outro fato é que, como temos duas implementações para a interface, para que o Spring saiba qual usar, vamos adicionar também a anotação **@Qualifier("jpaDespesaDao")**. No final, ficará assim:

```
@Controller
public class DespesasController {

    @Autowired
    @Qualifier("jpaDespesaDao")
    DespesaDao dao;
    // restante do código...
```

**4.6.12** Finalmente, para que o Spring saiba quais ações do controller precisam de uma transação, podemos anotar o método com **@Transactional**. Como todos os métodos da classe **DespesasController** vão precisar de transação, podemos colocar essa anotação direto na classe (importe de `javax.transactional`):

```
@Transactional
@Controller
public class DespesasController {

    @Autowired
    @Qualifier("jpaDespesaDao")
    DespesaDao dao;
    //restante do código...
```

**4.6.13** Inclua também uma nova tag no **spring-context.xml** (dentro da tag `<beans>...</beans>`) para avisar ao Spring MVC que os contextos transacionais estão anotados na classe:

```
<tx:annotation-driven />
```

**4.6.14** Reinicie o Tomcat. Tudo ainda deve estar funcionando.

**4.6.15** [Opcional] – Apague a tabela do banco de dados. Veja se o Hibernate a criará novamente.

## 5. Apêndice A – Convertendo o projeto em MAVEN

### 5.1 MAVEN PROJECT

Durante a realização das atividades do curso, por muitas vezes, tivemos que copiar as implementações para as especificações (**jars**) para a pasta lib do projeto:

- Os jars do Hibernate;
- Os jars do Spring MVC;
- Os jars do JSTL;
- Os jars do MySQL;
- Etc...

Uma das funções do Maven é justamente livrar-nos dessa tarefa facilitando o processo de construção do projeto. Esse processo de construção baseia-se no conceito de “**Project Object Model**” que é gerenciado pelo arquivo **pom.xml** que deverá ser inserido na aplicação.

Iremos utilizar o conjunto de tags do **The Basics** do arquivo **pom.xml**. Essas tags permitem-nos dizer, quais as **dependências** do nosso projeto, veja um exemplo:

```
<dependencies>
  <!-- DEPENDÊNCIAS DO SPRING -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
  </dependency>
  ...
</dependencies>
```

```
<!-- DEPENDÊNCIAS HIBERNATE -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.0.1.Final</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.2.0.Final</version>
</dependency>
...

<!-- OUTRAS DEPENDÊNCIAS -->
<dependency>
  <groupId>javax.transaction</groupId>
  <artifactId>javax.transaction-api</artifactId>
  <version>1.2</version>
</dependency>

<!-- MYSQL -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.6</version>
</dependency>
</dependencies>
```

Observe os trechos destacados em amarelo. A tag **<version></version>** permite especificar qual a versão do *artefato* iremos utilizar no projeto. Podemos isolar a versão do artefato utilizando uma outra tag: **<properties></properties>**, facilitando assim o processo de atualização das dependências. Veja um exemplo:

```
<properties>
  <spring.version>4.2.1.RELEASE</spring.version>
</properties>

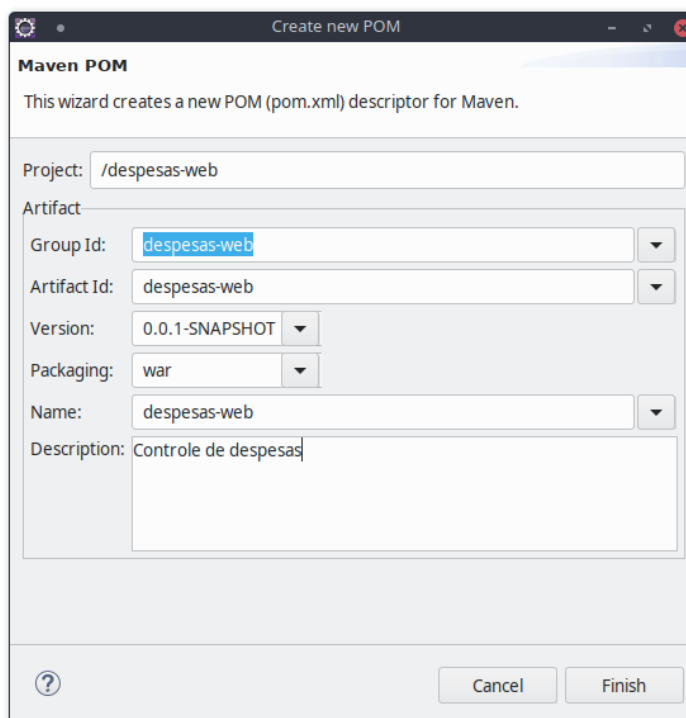
<dependencies>

  <!-- DEPENDÊNCIAS DO SPRING -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
```

Podemos facilmente converter um projeto WEB em um Maven Project utilizando o eclipse. Veremos na atividade a seguir, como converter o projeto **despesas-web** em **Maven Project**.

## 5.2 Atividade – Convertendo o projeto em Maven Project

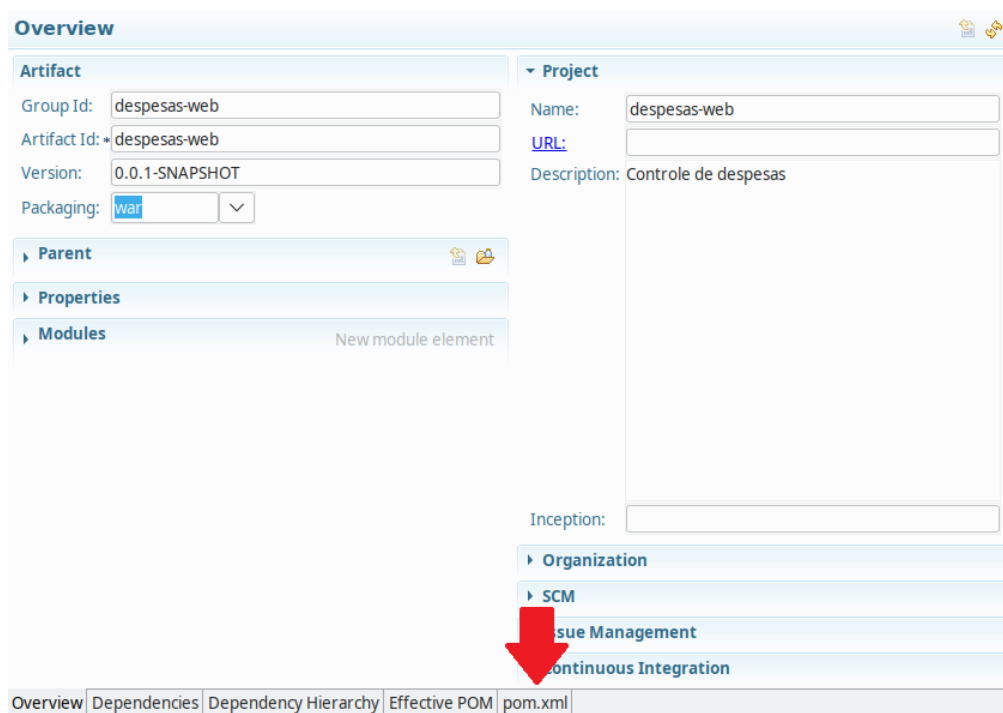
**5.2.1** No eclipse, clique com o direito sobre o projeto **despesas-web** > **Configure** > **Convert to Maven Project** (opcionalmente, preencha os campos **Name** e **Description**). Clique em **Finish**:



The 'Create new POM' dialog box in Eclipse IDE. It contains the following fields and options:

- Project: /despesas-web
- Artifact:
  - Group Id: despesas-web
  - Artifact Id: despesas-web
  - Version: 0.0.1-SNAPSHOT
  - Packaging: war
  - Name: despesas-web
  - Description: Controle de despesas
- Buttons: Cancel, Finish

**5.2.2** Automaticamente, o eclipse irá exibir o arquivo **pom.xml**. Mude a perspectiva para visualizar o conteúdo XML do arquivo:



The Eclipse IDE Overview perspective showing the Maven POM configuration for 'despesas-web'. The 'Artifact' section on the left lists the project details. The 'Project' section on the right shows the project name and description. A red arrow points to the 'pom.xml' tab in the bottom editor area.

**Artifact**

- Group Id: despesas-web
- Artifact Id: despesas-web
- Version: 0.0.1-SNAPSHOT
- Packaging: war

**Project**

- Name: despesas-web
- URL:
- Description: Controle de despesas

**Parent**

**Properties**

**Modules** New module element

**Inception:**

**Organization**

**SCM**

**Issue Management**

**Continuous Integration**

**Overview** Dependencies Dependency Hierarchy Effective POM **pom.xml**



**5.2.3** Repare que o arquivo já está pré-configurado com a versão do JDK e também os plug-ins de compilação do **Maven**. Modifique o arquivo adicionando as dependências do projeto logo abaixo da tag **</build>**:

```
<project xmlns...>
... código dos plugins
</build>

<properties>
  <spring.version>4.2.1.RELEASE</spring.version>
</properties>

<dependencies>

  <!-- DEPENDÊNCIAS DO SPRING -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring.version}</version>
  </dependency>

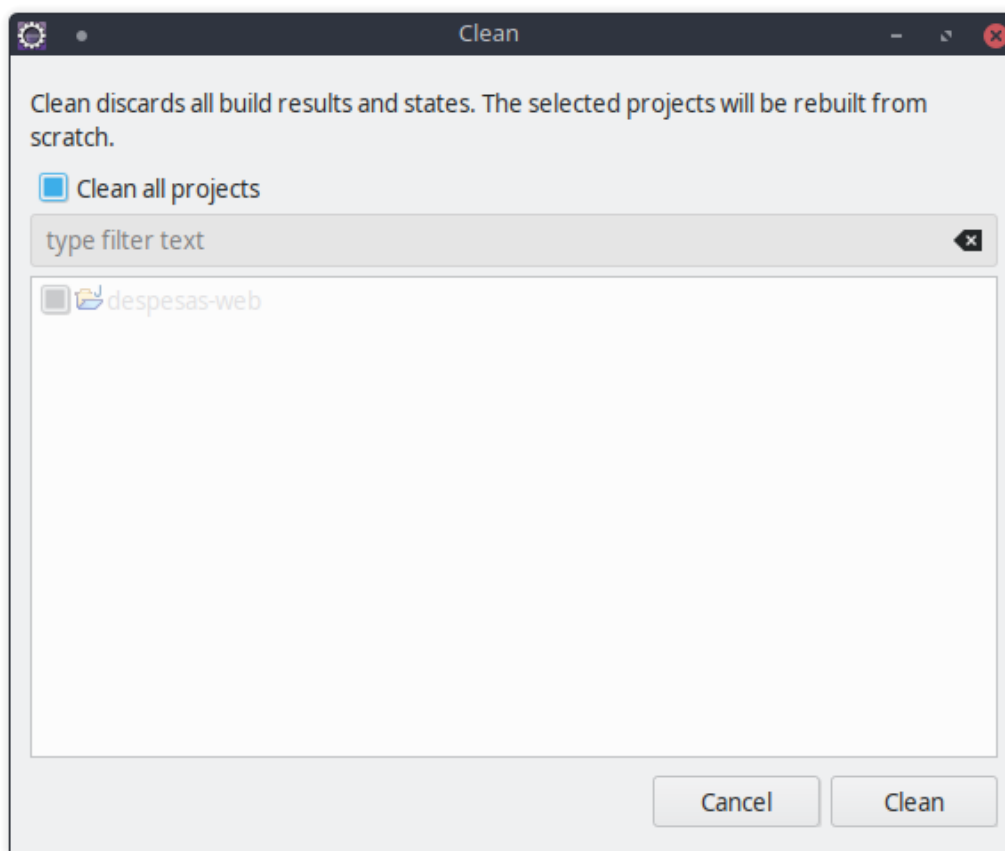
  <!-- DEPENDÊNCIAS HIBERNATE -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>4.0.1.Final</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate.common</groupId>
    <artifactId>hibernate-commons-annotations</artifactId>
    <version>4.0.1.Final</version>
    <classifier>tests</classifier>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>4.0.1.Final</version>
  </dependency>

  <!-- OUTRAS DEPENDÊNCIAS -->
  <dependency>
    <groupId>javax.transaction</groupId>
    <artifactId>javax.transaction-api</artifactId>
```

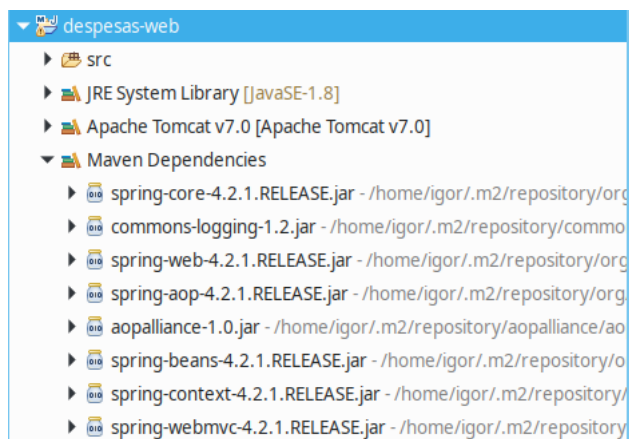
```
<version>1.2</version>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.2.2</version>
</dependency>

<!-- MYSQL -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.6</version>
</dependency>
</dependencies>
</project>
```

**5.2.4** Salve o arquivo **pom.xml** e em seguida limpe o projeto acessando o menu **Project** > **Clean...** Clique no botão **Clean**:



**5.2.5** Verifique se as dependências foram baixadas:



**5.2.6** Agora você pode apagar a pasta **lib** de **WEB-INF**. Reinicie o Tomcat e teste novamente o projeto. Tudo ainda deve estar funcionando.

[Criar nova despesa](#)

Id	Descrição	Pago?	Data pagamento		
1	Pagamento de funcionários: R\$ 130.000,00	Pago	16/08/2018	<a href="#">Remover</a>	<a href="#">Alterar</a>
2	Energia elétrica: R\$ 450,00	<a href="#">Pagar</a>		<a href="#">Remover</a>	<a href="#">Alterar</a>
3	Pintura imóveis: R\$ 6700.00	Pago	19/08/2018	<a href="#">Remover</a>	<a href="#">Alterar</a>