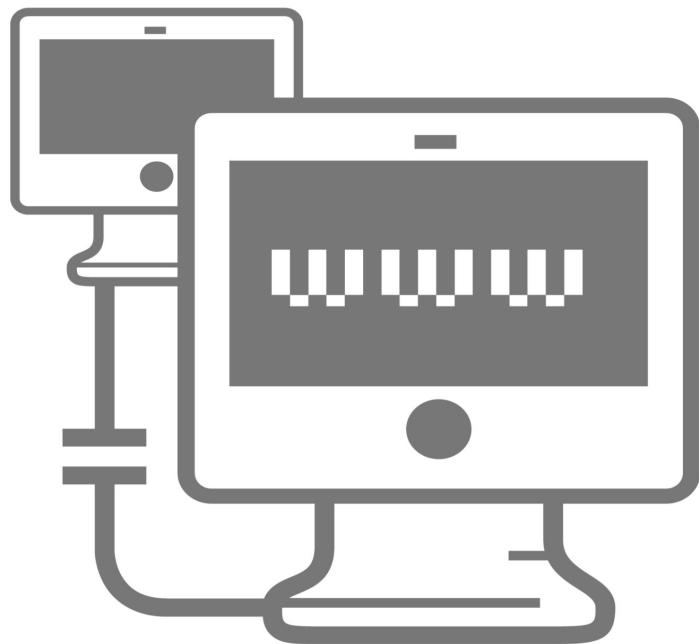


Java para Desenvolvimento Web

Curso FJ-21



Sumário

1 Enfrentando o Java na Web	1
1.1 O grande mercado do Java na Web	1
1.2 Livros e sites interessantes	2
2 Bancos de dados e JDBC	3
2.1 Por que usar um banco de dados?	3
2.2 Persistindo através de Sockets?	4
2.3 A conexão em Java	4
2.4 Fábrica de Conexões	7
2.5 Design Patterns	8
2.6 Exercícios: ConnectionFactory	8
2.7 A tabela Contato	12
2.8 Javabeans	12
2.9 Inserindo dados no banco	14
2.10 DAO - Data Access Object	17
2.11 Exercícios: Javabeans e ContatoDao	19
2.12 Fazendo pesquisas no banco de dados	22
2.13 Exercícios: Listagem	23
2.14 Um pouco mais...	24
2.15 Exercícios opcionais	25
2.16 Outros métodos para o seu DAO	25
2.17 Exercícios opcionais - Alterar e remover	26
3 O que é Java EE?	28
3.1 Como o Java EE pode te ajudar a enfrentar problemas	28
3.2 Algumas especificações do Java EE	29
3.3 Servidor de Aplicação	30
3.4 Servlet Container	31
3.5 Exercícios: Preparando o Tomcat	32
3.6 Preparando o Tomcat em casa	33

Sumário	Caelum
3.7 Outra opção: Jetty	33
3.8 Integrando o Tomcat no Eclipse	34
3.9 O plugin WTP	34
3.10 Exercícios: Configurando o Tomcat no Eclipse	34
4 Novo projeto Web usando Eclipse	39
4.1 Novo projeto	39
4.2 Exercícios: Novo projeto web	39
4.3 Análise do resultado final	42
4.4 Criando nossas páginas e HTML Básico	46
4.5 Exercícios: primeira página	46
4.6 Para saber mais: configurando o Tomcat sem o plugin	47
4.7 Algumas tags HTML	47
5 Servlets	49
5.1 Páginas dinâmicas	49
5.2 Servlets	50
5.3 Mapeando uma servlet no web.xml	52
5.4 A estrutura de diretórios	53
5.5 Exercícios: Primeira Servlet	53
5.6 Erros comuns	55
5.7 Facilidades das Servlets 3.0	56
5.8 Para saber mais: Web Servlet e InitParam Annotation	57
5.9 Enviando parâmetros na requisição	58
5.10 Pegando os parâmetros da requisição	59
5.11 Exercícios: Criando funcionalidade para gravar contatos	61
5.12 GET, POST e métodos HTTP	65
5.13 Tratando exceções dentro da Servlet	65
5.14 Exercício: Tratando exceções e códigos HTTP	66
5.15 Init e Destroy	68
5.16 Uma única instância de cada Servlet	69
5.17 Exercícios opcionais	70
5.18 Discussão: Criando páginas dentro de uma servlet	70
6 JavaServer Pages	71
6.1 Colocando o HTML no seu devido lugar	71
6.2 Exercícios: Primeiro JSP	72
6.3 Listando os contatos com Scriptlet	74
6.4 Exercícios opcionais: Lista de contatos com scriptlet	74

6.5 Misturando código Java com HTML	75
6.6 EL: Expression language	76
6.7 Exercícios: parâmetros com a Expression Language	76
6.8 Para saber mais: Compilando os arquivos JSP	77
7 Usando Taglibs	78
7.1 Taglibs	78
7.2 Instanciando POJOs	78
7.3 JSTL	79
7.4 Instalação	80
7.5 Cabeçalho para a JSTL core	80
7.6 ForEach	80
7.7 Exercícios: forEach	82
7.8 Exercícios opcionais	83
7.9 Evoluindo nossa listagem	83
7.10 Fazendo ifs com a JSTL	84
7.11 Exercícios: lista de contatos com condicionais	84
7.12 Importando páginas	86
7.13 Exercícios: cabeçalhos e rodapés	87
7.14 Formatação de datas	88
7.15 Exercícios: Formatando a data de nascimento dos contatos	89
7.16 Para saber mais: links com	90
7.17 Exercícios opcionais: Caminho absoluto	91
7.18 Para saber mais: Outras tags	91
8 Tags customizadas com Tagfiles	92
8.1 Porque eu precisaria de outras tags além da JSTL?	92
8.2 Calendários com jQuery	93
8.3 Criando minhas próprias tags com Tagfiles	93
8.4 Exercícios: criando nossa própria tag para calendário	95
8.5 Para saber mais: Outras taglibs no mercado	98
8.6 Desafio: Colocando displaytag no projeto	99
9 MVC - Model View Controller	100
9.1 Servlet ou JSP?	100
9.2 Request Dispatcher	101
9.3 Exercícios: RequestDispatcher	102
9.4 Melhorando o processo	103
9.5 Retomando o design pattern Factory	108

9.6 Exercícios: Criando nossas lógicas e a servlet de controle	108
9.7 Exercícios: Criando uma lógica para remover contatos	109
9.8 Fazendo a lógica para listar os contatos	110
9.9 Exercícios: Lógica para listar contatos	111
9.10 Escondendo nossas páginas	112
9.11 Exercícios opcionais	113
9.12 Model View Controller	113
9.13 Lista de tecnologias: camada de controle	113
9.14 Lista de tecnologias: camada de visualização	114
9.15 Discussão em aula: os padrões Command e Front Controller	115
10 Recursos importantes: Filtros	116
10.1 Reduzindo o acoplamento com Filtros	124
10.2 Exercícios opcionais: Filtro para medir o tempo de execução	121
10.3 Problemas na criação das conexões	122
10.4 Tentando outras estratégias	122
10.5 Reduzindo o acoplamento com Filtros	124
10.6 Exercícios: Filtros	126
11 Spring MVC	128
11.1 Porque precisamos de frameworks MVC?	128
11.2 Um pouco de história	129
11.3 Configurando o Spring MVC	130
11.4 Criando as lógicas	131
11.5 A lógica Olá Mundo!	132
11.6 Para saber mais: Configurando o Spring MVC em casa	133
11.7 Exercícios: Configurando o Spring MVC e testando a configuração	133
11.8 Adicionando tarefas e passando parâmetros	135
11.9 Exercícios: Criando tarefas	138
11.10 Incluindo validação no cadastro de tarefas	140
11.11 Validação com Bean Validation	141
11.12 Exercícios: Validando tarefas	144
11.13 Listando as tarefas e disponibilizando objetos para a view	145
11.14 Exercícios: Listando tarefas	146
11.15 Redirecionando a requisição para outra ação	148
11.16 Exercícios: Removendo e alterando tarefas	149
11.17 Desafio - Calendário	150
11.18 Melhorando a usabilidade da nossa aplicação	150
11.19 Utilizando AJAX para marcar tarefas como finalizadas	152

11.20 Configurar o Spring MVC para acessar arquivos comuns	153
11.21 Exercícios: Ajax	153
11.22 Para saber mais: Alterando valor da data com AJAX	155
11.23 Exercícios Opcionais: Melhorando nosso AJAX	158
12 Spring MVC: Autenticação e autorização	160
12.1 Autenticando usuários: como funciona?	160
12.2 Cookies	160
12.3 Sessão	161
12.4 Configurando o tempo limite	161
12.5 Registrando o usuário logado na sessão	161
12.6 Exercício: Fazendo o login na aplicação	162
12.7 Bloqueando acessos de usuários não logados com Interceptadores	163
12.8 Exercícios: Interceptando as requisições	165
12.9 Exercícios opcionais: Logout	166
13 Spring IoC e deploy da aplicação	168
13.1 Menos acoplamento com inversão de controle e injeção de dependências	168
13.2 Container de Injeção de dependências	171
13.3 Container Spring IoC	171
13.4 Outras formas de injeção	173
13.5 Exercícios: Inversão de controle com o Spring Container	175
13.6 Aprimorando o visual através de CSS	177
13.7 Exercícios opcionais: Aplicando CSS nas páginas	178
13.8 Deploy do projeto em outros ambientes	179
13.9 Exercícios: Deploy com war	180
13.10 Discussão em aula: lidando com diferentes nomes de contexto	182
14 Uma introdução prática ao JPA com Hibernate	183
14.1 Mapeamento Objeto Relacional	183
14.2 Java Persistence API e Frameworks ORM	184
14.3 Bibliotecas do Hibernate e JPA	184
14.4 Mapeando uma classe Tarefa para nosso Banco de Dados	184
14.5 Configurando o JPA com as propriedades do banco	186
14.6 Usando o JPA	187
14.7 Para saber mais: Configurando o JPA com Hibernate em casa	187
14.8 Exercícios: Configurando o JPA e gerando o schema do banco	188
14.9 Trabalhando com os objetos: o EntityManager	190
14.10 Exercícios: Gravando e Carregando objetos	192

Sumário	Caelum
14.11 Removendo e atualizando objeto	193
14.12 Buscando com uma cláusula where	194
14.13 Exercícios: Buscando com JPQL	194
15 E agora?	196
15.1 Os apêndices dessa apostila	196
15.2 Frameworks Web	196
15.3 Frameworks de persistência	197
15.4 Onde seguir seus estudos	198
16 Apêndice - Integração do Spring com JPA	199
16.1 Gerenciando o EntityManager	199
16.2 Configurando o JPA no Spring	200
16.3 Injetando o EntityManager	200
16.4 Baixo acoplamento pelo uso de interface	202
16.5 Gerenciando a transação	204
16.6 Exercícios: Integrando JPA com Spring	206
17 Apêndice - VRaptor3 e produtividade na Web	209
17.1 Motivação: evitando APIs complicadas	209
17.2 Vantagens de um código independente de Request e Response	211
17.3 VRaptor 3	212
17.4 A classe de modelo	213
17.5 Minha primeira lógica de negócios	214
17.6 Redirecionando após a inclusão	215
17.7 Criando o formulário	216
17.8 A lista de produtos	216
17.9 Exercícios	218
17.10 Aprofundando em Injeção de Dependências e Inversão de Controle	221
17.11 Injeção de Dependências com o VRaptor	221
17.12 Escopos dos componentes	222
17.13 Exercícios: Usando Injeção de Dependências para o DAO	223
17.14 Adicionando segurança em nossa aplicação	224
17.15 Interceptando requisições	226
17.16 Exercícios: Construindo a autenticação e a autorização	227
17.17 Melhorando a usabilidade da nossa aplicação	229
17.18 Para saber mais: Requisições: Síncrono x Assíncrono	230
17.19 Para saber mais: AJAX	230
17.20 Adicionando AJAX na nossa aplicação	230

17.21 Exercícios opcionais: Adicionando AJAX na nossa aplicação	232
18 Apêndice - Java EE 6	235
18.1 Java EE 6 e as novidades	235
18.2 Processamento assíncrono	236
18.3 Plugabilidade e Web fragments	238
18.4 Registro dinâmico de Servlets	240
19 Apêndice - Tópicos da Servlet API	242
19.1 Init-params e context-params	242
19.2 welcome-file-list	243
19.3 Propriedades de páginas JSP	243
19.4 Inclusão estática de arquivos	244
19.5 Tratamento de erro em JSP	244
19.6 Descobrindo todos os parâmetros do request	245
19.7 Trabalhando com links com a c:url	245
19.8 Context listener	246
19.9 O ServletContext e o escopo de aplicação	246
19.10 Outros listeners	247

Versão: 20.6.10

ENFRENTANDO O JAVA NA WEB

"Todo homem tem algumas lembranças que ele não conta a todo mundo, mas apenas a seus amigos. Ele tem outras lembranças que ele não revelaria nem mesmo para seus amigos, mas apenas para ele mesmo, e faz isso em segredo. Mas ainda há outras lembranças que o homem tem medo de contar até a ele mesmo, e todo homem decente tem um considerável número dessas coisas guardadas bem no fundo. Alguém até poderia dizer que, quanto mais decente é o homem, maior o número dessas coisas em sua mente." -- Fiodór Dostoiévski, em Memórias do Subsolo

Como fazer a plataforma Java e a Web trabalharem juntas?

1.1 O GRANDE MERCADO DO JAVA NA WEB

Certamente o mercado com mais oportunidades em que o Java está presente é o de Web. Não é por acaso sua popularidade: criar um projeto com Java dá muita liberdade, evitando cair no **vendor lock-in**. Em outras palavras, sua empresa fica independente do fabricante de vários softwares: do servlet container, do banco de dados e até da própria fabricante da sua Virtual Machine! Além disso, podemos fazer todo o desenvolvimento em um sistema operacional e fazer o *deploy* (implantação) em outro.

Apesar de tanta popularidade no ambiente Web, o desenvolvimento com Java não é trivial: é necessário conhecer com certa profundidade as APIs de servlets e de JSP, mesmo que sua equipe venha utilizar frameworks como Struts, VRaptor ou JSF. Conceitos de HTTP, session e cookies também são mandatórios para poder enxergar gargalos e problemas que sua aplicação enfrentará.

Esse curso aborda desde o banco de dados, até o uso de frameworks MVC para desacoplar o seu código. Ao final do curso, ainda veremos o Spring MVC e o Hibernate, duas das ferramentas mais populares entre os requisitos nas muitas vagas de emprego para desenvolvedor Web.

Por fim, falta mencionar sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes e os desafios podem ser feitos quando o curso acabar. De qualquer maneira, recomendamos aos alunos estudar em casa, principalmente àqueles que fazem os cursos intensivos.

Como estudaremos várias tecnologias e ambientes, é comum esbarrarmos em algum erro que se torne um impedimento para prosseguir nos exercícios, portanto não desanime.

Lembre-se de usar o fórum do GUJ (<http://www.guj.com.br/>), onde sua dúvida será respondida prontamente. Você também pode entrar em contato com seu instrutor para tirar dúvidas durante todo o

seu curso.

1.2 LIVROS E SITES INTERESSANTES

É possível aprender muitos dos detalhes e pontos não cobertos no treinamento em tutoriais na Internet em portais como o GUJ, em blogs e em muitos sites especializados.

<http://www.guj.com.br/>

A editora Casa do Código possui livros de Java, como de Android, JSF e JPA, e vende e-books a preços bastante justos:

<http://www.CasaDoCodigo.com.br/>

Não deixe de conhecer os outros cursos da Caelum, inclusive os novos cursos online, onde você pode começar a estudar agora mesmo.

<http://www.alura.com.br/>

BANCOS DE DADOS E JDBC

"O medo é o pai da moralidade" -- Friedrich Wilhelm Nietzsche

Ao término desse capítulo, você será capaz de:

- conectar-se a um banco de dados qualquer através da API JDBC;
- criar uma fábrica de conexões usando o design pattern **Factory**;
- pesquisar dados através de **queries**;
- encapsular suas operações com bancos de dados através de DAO - Data Access Object.

2.1 POR QUE USAR UM BANCO DE DADOS?

Muitos sistemas precisam manter as informações com as quais eles trabalham para permitir consultas futuras, geração de relatórios ou possíveis alterações nas informações. Para que esses dados sejam mantidos para sempre, esses sistemas geralmente guardam essas informações em um banco de dados, que as mantém de forma organizada e prontas para consultas.

A maioria dos bancos de dados comerciais são os chamados relacionais, que é uma forma de trabalhar e pensar diferente ao paradigma orientado a objetos.

O MySQL é o banco de dados que usaremos durante o curso. É um dos mais importantes bancos de dados relacionais, e é gratuito, além de ter uma instalação fácil para todos os sistemas operacionais. Depois de instalado, para acessá-lo via terminal, fazemos da seguinte forma:

```
mysql -u root
```

BANCO DE DADOS

Para aqueles que não conhecem um banco de dados, é recomendado ler um pouco sobre o assunto e também ter uma base de SQL para começar a usar a API JDBC.

O processo de armazenamento de dados é também chamado de **persistência**. A biblioteca de persistência em banco de dados relacionais do Java é chamada JDBC (*Java DataBase Connectivity*), e também existem diversas ferramentas do tipo **ORM** (*Object Relational Mapping*) que facilitam bastante

o uso do JDBC. Neste momento, focaremos nos conceitos e no uso do JDBC. Veremos um pouco da ferramenta de ORM Hibernate ao final deste mesmo curso e, no curso FJ-25, com muitos detalhes, recursos e tópicos avançados.

2.2 PERSISTINDO ATRAVÉS DE SOCKETS?

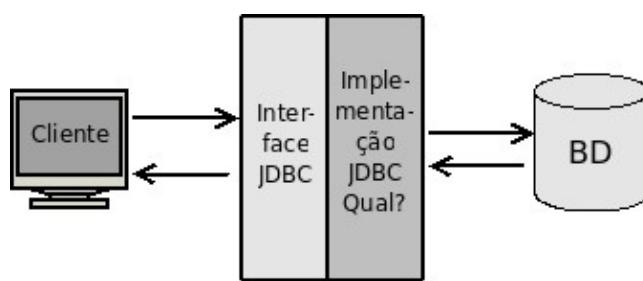
Para conectar-se a um banco de dados, poderíamos abrir *sockets* diretamente com o servidor que o hospeda, por exemplo, um Oracle ou MySQL, e nos comunicarmos com ele através de seu protocolo proprietário.

Mas você conhece o protocolo proprietário de algum banco de dados? Conhecer um protocolo complexo em profundidade é difícil, trabalhar com ele é muito trabalhoso.

Uma segunda ideia seria utilizar uma API específica para cada banco de dados. Antigamente, no PHP, por exemplo, a única maneira de acessar o Oracle era através de funções como `oracle_connect`, `oracle_result`, e assim por diante. O MySQL tinha suas funções análogas, como `mysql_connect`. Essa abordagem facilita muito nosso trabalho por não precisarmos entender o protocolo de cada banco, mas faz com que tenhamos de conhecer uma API um pouco diferente para cada tipo de banco. Além disso, caso precisemos trocar de banco de dados um dia, precisaremos trocar todo o nosso código para refletir a função correta de acordo com o novo banco de dados que estamos utilizando.

2.3 A CONEXÃO EM JAVA

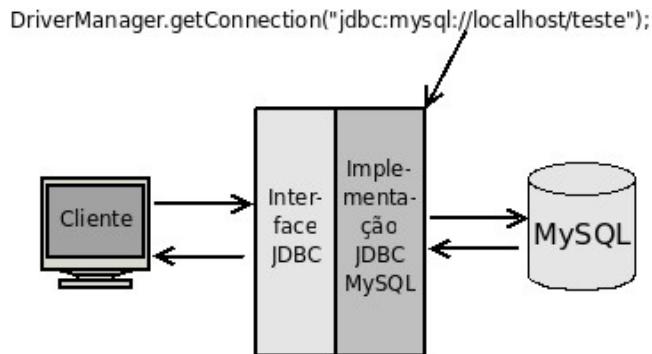
A conexão a um banco de dados é feita de maneira elegante com Java. Para evitar que cada banco tenha a sua própria API e um conjunto de classes e métodos, temos um único conjunto de interfaces muito bem definidas que devem ser implementadas. Esse conjunto de interfaces fica dentro do pacote `java.sql` e nos referiremos a ele como **JDBC**.



Entre as diversas interfaces deste pacote, existe a interface `Connection`, que define métodos para executar uma query (como um `insert` e `select`), comitar transação, fechar a conexão, entre outros. Caso queiramos trabalhar com o MySQL, precisamos de classes concretas que implementem essas interfaces do pacote `java.sql`.

Esse conjunto de classes concretas é quem fará a ponte entre o código cliente que usa a API JDBC e o

banco de dados. São essas classes que sabem se comunicar através do protocolo proprietário do banco de dados. Esse conjunto de classes recebe o nome de **driver**. Todos os principais bancos de dados do mercado possuem **drivers JDBC** para que você possa utilizá-los com Java. O nome driver é análogo ao que usamos para impressoras: como é impossível que um sistema operacional saiba conversar com todo tipo de impressora existente, precisamos de um driver que faça o papel de "tradutor" dessa conversa.



Para abrir uma conexão com um banco de dados, precisamos utilizar sempre um driver. A classe `DriverManager` é a responsável por se comunicar com todos os drivers que você deixou disponível. Para isso, invocamos o método estático `getConnection` com uma `String` que indica a qual banco desejamos nos conectar.

Essa `String` - chamada de **String de conexão JDBC** - que utilizaremos para acessar o MySQL tem sempre a seguinte forma:

```
jdbc:mysql://ip/nome_do_banco
```

Devemos substituir `ip` pelo IP da máquina do servidor e `nome_do_banco` pelo nome do banco de dados a ser utilizado.

Seguindo o exemplo da linha acima e tudo que foi dito até agora, seria possível rodar o exemplo abaixo e receber uma conexão para um banco MySQL, caso ele esteja rodando na mesma máquina:

```
public class JDBCExemplo {  
    public static void main(String[] args) throws SQLException {  
        Connection conexao = DriverManager.getConnection(  
            "jdbc:mysql://localhost/fj21");  
        System.out.println("Conectado!");  
        conexao.close();  
    }  
}
```

Repare que estamos deixando passar a `SQLException`, que é uma exception *checked*, lançada por muitos dos métodos da API de JDBC. Numa aplicação real devemos utilizar `try/catch` nos lugares que julgamos haver possibilidade de recuperar de uma falha com o banco de dados. Também precisamos tomar sempre cuidado para fechar todas as conexões que foram abertas.

Ao testar o código acima, recebemos uma exception. A conexão não pôde ser aberta. Recebemos a

mensagem:

```
java.sql.SQLException: No suitable driver found for  
    jdbc:mysql://localhost/fj21
```

Por quê?

O sistema ainda não achou uma implementação de **driver JDBC** que pode ser usada para abrir a conexão indicada pela URL `jdbc:mysql://localhost/fj21`.

O que precisamos fazer é adicionar o driver do MySQL ao *classpath*, ou seja, o arquivo **.jar** contendo a implementação JDBC do MySQL (*mysql connector*) precisa ser colocado em um lugar visível pelo seu projeto ou adicionado à variável de ambiente `CLASSPATH`. Como usaremos o Eclipse, depois de adicionar o jar do driver JDBC do MySQL na pasta do projeto, faremos isso através de um clique da direita em nosso projeto, *Build Path* e em *Add to Build Path*. Veremos isto passo a passo nos exercícios.

E o `CLASS.forName`?

Até a versão 3 do JDBC, antes de chamar o `DriverManager.getConnection()` era necessário registrar o driver JDBC que iria ser utilizado através do método `Class.forName("com.mysql.jdbc.Driver")`, no caso do MySQL, que carregava essa classe, e essa se comunicava com o `DriverManager`.

A partir do JDBC 4, que está presente no Java 6, esse passo não é mais necessário. Mas lembre-se: caso você utilize JDBC em um projeto com Java 5 ou anterior, será preciso fazer o registro do Driver JDBC, carregando a sua classe, que vai se registrar no `DriverManager`.

Isso também pode ser necessário em alguns servidores de aplicação e web, como no Tomcat 7 ou posterior, por proteção para possíveis vazamentos de memória:

<http://bit.ly/18BpDfG>

Alterando o banco de dados

Teoricamente, basta alterar as duas `Strings` que escrevemos para mudar de um banco para outro. Porém, não é tudo tão simples assim! O código `SQL` que veremos a seguir pode funcionar em um banco e não em outros. Depende de quão aderente ao padrão `ANSI SQL` é seu banco de dados.

Isso só causa dor de cabeça e existem projetos que resolvem isso, como é o caso do Hibernate (www.hibernate.org) e da especificação JPA (Java Persistence API). Veremos um pouco do Hibernate ao final desse curso e bastante sobre ele no FJ-25.

Drivers de outros bancos de dados

Os drivers podem ser baixados normalmente no site do fabricante do banco de dados.

Alguns casos, como no Microsoft SQL Server, existem outros grupos que desenvolvem o driver em <http://jtds.sourceforge.net> . Enquanto isso, você pode achar o driver do MYSQL (chamado de **mysql connector**) no site <http://www.mysql.org>.

2.4 FÁBRICA DE CONEXÕES

Em determinado momento de nossa aplicação, gostaríamos de ter o controle sobre a construção dos objetos da nossa classe. Muito pode ser feito através do construtor, como saber quantos objetos foram instanciados ou fazer o log sobre essas instanciações.

Às vezes, também queremos controlar um processo muito repetitivo e trabalhoso, como abrir uma conexão com o banco de dados. Tomemos como exemplo a classe a seguir, que seria responsável por abrir uma conexão com o banco:

```
public class ConnectionFactory {  
    public Connection getConnection() {  
        try {  
            return DriverManager.getConnection(  
                "jdbc:mysql://localhost/fj21", "root", "");  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Assim, sempre que quisermos obter uma conexão com o banco no nosso código, usaremos o comando a seguir:

```
Connection con = new ConnectionFactory().getConnection();
```

Note que o método `getConnection()` é uma fábrica de conexões, isto é, ele cria novas conexões para nós. Basta invocar o método e recebemos uma conexão pronta para uso, não importando de onde ela veio e eventuais detalhes de criação. Portanto, vamos chamar a classe de `ConnectionFactory` e o método de `getConnection` .

Encapsulando dessa forma, podemos mais tarde mudar a obtenção de conexões, para, por exemplo, usar um mecanismo de *pooling*, que é fortemente recomendável em uma aplicação real.

TRATAMENTO DE EXCEÇÕES

Repare que estamos fazendo um `try/catch` em `SQLException` e relançando-a como uma `RuntimeException`. Fazemos isso para que o seu código que chamará a fábrica de conexões não fique acoplado com a API de JDBC. Toda vez que tivermos que lidar com uma `SQLException`, vamos relançá-las como `RuntimeException`.

Poderíamos ainda criar nossa própria exceção para indicar que ocorreu um erro dentro da nossa Factory, algo como uma `ConnectionFactoryException`.

2.5 DESIGN PATTERNS

Orientação a objetos resolve as grandes dores de cabeça que tínhamos na programação procedural, restringindo e centralizando responsabilidades.

Mas alguns problemas não podemos resolver simplesmente com orientação a objetos, pois não existe palavra chave para uma funcionalidade tão específica.

Alguns desses pequenos problemas aparecem com tanta frequência que as pessoas desenvolvem uma solução "padrão" para ele. Com isso, ao nos defrontarmos com um desses problemas clássicos, podemos rapidamente implementar essa solução genérica com uma ou outra modificação, de acordo com nossa necessidade. Essas soluções padrões tem o nome de **Design Patterns (Padrões de Projeto)**.

A melhor maneira para aprender o que é um **Design Pattern** é vendo como surgiu sua necessidade.

A nossa `ConnectionFactory` implementa o design pattern `Factory`, que prega o encapsulamento da construção (fabricação) de objetos complicados.

A BÍBLIA DOS DESIGN PATTERNS

O livro mais conhecido de **Design Patterns** foi escrito em 1995 e tem trechos de código em C++ e Smalltalk. Mas o que realmente importa são os conceitos e os diagramas que fazem desse livro independente de qualquer linguagem. Além de tudo, o livro é de leitura agradável.

Design Patterns, Erich Gamma et al.

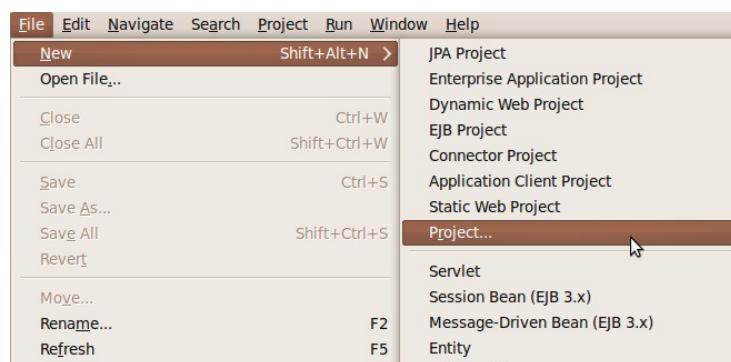
2.6 EXERCÍCIOS: CONNECTIONFACTORY

1. Nos computadores da Caelum, clique no ícone do Eclipse no Desktop;

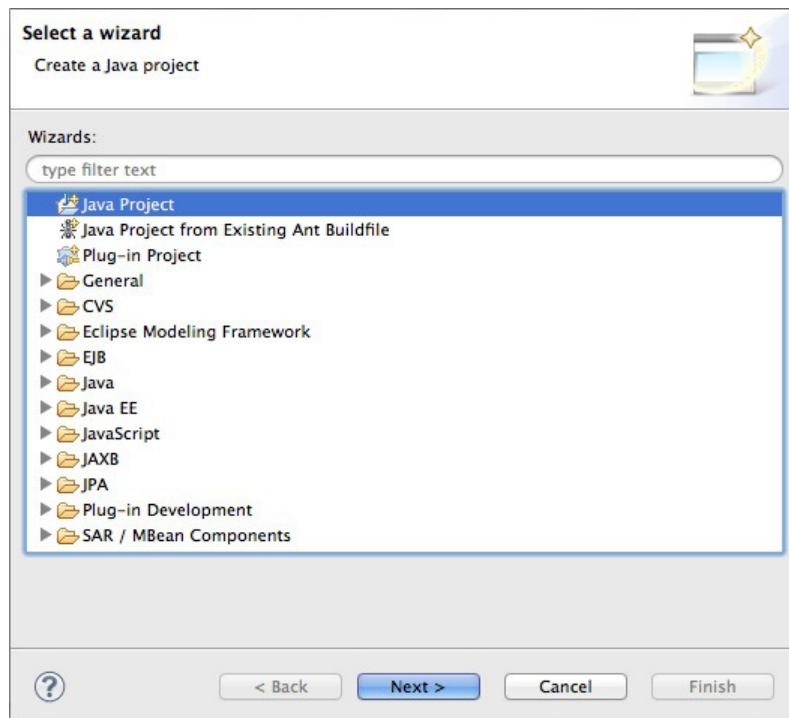
BAIXANDO O ECLIPSE EM CASA

Estamos usando o Eclipse for Java EE Developers. Você pode obtê-lo direto no site do Eclipse em www.eclipse.org

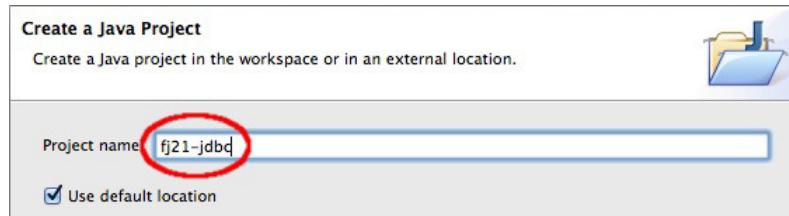
- Feche a tela de *Welcome* caso ela apareça
- Vamos criar um projeto no Eclipse chamado `fj21-jdbc` .
- Vá em **File -> New -> Project:**



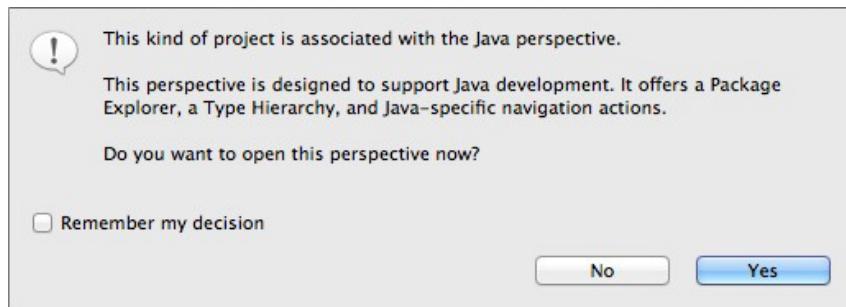
- Selecione **Java Project** e clique em **Next:**



- Coloque o nome do projeto como **fj21-jdbc** e clique em **Finish**:



- Aceite a mudança de perspectiva:

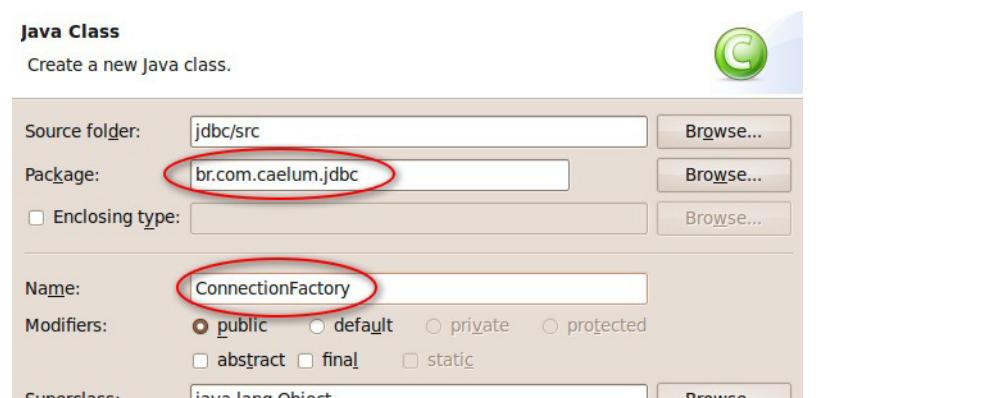


2. Copie o driver do MySQL para o seu projeto.

- no seu Desktop, clique no atalho **Atalho para arquivos do cursos**;
- copie a pasta **21** para o seu Desktop;
- entre na pasta **21** do Desktop e clique da direita no driver do MySQL mais novo, escolha **Copy**;
- vá para sua pasta principal (**home**);
- entre no diretório **workspace**, **fj21-jdbc** ;
- clique da direita e escolha **Paste**: você acaba de colocar o arquivo ".jar" no seu projeto.

3. Vamos criar a classe que fabrica conexões:

- Clique em **File -> New -> Class**.
- Crie-a no pacote **br.com.caelum.jdbc** e nomeie-a como **ConnectionFactory**.



- Clique em **Finish**
- No código, crie o método `getConnection`, que retorna uma nova conexão. Quando perguntado, importe as classes do pacote `java.sql` (**cuidado** para não importar NADA de `com.mysql`).

```
public Connection getConnection() {
    try {
        return DriverManager.getConnection(
            "jdbc:mysql://localhost/fj21", "root", "");
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

4. Crie uma classe chamada `TestaConexao` no pacote `br.com.caelum.jdbc.teste`. Todas as nossas classes de teste deverão ficar nesse pacote.

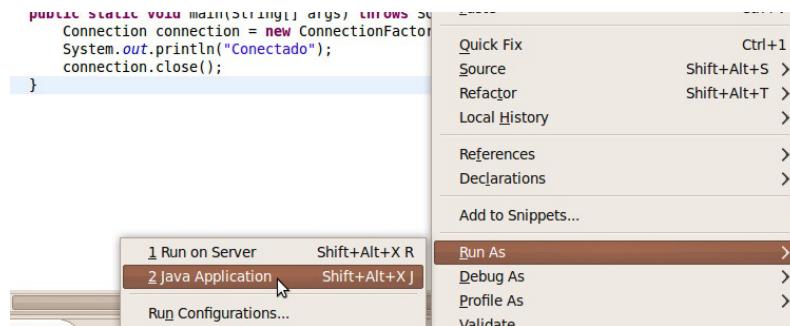
- Crie um método `main` dentro da classe. Use o atalho do Eclipse para ajudar.
- Dentro do `main`, fabrique uma conexão usando a `ConnectionFactory` que criamos. Vamos apenas testar a abertura da conexão e depois fechá-la com o método `close`:

```
Connection connection = new ConnectionFactory().getConnection();
System.out.println("Conexão aberta!");
connection.close();
```

- Trate os erros com `throws`. (Use: `Ctrl + 1` e escolha " add throws declaration ").

5. Rode a sua classe `TestaConexao` pelo Eclipse.

- Clique da direita na sua classe `TestaConexao`
- Escolha **Run as, Java Application** (caso prefira, aprenda a tecla de atalho para agilizar nas próximas execuções)



6. A aplicação não funciona pois o driver não foi encontrado? Esquecemos de colocar o JAR no **classpath!** (*Build Path* no Eclipse)

- Clique no seu projeto com o botão da direita e escolha *Refresh* (ou pressione **F5**).

- Selecione o seu driver do MySQL, clique da direita e escolha **Build Path**, **Add to Build Path**:



- Rode novamente sua aplicação `TestaConexao` agora que colocamos o driver no classpath.

2.7 A TABELA CONTATO

Para criar uma tabela nova, primeiro devemos acessar o terminal e fazermos o comando para logarmos no mysql.

```
mysql -u root
```

Nos preparamos para usar o banco de dados **fj21**:

```
use fj21;
```

A seguinte tabela será usada nos exemplos desse capítulo:

```
create table contatos (
    id BIGINT NOT NULL AUTO_INCREMENT,
    nome VARCHAR(255),
    email VARCHAR(255),
    endereco VARCHAR(255),
    dataNascimento DATE,
    primary key (id)
);
```

No banco de dados relacional, é comum representar um contato (entidade) em uma tabela de contatos.

2.8 JAVABEANS

O que são Javabeans? A pergunta que não quer se calar pode ser respondida muito facilmente uma vez que uma das maiores confusões feitas aí fora é entre Javabeans e Enterprise Java Beans (EJB).

Javabeans são classes que possuem o construtor sem argumentos e métodos de acesso do tipo `get` e `set`! Mais nada! Simples, não? Já os EJBs costumam ser javabeans com características mais avançadas e são o assunto principal do curso FJ-31 da Caelum.

Podemos usar *beans* em diversas situações, como em classes que representam nosso modelo de

dados.

A seguir, você vê um exemplo de uma classe JavaBean que seria equivalente ao nosso modelo de entidade do banco de dados:

```
package br.com.caelum.jdbc.modelo;

public class Contato {

    private Long id;
    private String nome;
    private String email;
    private String endereco;
    private Calendar dataNascimento;

    // métodos get e set para id, nome, email, endereço e dataNascimento

    public String getName() {
        return this.nome;
    }
    public void setName(String novo) {
        this.nome = novo;
    }

    public String getEmail() {
        return this.email;
    }
    public void setEmail(String novo) {
        this.email = novo;
    }

    public String getAddress() {
        return this.endereco;
    }
    public void setAddress(String novo) {
        this.endereco = novo;
    }

    public Long getId() {
        return this.id;
    }
    public void setId(Long novo) {
        this.id = novo;
    }

    public Calendar getDataNascimento() {
        return this.dataNascimento;
    }
    public void setDataNascimento(Calendar dataNascimento) {
        this.dataNascimento = dataNascimento;
    }
}
```

A especificação JavaBeans é muito grande e mais informações sobre essa vasta área que é a base dos componentes escritos em Java pode ser encontrada em:

<http://docs.oracle.com/javase/tutorial/javabeans/>

MÉTODOS GETTERS E SETTERS

Um erro muito comum cometido pelos desenvolvedores Java é a criação dos métodos getters e setters indiscriminadamente, sem ter a real necessidade da existência de tais métodos.

Existe um artigo no blog da Caelum que trata desse assunto:
<http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/>

Os cursos FJ-11 e FJ-22 também mostram isso claramente, quando criam algumas entidades que não possuem apenas getters e setters.

Se você quer saber mais sobre Enterprise Java Beans (EJB), a Caelum oferece o curso FJ-31. Não os confunda com Java Beans!

2.9 INSERINDO DADOS NO BANCO

Para inserir dados em uma tabela de um banco de dados entidade-relacional, basta usar a cláusula **INSERT**. Precisamos especificar quais os campos que desejamos atualizar e os valores.

Primeiro o código SQL:

```
String sql = "insert into contatos " +
    "(nome,email,endereco, dataNascimento)" +
    " values ('" + nome + "', '" + email + "', '" +
    endereco + "', '" + dataNascimento + "')";
```

O exemplo acima possui três pontos negativos que são importantíssimos. O primeiro é que o programador que não escreveu o código original não consegue bater o olho e entender o que está escrito. O que o código acima faz? Lendo rapidamente fica difícil. Mais difícil ainda é saber se faltou uma vírgula, um fecha parênteses talvez? E ainda assim, esse é um caso simples. Existem tabelas com 10, 20, 30 e até mais campos, tornando inviável entender o que está escrito no SQL misturado com as concatenações.

Outro problema é o clássico "preconceito contra Joana D'arc", formalmente chamado de **SQL Injection**. O que acontece quando o contato a ser adicionado possui no nome uma aspas simples? O código SQL se quebra todo e para de funcionar ou, pior ainda, o usuário final é capaz de alterar seu código SQL para executar aquilo que ele deseja (SQL injection)... tudo isso porque escolhemos aquela linha de código e não fizemos o escape de caracteres especiais.

Mais um problema que enxergamos aí é na data. Ela precisa ser passada como uma `String` e no formato que o banco de dados entenda, portanto, se você possui um objeto `java.util.Calendar`, que é o nosso caso, você precisará fazer a conversão desse objeto para a `String`.

Por esses três motivos não usaremos código SQL como mostrado anteriormente. Vamos imaginar algo mais genérico e um pouco mais interessante:

```
String sql = "insert into contatos " +
    "(nome,email,endereco,dataNascimento) " +
    "values (?,?,?,?,?);
```

Existe uma maneira em Java de escrever o código SQL como no primeiro exemplo dessa seção (com concatenações de `String`). Essa maneira não será ensinada durante o curso pois é uma péssima prática que dificulta a manutenção do seu projeto.

Perceba que não colocamos os pontos de interrogação de brincadeira, mas sim porque realmente não sabemos o que desejamos inserir. Estamos interessados em executar aquele código mas não sabemos ainda quais são os **parâmetros** que utilizaremos nesse código SQL que será executado, chamado de `statement`.

As cláusulas são executadas em um banco de dados através da interface `PreparedStatement`. Para receber um `PreparedStatement` relativo à conexão, basta chamar o método `prepareStatement`, passando como argumento o comando SQL com os valores vindos de variáveis preenchidos com uma interrogação.

```
String sql = "insert into contatos " +
    "(nome,email,endereco,dataNascimento) " +
    "values (?,?,?,?,?);"
PreparedStatement stmt = connection.prepareStatement(sql);
```

Logo em seguida, chamamos o método `setString` do `PreparedStatement` para preencher os valores que são do tipo `String`, passando a posição (começando em 1) da interrogação no SQL e o valor que deve ser colocado:

```
// preenche os valores
stmt.setString(1, "Caelum");
stmt.setString(2, " contato@caelum.com.br");
stmt.setString(3, "R. Vergueiro 3185 cj57");
```

Precisamos definir também a data de nascimento do nosso contato, para isso, precisaremos de um objeto do tipo `java.sql.Date` para passarmos para o nosso `PreparedStatement`. Nesse exemplo, vamos passar a data atual. Para isso, vamos passar um `long` que representa os milissegundos da data atual para dentro de um `java.sql.Date`, que é o tipo suportado pela API JDBC. Vamos utilizar a classe `Calendar` para conseguirmos esses milissegundos:

```
java.sql.Date dataParaGravar = new java.sql.Date(
    Calendar.getInstance().getTimeInMillis());
stmt.setDate(4, dataParaGravar);
```

Por fim, uma chamada a `execute()` executa o comando SQL:

```
stmt.execute();
```

Imagine todo esse processo sendo escrito toda vez que desejar inserir algo no banco? Ainda não

consegue visualizar o quanto destrutivo isso pode ser?

Veja o exemplo abaixo, que abre uma conexão e insere um contato no banco:

```
public class JDBCInsere {  
  
    public static void main(String[] args) throws SQLException {  
  
        // conectando  
        Connection con = new ConnectionFactory().getConnection();  
  
        // cria um preparedStatement  
        String sql = "insert into contatos" +  
                    " (nome,email,endereco,dataNascimento)" +  
                    " values (?, ?, ?, ?);";  
        PreparedStatement stmt = con.prepareStatement(sql);  
  
        // preenche os valores  
        stmt.setString(1, "Caelum");  
        stmt.setString(2, "contato@caelum.com.br");  
        stmt.setString(3, "R. Vergueiro 3185 cj57");  
        stmt.setDate(4, new java.sql.Date(  
            Calendar.getInstance().getTimeInMillis()));  
  
        // executa  
        stmt.execute();  
        stmt.close();  
  
        System.out.println("Gravado!");  
  
        con.close();  
    }  
}
```

Para saber mais: Fechando a conexão propriamente

Não é comum utilizar JDBC diretamente hoje em dia. O mais praticado é o uso de alguma API de ORM como a **JPA** ou o **Hibernate**. Tanto na JDBC quanto em bibliotecas ORM deve-se prestar atenção no momento de fechar a conexão.

O exemplo dado acima não fecha caso algum erro ocorra no momento de inserir um dado no banco de dados. O comum é fechar a conexão em um bloco `finally`:

```
public class JDBCInsere {  
    public static void main(String[] args) throws SQLException {  
        Connection con = null;  
        try {  
            con = new ConnectionFactory().getConnection();  
  
            // faz um monte de operações.  
            // que podem lançar exceptions runtime e SQLException  
        } catch(SQLException e) {  
            System.out.println(e);  
        } finally {  
            con.close();  
        }  
    }  
}
```

Dessa forma, mesmo que o código dentro do `try` lance exception, o `con.close()` será executado. Garantimos que não deixaremos uma conexão pendurada sem uso. Esse código pode ficar muito maior se quisermos ir além. Pois o que acontece no caso de `con.close` lançar uma exception? Quem a tratará?

Trabalhar com recursos caros, como conexões, sessões, threads e arquivos, sempre deve ser muito bem pensado. Deve-se tomar cuidado para não deixar nenhum desses recursos abertos, pois poderá vazar algo precioso da nossa aplicação. Como veremos durante o curso, é importante centralizar esse tipo de código em algum lugar, para não repetir uma tarefa complicada como essa.

Além disso, há a estrutura do Java 7 conhecida como *try-with-resources*. Ela permite declarar e inicializar, dentro do `try`, objetos que implementam `AutoCloseable`. Dessa forma, ao término do `try`, o próprio compilador inserirá instruções para invocar o `close` desses recursos, além de se precaver em relação a exceções que podem surgir por causa dessa invocação. Nossa código ficaria mais reduzido e organizado, além do escopo de `con` só valer dentro do `try`:

```
try(Connection con = new ConnectionFactory().getConnection()) {  
    // faz um monte de operações.  
    // que podem lançar exceptions runtime e SQLException  
} catch(SQLException e) {  
    System.out.println(e);  
}
```

Para saber mais: A má prática Statement

Em vez de usar o `PreparedStatement`, você pode usar uma interface mais simples chamada `Statement`, que simplesmente executa uma cláusula SQL no método `execute`:

```
Statement stmt = con.createStatement();  
stmt.execute("INSERT INTO ...");  
stmt.close();
```

Mas prefira a classe `PreparedStatement` que é mais rápida que `Statement` e deixa seu código muito mais limpo.

Geralmente, seus comandos SQL conterão valores vindos de variáveis do programa Java; usando `Statements`, você terá que fazer muitas concatenações, mas usando `PreparedStatements`, isso fica mais limpo e fácil.

Para saber mais: JodaTime

A API de datas do Java, mesmo considerando algumas melhorias da `Calendar` em relação a `Date`, ainda é muito pobre. Na versão 8 do Java temos uma API nova, a `java.time`, que facilita bastante o trabalho. Ela é baseada na excelente biblioteca de datas chamada JodaTime.

2.10 DAO - DATA ACCESS OBJECT

Já foi possível sentir que colocar código SQL dentro de suas classes de lógica é algo nem um pouco elegante e muito menos viável quando você precisa manter o seu código.

Quantas vezes você não ficou bravo com o programador responsável por aquele código ilegível?

A ideia a seguir é remover o código de acesso ao banco de dados de suas classes de lógica e colocá-lo em uma classe responsável pelo acesso aos dados. Assim o código de acesso ao banco de dados fica em um lugar só, tornando mais fácil a manutenção.

Que tal se pudéssemos chamar um método adciona que adiciona um `Contato` ao banco?

Em outras palavras quero que o código a seguir funcione:

```
// adiciona os dados no banco
Misterio bd = new Misterio();
bd.adiciona("meu nome", "meu email", "meu endereço", meuCalendar);
```

Tem algo estranho nesse código. Repare que todos os parâmetros que estamos passando são as informações do contato. Se contato tivesse 20 atributos, passaríamos 20 parâmetros? Java é orientado a `Strings`? Vamos tentar novamente: em outras palavras quero que o código a seguir funcione:

```
// adiciona um contato no banco
Misterio bd = new Misterio();

// método muito mais elegante
bd.adiciona(contato);
```

Tentaremos chegar ao código anterior: seria muito melhor e mais elegante poder chamar um único método responsável pela inclusão, certo?

```
public class TestaInsere {

    public static void main(String[] args) {

        // pronto para gravar
        Contato contato = new Contato();
        contato.setNome("Caelum");
        contato.setEmail("contato@caelum.com.br");
        contato.setEndereco("R. Vergueiro 3185 cj87");
        contato.setDataNascimento(Calendar.getInstance());

        // grava nessa conexão!!!
        Misterio bd = new Misterio();

        // método elegante
        bd.adiciona(contato);

        System.out.println("Gravado!");
    }
}
```

O código anterior já mostra o poder que alcançaremos: através de uma única classe seremos capazes de acessar o banco de dados e, mais ainda, somente através dessa classe será possível acessar os dados.

Esta ideia, inocente à primeira vista, é capaz de isolar todo o acesso a banco em classes bem simples,

cuja instância é um **objeto** responsável por **acessar os dados**. Da responsabilidade deste objeto surgiu o nome de **Data Access Object** ou simplesmente **DAO**, um dos mais famosos padrões de projeto (*design pattern*).

O que falta para o código acima funcionar é uma classe chamada `ContatoDao` com um método chamado `adiciona`. Vamos criar uma que se conecta ao banco ao construirmos uma instância dela:

```
public class ContatoDao {  
  
    // a conexão com o banco de dados  
    private Connection connection;  
  
    public ContatoDao() {  
        this.connection = new ConnectionFactory().getConnection();  
    }  
}
```

Agora que todo `ContatoDao` possui uma conexão com o banco, podemos focar no método `adiciona`, que recebe um `Contato` como argumento e é responsável por adicioná-lo através de código SQL:

```
public void adiciona(Contato contato) {  
    String sql = "insert into contatos " +  
        "(nome,email,endereco,dataNascimento)" +  
        " values (?,?,?,?,?)";  
  
    try {  
        // prepared statement para inserção  
        PreparedStatement stmt = con.prepareStatement(sql);  
  
        // seta os valores  
  
        stmt.setString(1,contato.getNome());  
        stmt.setString(2,contato.getEmail());  
        stmt.setString(3,contato.getEndereco());  
        stmt.setDate(4, new Date(  
            contato.getDataNascimento().getTimeInMillis()));  
  
        // executa  
        stmt.execute();  
        stmt.close();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Encapsulamos a `SQLException` em uma `RuntimeException` mantendo a ideia anterior da `ConnectionFactory` de desacoplar o código de API de JDBC.

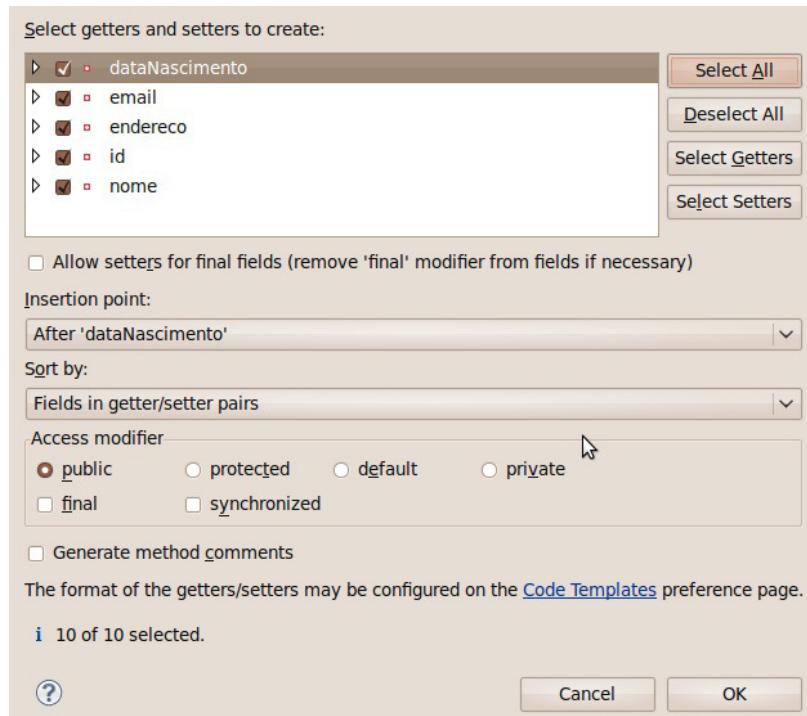
2.11 EXERCÍCIOS: JAVABEANS E CONTATODAO

1. Crie a classe de `Contato` no pacote `br.com.caelum.jdbc.modelo`. Ela será nosso JavaBean para representar a entidade do banco. Deverá ter id, nome, email, endereço e uma data de nascimento.

Coloque os atributos na classe e gere os getters e setters para cada atributo:

```
public class Contato {  
    private Long id;  
    private String nome;  
    private String email;  
    private String endereco;  
    private Calendar dataNascimento;  
}
```

Dica: use o atalho do Eclipse para gerar os getters e setters. Aperte `Ctrl + 3`, digite `ggas` que é a abreviação de `Generate getters and setters` e selecione todos os getters e setters.



2. Vamos desenvolver nossa classe de DAO. Crie a classe `ContatoDao` no pacote `br.com.caelum.jdbc.dao`. Seu papel será gerenciar a conexão e inserir Contatos no banco de dados.

Para a conexão, vamos criá-la no construtor e salvar em um atributo:

```
public class ContatoDao {
```

```

    // a conexão com o banco de dados
    private Connection connection;

    public ContatoDao() {
        this.connection = new ConnectionFactory().getConnection();
    }

}

```

Use o Eclipse para ajudar com os imports! (atälho *Ctrl + Shift + O*)

O próximo passo é criar o método de adição de contatos. Ele deve receber um objeto do tipo `Contato` como argumento e encapsular totalmente o trabalho com o banco de dados. Internamente, use um `PreparedStatement` como vimos na aula para executar o SQL:

```

public void adiciona(Contato contato) {
    String sql = "insert into contatos " +
        "(nome,email,endereco,dataNascimento)" +
        " values (?,?,?,?)";

    try {
        // prepared statement para inserção
        PreparedStatement stmt = connection.prepareStatement(sql);

        // seta os valores
        stmt.setString(1,contato.getNome());
        stmt.setString(2,contato.getEmail());
        stmt.setString(3,contato.getEndereco());
        stmt.setDate(4, new Date(
            contato.getDataNascimento().getTimeInMillis()));

        // executa
        stmt.execute();
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

ATENÇÃO: Lembre-se de importar as classes de SQL do pacote `java.sql`, inclusive a classe `Date`!

3. Para testar nosso DAO, desenvolva uma classe de testes com o método `main`. Por exemplo, uma chamada `TestaInsere` no pacote `br.com.caelum.jdbc.teste`. Gere o `main` pelo Eclipse.

O nosso programa de testes deve, dentro do `main`, criar um novo objeto `Contato` com dados de teste e chamar a nova classe `ContatoDao` para adicioná-lo ao banco de dados:

```

// pronto para gravar
Contato contato = new Contato();
contato.setNome("Caelum");
contato.setEmail("contato@caelum.com.br");
contato.setEndereco("R. Vergueiro 3185 cj57");
contato.setDataNascimento(Calendar.getInstance());

// grava nessa conexão!!!
ContatoDao dao = new ContatoDao();

```

```

// método elegante
dao.adiciona(contato);

System.out.println("Gravado!");

```

Execute seu programa e veja se tudo correu bem.

4. Verifique se o contato foi adicionado. Abra o terminal e digite:

```

mysql -u root
use fj21;
select * from contatos;

exit para sair do console do MySQL.

```

2.12 FAZENDO PESQUISAS NO BANCO DE DADOS

Para pesquisar também utilizamos a interface `PreparedStatement` para montar nosso comando SQL. Mas como uma pesquisa possui um retorno (diferente de uma simples inserção), usaremos o método `executeQuery` que retorna todos os registros de uma determinada query.

O objeto retornado é do tipo `ResultSet` do JDBC, o que nos permite navegar por seus registros através do método `next`. Esse método retornará `false` quando chegar ao fim da pesquisa, portanto ele é normalmente utilizado para fazer um laço nos registros:

```

// pega a conexão e o Statement
Connection con = new ConnectionFactory().getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");

// executa um select
ResultSet rs = stmt.executeQuery();

// itera no ResultSet
while (rs.next()) {
}

rs.close();
stmt.close();
con.close();

```

Para retornar o valor de uma coluna no banco de dados, basta chamar um dos métodos `get` do `ResultSet`, dentre os quais, o mais comum: `getString`.

```

// pega a conexão e o Statement
Connection con = new ConnectionFactory().getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");

// executa um select
ResultSet rs = stmt.executeQuery();

// itera no ResultSet
while (rs.next()) {
    String nome = rs.getString("nome");
    String email = rs.getString("email")
}

```

```

        System.out.println(nome + " _ " + email);
    }

stmt.close();
con.close();

```

RECURSO AVANÇADO: O CURSOR

Assim como o cursor do banco de dados, só é possível mover para o próximo registro. Para permitir um processo de leitura para trás é necessário especificar na abertura do `ResultSet` que tal cursor deve ser utilizado.

Mas, novamente, podemos aplicar as ideias de **DAO** e criar um método `getLista()` no nosso `ContatoDao`. Mas o que esse método retornaria? Um `ResultSet`? E teríamos o código de manipulação de `ResultSet` espalhado por todo o código? Vamos fazer nosso `getLista()` devolver algo mais interessante, uma lista de `Contato`:

```

PreparedStatement stmt = this.connection
    .prepareStatement("select * from contatos");
ResultSet rs = stmt.executeQuery();

List<Contato> contatos = new ArrayList<Contato>();

while (rs.next()) {

    // criando o objeto Contato
    Contato contato = new Contato();
    contato.setNome(rs.getString("nome"));
    contato.setEmail(rs.getString("email"));
    contato.setEndereco(rs.getString("endereco"));

    // montando a data através do Calendar
    Calendar data = Calendar.getInstance();
    data.setTime(rs.getDate("dataNascimento"));
    contato.setDataNascimento(data);

    // adicionando o objeto à lista
    contatos.add(contato);
}

rs.close();
stmt.close();

return contatos;

```

2.13 EXERCÍCIOS: LISTAGEM

- Crie o método `getLista` na classe `ContatoDao`. Importe `List` de `java.util`:

```

public List<Contato> getLista() {
    try {
        List<Contato> contatos = new ArrayList<Contato>();

```

```

        PreparedStatement stmt = this.connection.
            prepareStatement("select * from contatos");
        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            // criando o objeto Contato
            Contato contato = new Contato();
            contato.setId(rs.getLong("id"));
            contato.setNome(rs.getString("nome"));
            contato.setEmail(rs.getString("email"));
            contato.setEndereco(rs.getString("endereco"));

            // montando a data através do Calendar
            Calendar data = Calendar.getInstance();
            data.setTime(rs.getDate("dataNascimento"));
            contato.setDataNascimento(data);

            // adicionando o objeto à lista
            contatos.add(contato);
        }
        rs.close();
        stmt.close();
        return contatos;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

2. Vamos usar o método `getLista` para listar todos os contatos do nosso banco de dados.

Crie uma classe chamada `TestaLista` com um método `main`:

- Crie um `ContatoDao`:

```
ContatoDao dao = new ContatoDao();
```

- Liste os contatos com o DAO:

```
List<Contato> contatos = dao.getLista();
```

- Itere nessa lista e imprima as informações dos contatos:

```

for (Contato contato : contatos) {
    System.out.println("Nome: " + contato.getNome());
    System.out.println("Email: " + contato.getEmail());
    System.out.println("Endereço: " + contato.getEndereco());
    System.out.println("Data de Nascimento: " +
        contato.getDataNascimento().getTime() + "\n");
}

```

3. Rode o programa anterior clicando em **Run > Run as > Java Application** (aproveite para aprender a tecla de atalho para executar a aplicação).

2.14 UM POUCO MAIS...

- Assim como o MySQL existem outros bancos de dados gratuitos e **opensource** na internet. O HSQLDB é um banco desenvolvido em Java que pode ser acoplado a qualquer aplicação e

libera o cliente da necessidade de baixar qualquer banco de dados antes da instalação de um produto Java!

- O Hibernate tomou conta do mercado e virou febre mundial pois não se faz necessário escrever uma linha de código SQL!
- Se um projeto não usa nenhuma das tecnologias de ORM (**Object Relational Mapping**) disponíveis, o mínimo a ser feito é seguir o DAO.

2.15 EXERCÍCIOS OPCIONAIS

1. A impressão da data de nascimento ficou um pouco estranha. Para formatá-la, pesquise sobre a classe `SimpleDateFormat`.
2. Crie uma classe chamada `DAOException` que estenda de `RuntimeException` e utilize-a no seu `ContatoDao`.
3. Use cláusulas `where` para refinar sua pesquisa no banco de dados. Por exemplo: `where nome like 'C%'`
4. Crie o método `pesquisar` que recebe um id (int) e retorna um objeto do tipo `Contato`.

Desafios

1. Faça conexões para outros tipos de banco de dados disponíveis.

2.16 OUTROS MÉTODOS PARA O SEU DAO

Agora que você já sabe usar o `PreparedStatement` para executar qualquer tipo de código SQL e `ResultSet` para receber os dados retornados da sua pesquisa fica simples, porém maçante, escrever o código de diferentes métodos de uma classe típica de DAO.

Veja primeiro o método `altera`, que recebe um `contato` cujos valores devem ser alterados:

```
public void altera(Contato contato) {
    String sql = "update contatos set nome=?, email=?, endereco=?, " +
        "dataNascimento=? where id=?";
    try {
        PreparedStatement stmt = connection.prepareStatement(sql);
        stmt.setString(1, contato.getNome());
        stmt.setString(2, contato.getEmail());
        stmt.setString(3, contato.getEndereco());
        stmt.setDate(4, new Date(contato.getDataNascimento()
            .getTimeInMillis()));
        stmt.setLong(5, contato.getId());
        stmt.execute();
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

Não existe nada de novo nas linhas acima. Uma execução de query! Simples, não?

O código para remoção: começa com uma query baseada em um contato, mas usa somente o id dele para executar a query do tipo delete:

```
public void remove(Contato contato) {  
    try {  
        PreparedStatement stmt = connection.prepareStatement("delete " +  
            "from contatos where id=?");  
        stmt.setLong(1, contato.getId());  
        stmt.execute();  
        stmt.close();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

2.17 EXERCÍCIOS OPCIONAIS - ALTERAR E REMOVER

1. Adicione o método para alterar contato no seu `ContatoDao`.

```
public void altera(Contato contato) {  
    String sql = "update contatos set nome=?, email=?,"+  
        "endereco=?, dataNascimento=? where id=?";  
  
    try {  
        PreparedStatement stmt = connection  
            .prepareStatement(sql);  
        stmt.setString(1, contato.getNome());  
        stmt.setString(2, contato.getEmail());  
        stmt.setString(3, contato.getEndereco());  
        stmt.setDate(4, new Date(contato.getDataNascimento())  
            .getTimeInMillis());  
        stmt.setLong(5, contato.getId());  
        stmt.execute();  
        stmt.close();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

2. Adicione o método para remover contato no seu `ContatoDao`

```
public void remove(Contato contato) {  
    try {  
        PreparedStatement stmt = connection  
            .prepareStatement("delete from contatos where id=?");  
        stmt.setLong(1, contato.getId());  
        stmt.execute();  
        stmt.close();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

3. Use os métodos criados anteriormente para fazer testes com o seu banco de dados: atualize e remova um contato.

4. Crie uma classe chamada Funcionario com os campos id (Long), nome, usuario e senha (String).
5. Crie uma tabela no banco de dados chamada funcionarios.
6. Crie uma classe DAO para Funcionario.
7. Use-a para instanciar novos funcionários e colocá-los no seu banco.

O QUE É JAVA EE?

"Ensinar é aprender duas vezes." -- Joseph Joubert

Ao término desse capítulo, você será capaz de:

- Entender o que é o Java Enterprise Edition;
- Diferenciar um Servidor de Aplicação de um Servlet Container;
- Instalar um Servlet Container como o Apache Tomcat;
- Configurar um Servlet Container dentro do Eclipse.

3.1 COMO O JAVA EE PODE TE AJUDAR A ENFRENTAR PROBLEMAS

As aplicações Web de hoje em dia já possuem regras de negócio bastante complicadas. Codificar essas muitas regras já representam um grande trabalho. Além dessas regras, conhecidas como requisitos funcionais de uma aplicação, existem outros requisitos que precisam ser atingidos através da nossa infraestrutura: persistência em banco de dados, transação, acesso remoto, web services, gerenciamento de threads, gerenciamento de conexões HTTP, cache de objetos, gerenciamento da sessão web, balanceamento de carga, entre outros. São chamados de **requisitos não-funcionais**.

Se formos também os responsáveis por escrever código que trate desses outros requisitos, teríamos muito mais trabalho a fazer. Tendo isso em vista, a Sun criou uma série de especificações que, quando implementadas, podem ser usadas por desenvolvedores para tirar proveito e reutilizar toda essa infraestrutura já pronta.

O **Java EE** (Java Enterprise Edition) consiste de uma série de especificações bem detalhadas, dando uma receita de como deve ser implementado um software que faz cada um desses serviços de infraestrutura.

Veremos no decorrer desse curso vários desses serviços e como utilizá-los, focando no ambiente de desenvolvimento web através do Java EE. Veremos também conceitos muito importantes, para depois conceituar termos fundamentais como **servidor de aplicação** e **containers**.

Porque a Sun faz isso? A ideia é que você possa criar uma aplicação que utilize esses serviços. Como esses serviços são bem complicados, você não perderá tempo implementando essa parte do sistema. Existem implementações tanto open source quanto pagas, ambas de boa qualidade.

Algum dia, você pode querer trocar essa implementação atual por uma que é mais rápida em determinados pontos, que use menos memória, etc. Fazendo essa mudança de implementação você não precisará alterar seu software, já que o Java EE é uma especificação muito bem determinada. O que muda é a implementação da especificação: você tem essa liberdade, não está preso a um código e a especificação garante que sua aplicação funcionará com a implementação de outro fabricante. Esse é um atrativo muito grande para grandes empresas e governos, que querem sempre evitar o **vendor lock-in**: expressão usada quando você está preso sempre nas mãos de um único fabricante.

ONDE ENCONTRAR AS ESPECIFICAÇÕES

O grupo responsável por gerir as especificações usa o site do Java Community Process:
<http://www.jcp.org/>

Lá você pode encontrar tudo sobre as **Java Specification Requests** (JSR), isto é, os novos pedidos de bibliotecas e especificações para o Java, tanto para JavaSE, quanto EE e outros.

Sobre o Java EE, você pode encontrar em: <http://java.sun.com/javaee/>

J2EE

O nome J2EE era usado nas versões mais antigas, até a 1.4. Hoje, o nome correto é Java EE, por uma questão de marketing, mas você ainda vai encontrar muitas referências ao antigo termo J2EE.

3.2 ALGUMAS ESPECIFICAÇÕES DO JAVA EE

As APIs a seguir são as principais dentre as disponibilizadas pelo Java Enterprise:

- JavaServer Pages (JSP), Java Servlets, Java Server Faces (JSF) (trabalhar para a Web, onde é focado este curso)
- Enterprise JavaBeans Components (EJB) e Java Persistence API (JPA). (objetos distribuídos, clusters, acesso remoto a objetos etc)
- Java API for XML Web Services (JAX-WS), Java API for XML Binding (JAX-B) (trabalhar com arquivos xml e webservices)
- Java Authentication and Authorization Service (JAAS) (API padrão do Java para segurança)
- Java Transaction API (JTA) (controle de transação no contêiner)

- Java Message Service (JMS) (troca de mensagens assíncronas)
- Java Naming and Directory Interface (JNDI) (espaço de nomes e objetos)
- Java Management Extensions (JMX) (administração da sua aplicação e estatísticas sobre a mesma)

A última versão disponível da especificação do Java EE é a versão 7, lançada em 12 de junho de 2013. É uma versão ainda muito recente, com poucas ferramentas e servidores disponíveis. A versão mais usada no mercado é a versão 6, de 2009. Este curso é focado na versão 6 que você encontrará no mercado e já apresentando as novidades do novo Java EE 7 que deve ganhar espaço no mercado nos próximos anos.

Neste curso FJ-21, atacamos especialmente JSP e Servlets. No curso FJ-26, estuda-se com profundidade JSF com CDI e JBoss Seam. No FJ-25 é apresentado em detalhe JPA com Hibernate e o EJB. No FJ-31, estuda-se as especificações mais relacionadas a sistemas de alto desempenho: EJB, JNDI, JMS, JPA, JAX-B além de Web Services (JAX-WS e JAX-RS).

JSP e Servlets são sem dúvida as especificações essenciais que todo desenvolvedor Java vai precisar para desenvolver com a Web. Mesmo usando frameworks e bibliotecas que facilitam o trabalho para a Web, conhecer bem essas especificações é certamente um diferencial, e fará com que você entenda motivações e dificuldades, auxiliando na tomada de decisões arquiteturais e de design.

3.3 SERVIDOR DE APLICAÇÃO

Como vimos, o Java EE é um grande conjunto de especificações. Essas especificações, quando implementadas, vão auxiliar bastante o desenvolvimento da sua aplicação, pois você não precisará se preocupar com grande parte de código de infraestrutura, que demandaria muito trabalho.

Como fazer o "*download do Java EE*"? O Java EE é apenas um grande PDF, uma especificação, detalhando quais especificações fazem parte deste. Para usarmos o software, é necessário fazer o download de uma **implementação** dessas especificações.

Existem diversas dessas implementações. Já que esse software tem papel de **servir** sua aplicação para auxiliá-la com serviços de infraestrutura, esse software ganha o nome de **servidor de aplicação**. A própria Sun/Oracle desenvolve uma dessas implementações, o **Glassfish** que é open source e gratuito, porém não é o líder de mercado apesar de ganhar força nos últimos tempos.

Existem diversos servidores de aplicação famosos compatíveis com a especificação do J2EE 1.4, Java EE 5 e alguns já do Java EE 6. O JBoss é um dos líderes do mercado e tem a vantagem de ser gratuito e open source. Alguns softwares implementam apenas uma parte dessas especificações do Java EE, como o Apache Tomcat, que só implementa JSP e Servlets (como dissemos, duas das principais especificações), portanto não é totalmente correto chamá-lo de servidor de aplicação. A partir do Java EE 6, existe o

termo "*application server web profile*", para poder se referenciar a servidores que não oferecem tudo, mas um grupo menor de especificações, consideradas essenciais para o desenvolvimento web.

Você pode ver uma lista de servidores Java EE 5 aqui:
<http://java.sun.com/javaee/overview/compatibility-javaee5.jsp>

E Java EE 6 aqui, onde a lista ainda está crescendo:
<http://java.sun.com/javaee/overview/compatibility.jsp>

Alguns dos servidores de aplicação mais conhecidos do mercado:

- Oracle/Sun, GlassFish Server Open Source Edition 4.0, gratuito, Java EE 7;
- RedHat, JBoss Application Server 7.x, gratuito, Java EE 6;
- Apache, Apache Geronimo, gratuito, Java EE 6 (não certificado);
- Oracle/BEA, Oracle WebLogic Server 8.x, Java EE 6;
- IBM, IBM WebSphere Application Server, Java EE 6;
- SAP, SAP NetWeaver Application Server ou SAP Web Application Server, Java EE 6 Web Profile;

Nos cursos da Caelum utilizamos o Apache Tomcat e o RedHat JBoss, mas todo conhecimento adquirido aqui pode ser aplicado com facilidade para os outros servidores compatíveis, mudando apenas a forma de configurá-los.

No curso FJ-31, estuda-se profundamente algumas das outras tecnologias envolvidas no Java EE: a JPA, o EJB, o JMS, o JAX-WS para Web Services e o JNDI.

3.4 SERVLET CONTAINER

O Java EE possui várias especificações, entre elas, algumas específicas para lidar com o desenvolvimento de uma aplicação Web:

- Servlet
- JSP
- JSTL
- JSF

Um **Servlet Container** é um servidor que suporta essas funcionalidades, mas não necessariamente o Java EE Web Profile nem o Java EE completo. É indicado a quem não precisa de tudo do Java EE e está interessado apenas na parte web (boa parte das aplicações de médio porte encaixam-se nessa categoria).

Há alguns servlet containers famosos no mercado. O mais famoso é o Apache Tomcat, mas há outros como o Jetty, que nós da Caelum usamos muito em projetos e o Google usa em seu cloud Google App Engine.

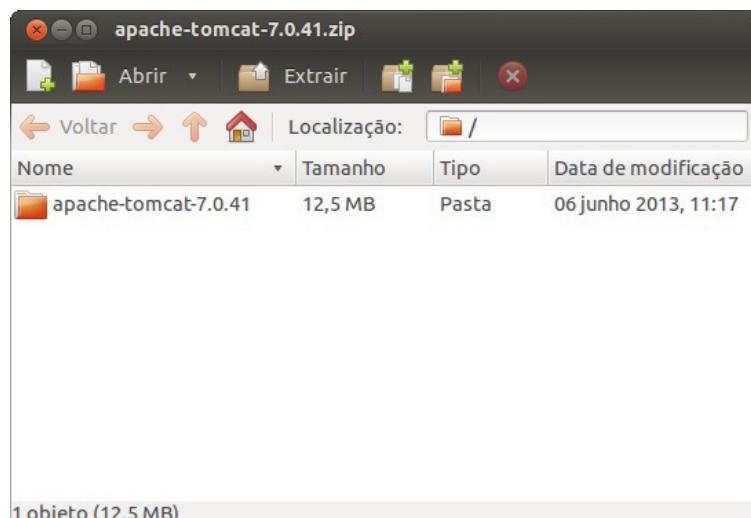
3.5 EXERCÍCIOS: PREPARANDO O TOMCAT

Para preparar o Tomcat na Caelum, siga os seguintes passos:

1. Entre na pasta **21** do Desktop e selecione o arquivo do **apache-tomcat-7.x**;

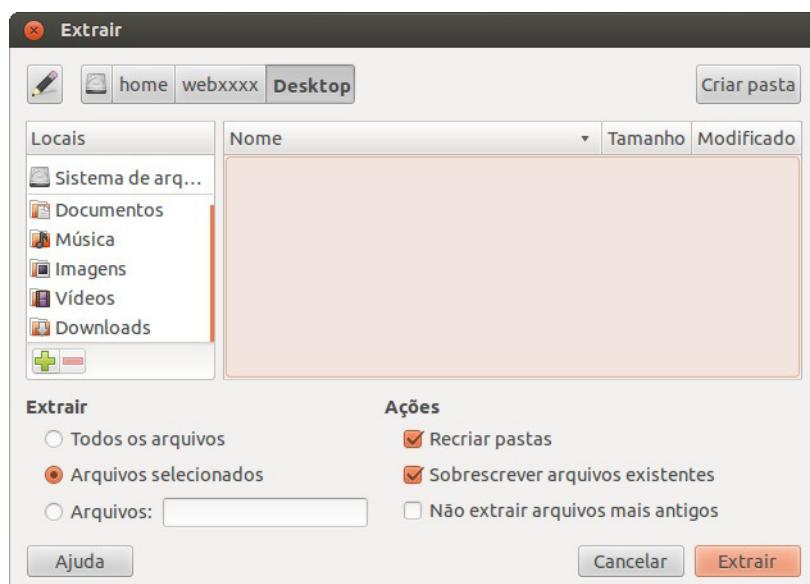


2. Dê dois cliques para abrir o Archive Manager do Linux;

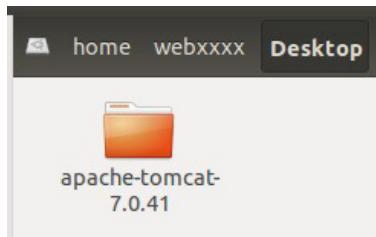


3. Clique em **Extract**;

4. Escolha o seu **Desktop** e clique em extract;



- O resultado é uma pasta chamada **apache-tomcat-7.x**. Pronto, o tomcat já está instalado.



3.6 PREPARANDO O TOMCAT EM CASA

Baixe o Tomcat 7 em <http://tomcat.apache.org/> na página de downloads da versão que escolher, você precisa de uma "Binary Distribution". Mesmo no windows, dê preferência a versão **.zip**, para você entender melhor o processo de inicialização do servidor. A versão executável é apenas um *wrapper* para executar a JVM, já que o Tomcat é 100% Java.

O Tomcat foi por muito tempo considerado implementação padrão e referência das novas versões da API de servlets. Ele também é o servlet container padrão utilizado pelo JBoss. Ele continua em primeira posição no mercado, mas hoje tem esse lugar disputado pelo Jetty e pelo Grizzly (esse último é o servlet container que faz parte do servidor de aplicação da Oracle/Sun, o Glassfish).

Entre no diretório de instalação e execute o script `startup.sh` :

```
cd apache-tomcat<TAB>/bin  
. ./startup.sh
```

Entre no diretório de instalação do tomcat e rode o programa `shutdown.sh` :

```
cd apache-tomcat<TAB>/bin  
. ./shutdown.sh
```

Aprenderemos futuramente como iniciar o container de dentro do próprio Eclipse, por comodidade e para facilitar o uso do debug.

Tomcat no Windows

Para instalar o Tomcat no Windows basta executar o arquivo `.exe` que pode ser baixado no site do Tomcat (como falamos, dê preferência ao zip). Depois disso, você pode usar os scripts `startup.bat` e `shutdown.bat`, analogamente aos scripts do Linux.

Tudo o que vamos desenvolver neste curso funciona em qualquer ambiente compatível com o Java Enterprise Edition, seja o Linux, Windows ou Mac OS.

3.7 OUTRA OPÇÃO: JETTY

O Jetty é uma outra implementação criada pela MortBay (<http://jetty.mortbay.org>) de Servlet

Container e HTTP Server.

Pequeno e eficiente, ele é uma opção ao Tomcat bastante utilizada devido a algumas de suas características. Especialmente:

- facilmente embarcável;
- escalável;
- "plugabilidade": é fácil trocar as implementações dos principais componentes da API.

O Jetty também costuma implementar, antes do Tomcat, ideias diferentes que ainda não estão na API de servlets do Java EE. Uma dessas implementações pioneiras foi do uso dos chamados conectores NIO, por exemplo, que permitiram uma performance melhor para o uso de AJAX.

O GUJ.com.br roda com o Jetty, em uma instalação customizada que pode ser lida aqui:
<http://blog.caelum.com.br/2008/06/27/melhorando-o-guj-jetty-nio-e-load-balancing/>

3.8 INTEGRANDO O TOMCAT NO ECLIPSE

Sempre que estamos trabalhando com o desenvolvimento de uma aplicação queremos ser o mais produtivos possível, e não é diferente com uma aplicação web. Uma das formas de aumentar a produtividade do desenvolvedor é utilizar uma ferramenta que auxilie no desenvolvimento e o torne mais ágil, no nosso caso, uma IDE.

3.9 O PLUGIN WTP

O **WTP**, *Web Tools Platform*, é um conjunto de plugins para o Eclipse que auxilia o desenvolvimento de aplicações Java EE, em particular, de aplicações Web. Contém desde editores para JSP, CSS, JS e HTML até perspectivas e jeitos de rodar servidores de dentro do Eclipse.

Este plugin vai nos ajudar bastante com *content-assists* e atalhos para tornar o desenvolvimento Web mais eficiente.

A versão de download Eclipse IDE for Java EE Developers já vem por padrão com o plugin. Sendo assim, basta ir no site do Eclipse e:

- Abra a página www.eclipse.org/downloads ;
- Baixe o Eclipse IDE for Java EE Developers;
- Descompacte o arquivo e pronto.

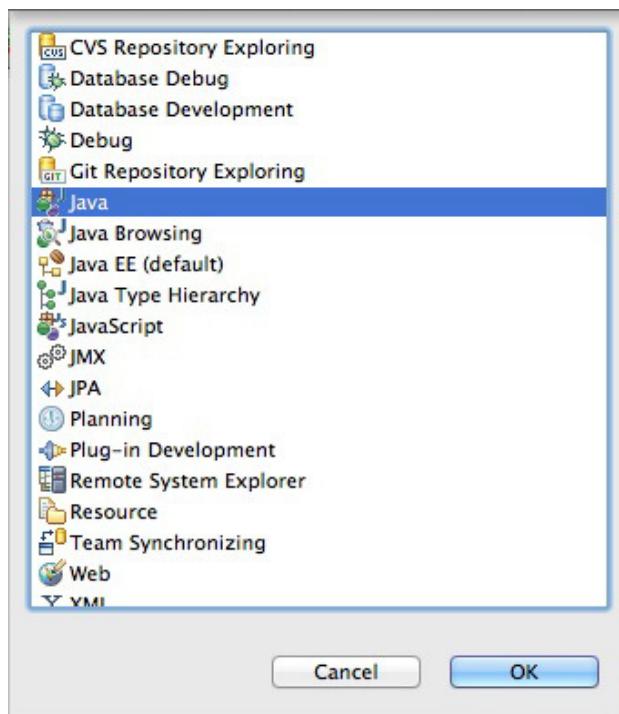
3.10 EXERCÍCIOS: CONFIGURANDO O TOMCAT NO ECLIPSE

Vamos primeiro configurar no WTP o servidor Tomcat que acabamos de descompactar.

1. Mude a perspectiva do Eclipse para **Java** (e não Java EE, por enquanto). Para isso, vá no canto direito superior e clique no botão:



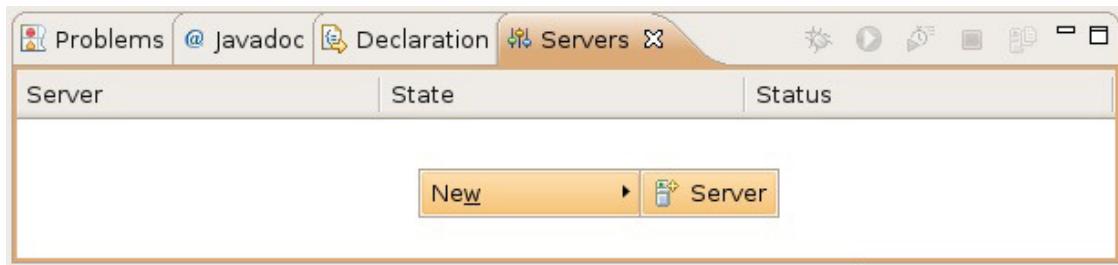
Depois selecione **Java**:



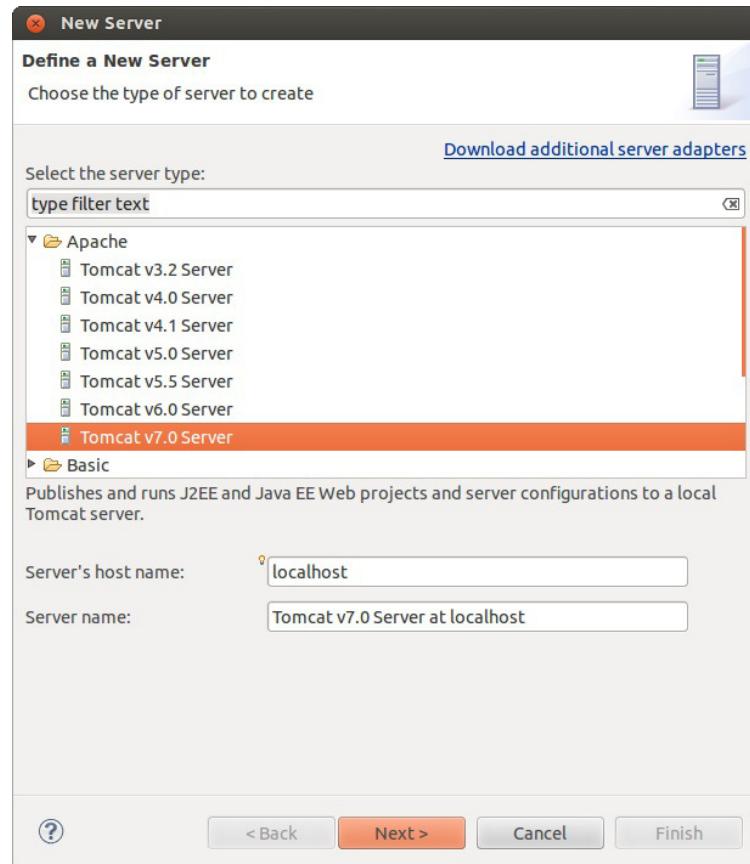
2. Abra a *View* de **Servers** na perspectiva atual. Aperte **Ctrl + 3** e digite **Servers**:



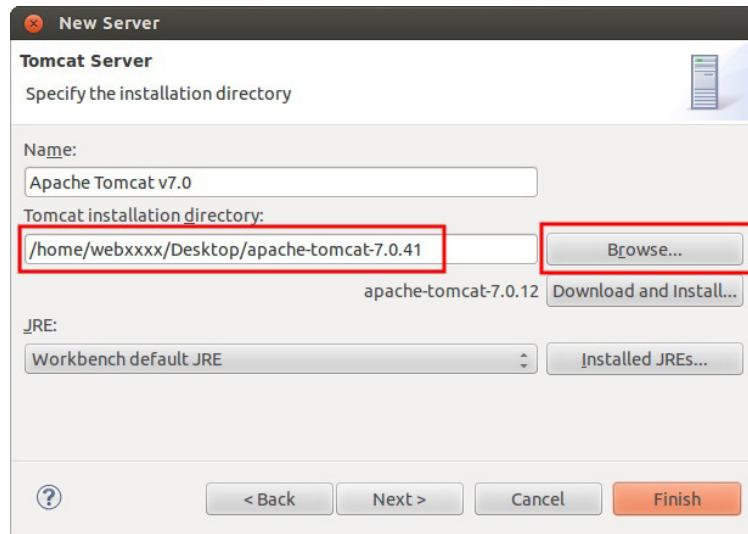
3. Clique com o botão direito dentro da aba **Servers** e vá em **New > Server**:



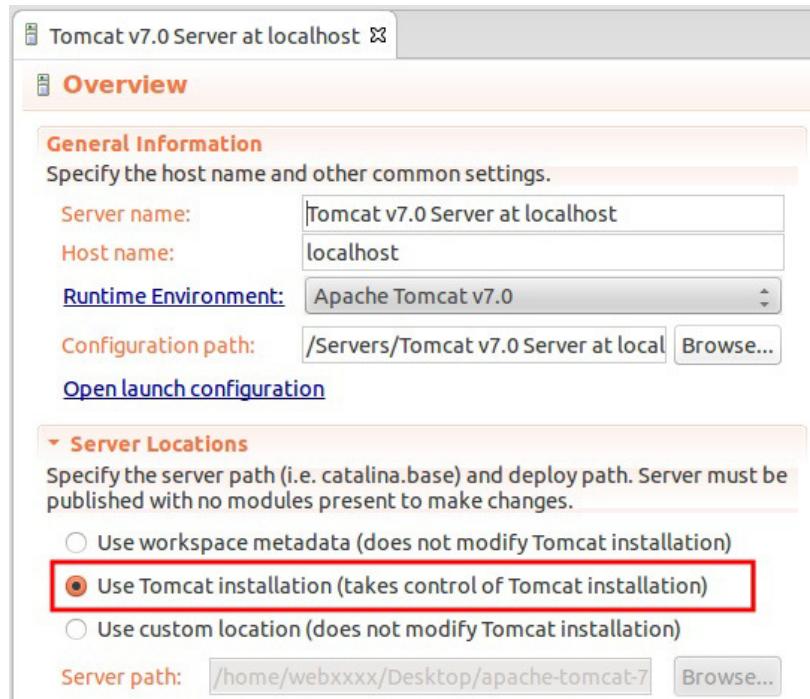
4. Selecione o **Apache Tomcat 7.0** e clique em **Next**:



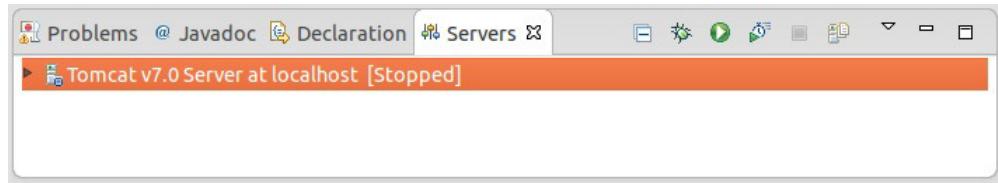
5. Na próxima tela, selecione o diretório onde você descompactou o Tomcat e clique em **Finish**:



6. Por padrão, o WTP gerencia todo o Tomcat para nós e não permite que configurações sejam feitas por fora do Eclipse. Para simplificar, vamos desabilitar isso e deixar o Tomcat no modo padrão do próprio Tomcat. Na aba Servers, dê dois cliques no servidor Tomcat que uma tela de configuração se abrirá. Localize a seção **Server Locations**. Repare que a opção *use workspace metadata* está marcada. Marque a opção **Use Tomcat installation**: Salve e feche essa tela.



7. Selecione o servidor que acabamos de adicionar e clique em **Start** (ícone play verde na view servers):



8. Abra o navegador e acesse a URL `http://localhost:8080/` Deve aparecer uma tela de mensagem do Tomcat. Pronto! O WTP está configurado para rodar com o Tomcat!

NOVO PROJETO WEB USANDO ECLIPSE

"São muitos os que usam a régua, mas poucos os inspirados." -- Platão

Ao término desse capítulo, você será capaz de:

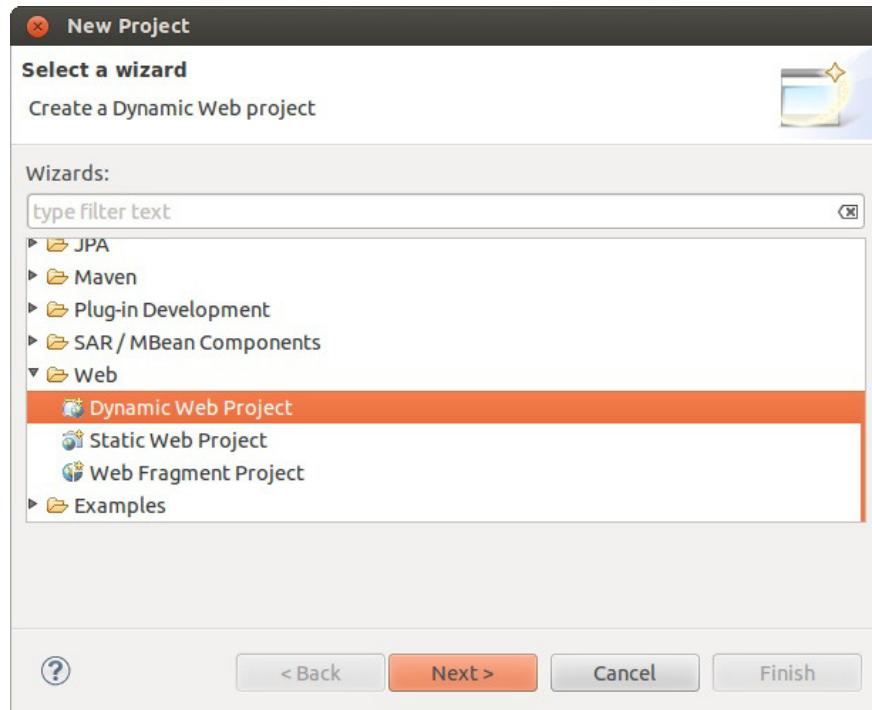
- criar um novo projeto Web no Eclipse;
- compreender quais são os diretórios importantes de uma aplicação Web;
- compreender quais são os arquivos importantes de uma aplicação Web;
- entender onde colocar suas páginas e arquivos estáticos.

4.1 NOVO PROJETO

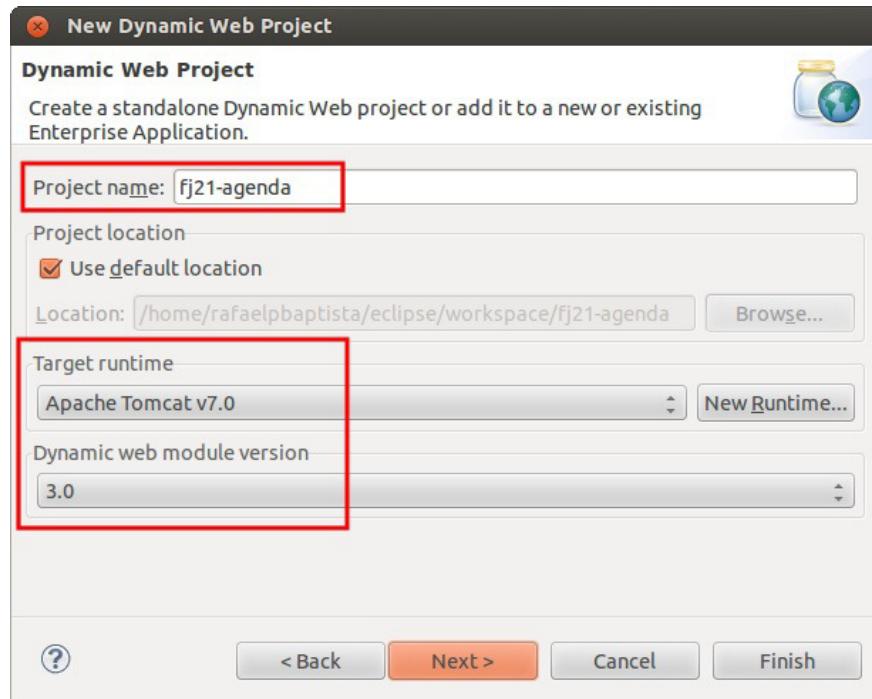
Nesse capítulo veremos como, passo a passo, criar um novo projeto Web no Eclipse usando os recursos do Eclipse JavaEE. Dessa forma não precisarmos iniciar o servlet container na mão, além de permitir a configuração de um projeto, de suas bibliotecas, e de seu debug, de uma maneira bem mais simples do que sem ele

4.2 EXERCÍCIOS: NOVO PROJETO WEB

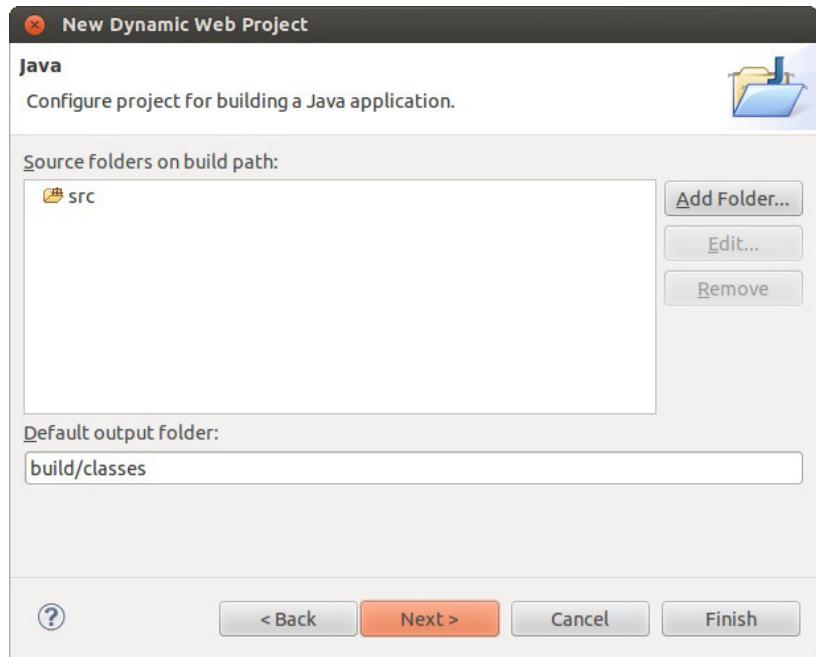
1. Vá em **New > Project** e selecione **Dynamic Web Project** e clique **Next**:



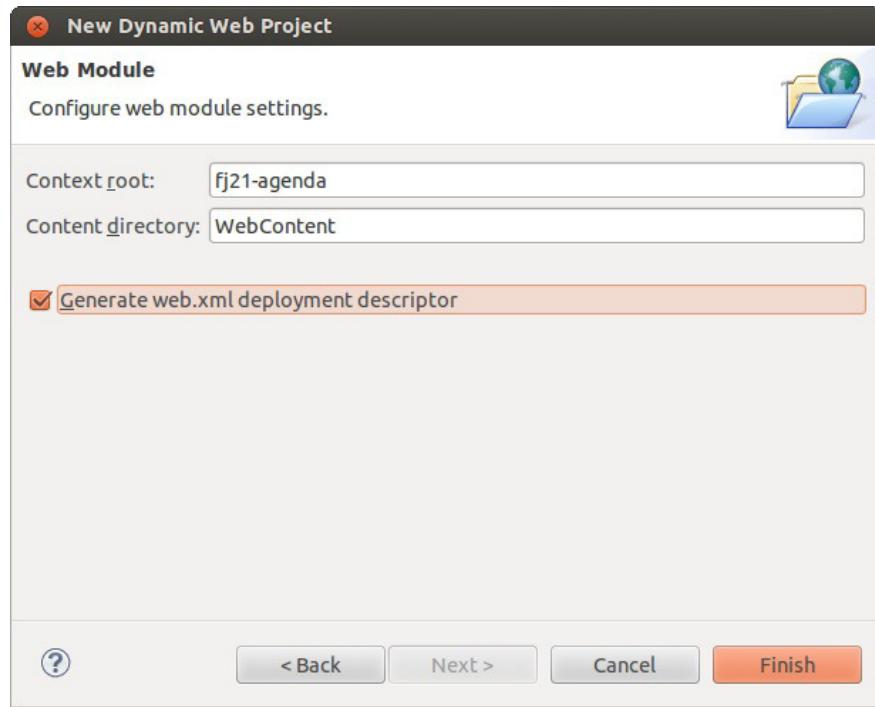
2. Coloque o nome do projeto como **fj21-agenda**, verifique se o *Target runtime* está apontando para a versão do Tomcat que acabamos de configurar e se o *Dynamic web module version* está configurado para **3.0**. Depois clique **Next**. ATENÇÃO: NÃO CLIQUE EM *Finish* AINDA!!!



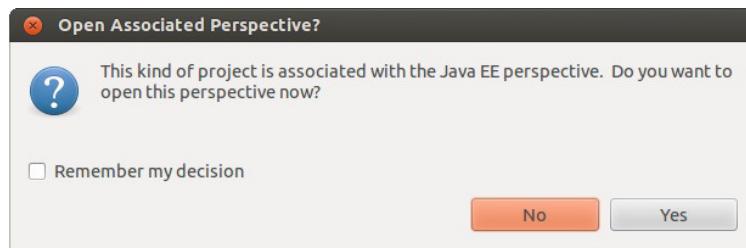
3. Clique em **Next** na configuração das pastas:



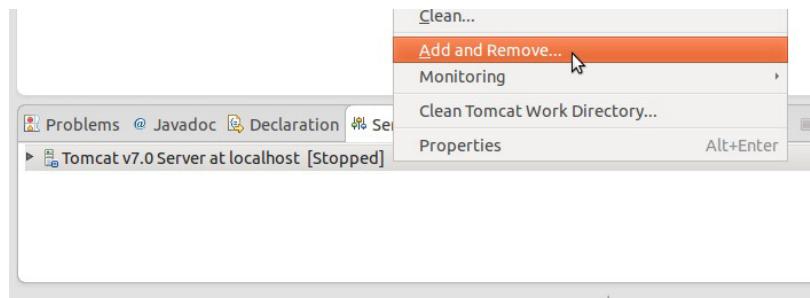
4. Na configuração *Web module* marque o box **Generate web.xml deployment descriptor** e clique em **Finish**:



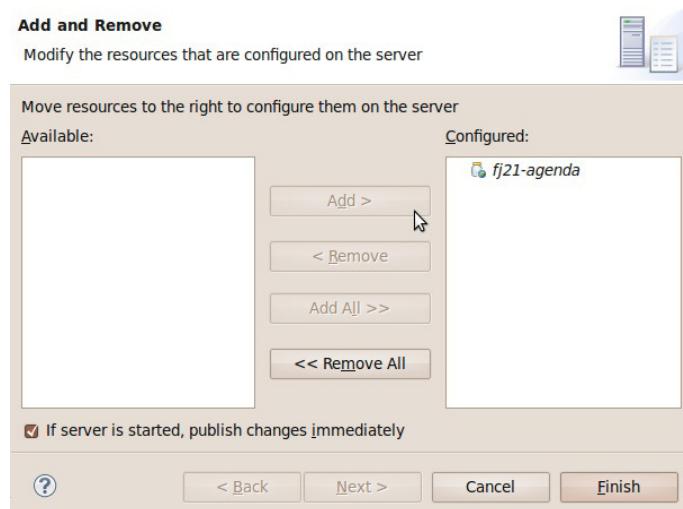
5. Se for perguntado sobre a mudança para a perspectiva Java EE, selecione **Não**.



6. O último passo é configurar o projeto para rodar no Tomcat que configuramos. Na aba **Servers**, clique com o botão direito no Tomcat e vá em **Add and Remove...**:



7. Basta selecionar o nosso projeto f21-agenda e clicar em **Add**:



8. Dê uma olhada nas pastas que foram criadas e na estrutura do nosso projeto nesse instante. Vamos analisá-la em detalhes.

4.3 ANÁLISE DO RESULTADO FINAL

Olhe bem a estrutura de pastas e verá algo parecido com o que segue:



O diretório `src` já é velho conhecido. É onde vamos colocar nosso código fonte Java. Em um projeto normal do Eclipse, essas classes seriam compiladas para a pasta `bin`, mas no Eclipse é costume se usar a pasta `build` que vemos em nosso projeto.

Nosso projeto será composto de muitas páginas Web e vamos querer acessá-lo no navegador. Já sabemos que o servidor é acessado pelo `http://localhost:8080`, mas como será que dizemos que queremos acessar o nosso projeto e não outros projetos?

No Java EE, trabalhamos com o conceito de **contextos Web** para diferenciar sites ou projetos distintos em um mesmo servidor. Na prática, é como uma *pasta virtual* que, quando acessada, remete a algum projeto em questão.

Por padrão, o Eclipse gera o **context name** com o mesmo nome do projeto; no nosso caso, **fj21-agenda**. Podemos mudar isso na hora de criar o projeto ou posteriormente indo em *Project > Properties > Web Project Settings* e mudando a opção **Context Root**. Repare que não é necessário que o nome do contexto seja o mesmo nome do projeto.

Assim, para acessar o projeto, usaremos a URL: `http://localhost:8080/fj21-agenda/`

Mas o que será que vai aparecer quando abrirmos essa URL? Será que veremos todo o conteúdo do projeto? Por exemplo, será possível acessar a pasta `src` que está dentro do nosso projeto? Tomara que não, afinal todo nosso código fonte está lá.

Para solucionar isso, uma outra configuração é importante no Eclipse: o **Content Directory**. Ao invés de abrir acesso a tudo, criamos uma pasta dentro do projeto e dizemos que ela é a raiz (root) do conteúdo a ser exibido no navegador. No Eclipse, por padrão, é criada a pasta **WebContent**, mas poderia ser qualquer outra pasta configurada na criação do projeto (outro nome comum de se usar é **web**).

Tudo que colocarmos na pasta `WebContent` será acessível na URL do projeto. Por exemplo, se

queremos uma página de boas vindas:

<http://localhost:8080/fj21-agenda/bemvindo.html>

então criamos o arquivo:

fj21-agenda/WebContent/bemvindo.html

WEB-INF

Repare também que dentro da `WebContent` há uma pasta chamada `WEB-INF`. Essa pasta é extremamente importante para qualquer projeto web Java EE. Ela contém *configurações* e *recursos* necessários para nosso projeto rodar no servidor.

O `web.xml` é o arquivo onde ficará armazenada as configurações relativas a sua aplicação, usaremos esse arquivo em breve.

Por enquanto, abra-o e veja sua estrutura, até então bem simples:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">

    <display-name>fj21-agenda</display-name>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

É o básico gerado pelo próprio Eclipse. Tudo o que ele faz é definir o nome da aplicação e a lista de arquivos acessados que vão ser procurados por padrão. Todas essas configurações são opcionais.

Repare ainda que há uma pasta chamada **lib** dentro da WEB-INF. Quase todos os projetos Web existentes precisam usar bibliotecas externas, como por exemplo o driver do MySQL no nosso caso. Copiaremos todas elas para essa pasta **lib**. Como esse diretório só aceita bibliotecas, apenas colocamos nele arquivos **.jar** ou arquivos zip com classes dentro. Caso um arquivo com outra extensão seja colocado no **lib**, ele será ignorado.

WEB-INF/lib

O diretório lib dentro do WEB-INF pode conter todas as bibliotecas necessárias para a aplicação Web, evitando assim que o classpath da máquina que roda a aplicação precise ser alterado.

Além do mais, cada aplicação Web poderá usar suas próprias bibliotecas com suas versões específicas! Você vai encontrar projetos open source que somente fornecem suporte e respondem perguntas aqueles usuários que utilizam tal diretório para suas bibliotecas, portanto evite ao máximo o uso do classpath global.

Para saber mais sobre o uso de *classloaders* e *classpath global*, você pode procurar no capítulo 2 do livro Introdução à Arquitetura e Design de Software, publicado pela Editora Casa do Código e pela Elsevier.

Há ainda um último diretório, oculto no Eclipse, o importante **WEB-INF/classes**. Para rodarmos nossa aplicação no servidor, precisamos ter acesso as classes compiladas (não necessariamente ao código fonte). Por isso, nossos `.class` são colocados nessa pasta dentro do projeto. Esse padrão é definido pela especificação de Servlets do Java EE. Repare que o Eclipse compila nossas classes na pasta **build** e depois *automaticamente* as copia para o **WEB-INF/classes**.

Note que a pasta **WEB-INF** é muito importante e contém recursos vitais para o funcionamento do projeto. Imagine se o usuário tiver acesso a essa pasta! Códigos compilados (facilmente descompiláveis), bibliotecas potencialmente sigilosas, arquivos de configuração internos contendo senhas, etc.

Para que isso não aconteça, a pasta **WEB-INF** com esse nome especial é uma **pasta invisível ao usuário final**. Isso quer dizer que se alguém acessar a URL `http://localhost:8080/fj21-agenda/WEB-INF` verá apenas uma página de erro (404).

Resumo final das pastas

- **src** - código fonte Java (`.java`)
- **build** - onde o Eclipse compila as classes (`.class`)
- **WebContent** - content directory (páginas, imagens, css etc vão aqui)
- **WebContent/WEB-INF/** - pasta oculta com configurações e recursos do projeto
- **WebContent/WEB-INF/lib/** - bibliotecas `.jar`
- **WebContent/WEB-INF/classes/** - arquivos compilados são copiados para cá

META-INF

A pasta `META-INF` é opcional mas é gerada pelo Eclipse. É onde fica o arquivo de manifesto como usado em arquivos `.jar`.

4.4 CRIANDO NOSSAS PÁGINAS E HTML BÁSICO

Para criarmos as nossas páginas, precisamos utilizar uma linguagem que consiga ser interpretada pelos navegadores. Essa linguagem é a HTML (**Hypertext Markup Language**).

Como o próprio nome diz, ela é uma linguagem de marcação, o que significa que ela é composta por tags que definem o comportamento da página que se está criando. Essas tags dizem como a página será visualizada, através da definição dos componentes visuais que aparecerão na tela. É possível exibir tabelas, parágrafos, blocos de texto, imagens e assim por diante.

Todo arquivo HTML deve conter a extensão `.html`, e seu conteúdo deve estar dentro da tag `<html>`. Em um HTML bem formado, todas as tags que são abertas também são fechadas, dessa forma, todo o seu código num arquivo `.html` ficará dentro de `<html>` e `</html>`. Além disso, podemos também definir alguns dados de cabeçalhos para nossa página, como por exemplo, o título que será colocado na janela do navegador, através das tags `<head>` e `<title>`:

```
<html>
  <head>
    <title>Título que vai aparecer no navegador</title>
  </head>
</html>
```

Para escrevermos algo que seja exibido dentro do navegador, no corpo da nossa página, basta colocarmos a tag `<body>` dentro de `<html>`, como a seguir:

```
<html>
  <body>
    Texto que vai aparecer no corpo da página
  </body>
</html>
```

4.5 EXERCÍCIOS: PRIMEIRA PÁGINA

Vamos testar nossas configurações criando um arquivo HTML de teste.

1. Crie o arquivo **WebContent/index.html** com o seguinte conteúdo:

```
<html>
  <head>
    <title>Projeto fj21-agenda</title>
  </head>
```

```

<body>
    <h1>Primeira página do projeto fj21-agenda</h1>
</body>
</html>

```

2. Inicie (ou reinicie) o Tomcat clicando no botão de *play* na aba Servers.
3. Acesse pelo navegador (nas máquinas da caelum existe um Firefox instalado):
<http://localhost:8080/fj21-agenda/index.html>

Teste também a configuração do **welcome-file** no `web.xml` : <http://localhost:8080/fj21-agenda/>

4.6 PARA SABER MAIS: CONFIGURANDO O TOMCAT SEM O PLUGIN

Se fosse o caso de criar uma aplicação web sem utilizar o plugin do Tomcat, precisaríamos colocar nossa aplicação dentro do Tomcat manualmente.

Dentro do diretório do Tomcat, há um diretório chamado *webapps*. Podemos colocar nossas aplicações dentro desse diretório, cada uma dentro de um diretório específico. O nome do diretório será o nome do contexto da aplicação.

Por exemplo, para colocar nossa aplicação *fj21-agenda* no Tomcat manualmente, basta copiar o **conteúdo** do diretório *WebContent* para um diretório chamado, por exemplo, *agenda* dentro desse diretório *webapps* do Tomcat. Pronto, o Tomcat servirá a nossa aplicação com o contexto **agenda**. Para que o Tomcat sirva nossa aplicação no contexto */*, basta colocar nossa aplicação dentro de um diretório chamado *ROOT*.

4.7 ALGUMAS TAGS HTML

Devido ao foco do curso não ser no HTML, não mostraremos a fundo as tags do HTML. No entanto, abaixo está uma lista com algumas das tags mais comuns do HTML:

- **h1 - h6** : define cabeçalhos, do **h1** ao **h6** , onde menor o número, maior a importância do cabeçalho;
- **a** : cria um link para outra página, ou para algum ponto da mesma página;
- **p** : define que o texto dentro dela estará em um parágrafo;
- **ul** : cria uma lista de elementos;
- **li** : cria um novo item numa lista de elementos;
- **input** : coloca um controle de formulário na tela (caixa de texto, botão, *radio button* etc.)
- **table** : define uma tabela;
- **tr** : colocada dentro de um **table** para definir uma linha da tabela;
- **td** : colocada dentro de um **tr** para definir uma célula;

TUTORIAL DE HTML

Para um tutorial completo sobre HTML, recomendamos uma visita ao site da Mozilla Developer Network, que possui referências das tags e exemplos de uso:
<https://developer.mozilla.org/pt-BR/docs/Web/HTML>

Você também pode visitar o tutorial de HTML da W3schools.com:
<http://www.w3schools.com/html/>

SERVLETS

"Vivemos todos sob o mesmo céu, mas nem todos temos o mesmo horizonte." -- Konrad Adenauer

Ao término desse capítulo, você será capaz de:

- fazer com que uma classe seja acessível via navegador;
- criar páginas contendo formulários;
- receber e converter parâmetros enviado por uma página;
- distinguir os métodos HTTP;
- executar suas lógicas e regras de negócio.

5.1 PÁGINAS DINÂMICAS

Quando a Web surgiu, seu objetivo era a troca de conteúdos através, principalmente, de páginas HTML estáticas. Eram arquivos escritos no formato HTML e disponibilizados em servidores para serem acessados nos navegadores. Imagens, animações e outros conteúdos também eram disponibilizados.

Mas logo se viu que a Web tinha um enorme potencial de comunicação e interação além da exibição de simples conteúdos. Para atingir esse novo objetivo, porém, páginas estáticas não seriam suficientes. Era preciso servir páginas HTML geradas dinamicamente baseadas nas requisições dos usuários.

Hoje, boa parte do que se acessa na Web (portais, blogs, home bankings etc) é baseado em conteúdo dinâmico. O usuário requisita algo ao servidor que, por sua vez, processa essa requisição e devolve uma resposta nova para o usuário.

Uma das primeiras ideias para esses "geradores dinâmicos" de páginas HTML foi fazer o servidor Web invocar um outro programa externo em cada requisição para gerar o HTML de resposta. Era o famoso **CGI** que permitia escrever pequenos programas para apresentar páginas dinâmicas usando, por exemplo, Perl, PHP, ASP e até C ou C++.

Na plataforma Java, a primeira e principal tecnologia capaz de gerar páginas dinâmicas são as **Servlets**, que surgiram no ano de 1997. Hoje, a versão mais encontrada no mercado é baseada nas versões 2.x, mais especificamente a 2.4 (parte do J2EE 1.4) e a 2.5 (parte do Java EE 5). A última versão disponível é a versão 3.0 lançada em Dezembro de 2009 com o Java EE 6, e que vem sendo gradativamente adotada no mercado.

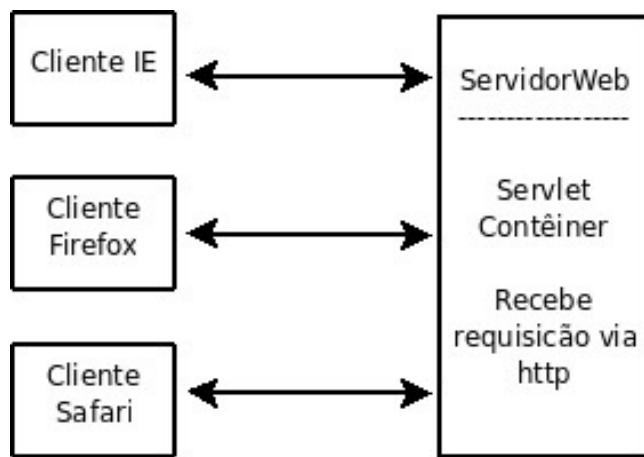
5.2 SERVLETS

As **Servlets** são a primeira forma que veremos de criar páginas dinâmicas com Java. Usaremos a própria linguagem Java para isso, criando uma classe que terá capacidade de gerar conteúdo HTML. O nome "servlet" vem da ideia de um pequeno servidor (*servidorzinho*, em inglês) cujo objetivo é receber chamadas HTTP, processá-las e devolver uma resposta ao cliente.

Uma primeira ideia da servlet seria que cada uma delas é responsável por uma página, sendo que ela lê dados da requisição do cliente e responde com outros dados (uma página HTML, uma imagem GIF etc). Como no Java tentamos sempre que possível trabalhar orientado a objetos, nada mais natural que uma servlet seja representada como um objeto a partir de uma classe Java.

Cada servlet é, portanto, um objeto Java que recebe tais requisições (**request**) e produz algo (**response**), como uma página HTML dinamicamente gerada.

O diagrama abaixo mostra três clientes acessando o mesmo servidor através do protocolo HTTP:



O comportamento das servlets que vamos ver neste capítulo foi definido na classe `HttpServlet` do pacote `javax.servlet`.

A interface `Servlet` é a que define exatamente como uma servlet funciona, mas não é o que vamos utilizar, uma vez que ela possibilita o uso de qualquer protocolo baseado em requisições e respostas, e não especificamente o HTTP.

Para escrevermos uma servlet, criamos uma classe Java que estenda `HttpServlet` e sobrescreva um método chamado `service`. Esse método será o responsável por atender requisições e gerar as respostas adequadas. Sua assinatura:

```
protected void service (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    ...
}
```

Repare que o método recebe dois objetos que representam, respectivamente, a requisição feita pelo usuário e a resposta que será exibida no final. Veremos que podemos usar esses objetos para obter informações sobre a requisição e para construir a resposta final para o usuário.

Nosso primeiro exemplo de implementação do método `service` não executa nada de lógica e apenas mostra uma mensagem estática de bem vindo para o usuário. Para isso, precisamos construir a resposta que a servlet enviará para o cliente.

É possível obter um objeto que represente a saída a ser enviada ao usuário através do método `getWriter` da variável `response`. E, a partir disso, utilizar um `PrintWriter` para imprimir algo na resposta do cliente:

```
public class OiMundo extends HttpServlet {  
    protected void service (HttpServletRequest request,  
                          HttpServletResponse response)  
        throws ServletException, IOException {  
        PrintWriter out = response.getWriter();  
  
        // escreve o texto  
        out.println("<html>");  
        out.println("<body>");  
        out.println("Primeira servlet");  
        out.println("</body>");  
        out.println("</html>");  
    }  
}
```

O único objetivo da servlet acima é exibir uma mensagem HTML simples para os usuários que a requisitarem. Mas note como seria muito fácil escrever outros códigos Java mais poderosos para gerar as Strings do HTML baseadas em informações dinâmicas vindas, por exemplo, de um banco de dados.

SERVLET X CGI

Diversas requisições podem ser feitas à mesma servlet ao mesmo tempo em um único servidor. Por isso, ela é mais rápida que um programa CGI comum que não permitia isso. A especificação de servlets cita algumas vantagens em relação ao CGI.

- Fica na memória entre requisições, não precisa ser reinstantiada;
- O nível de segurança e permissão de acesso pode ser controlado em Java;
- em CGI, cada cliente é representado por um processo, enquanto que com Servlets, cada cliente é representado por uma linha de execução.

Esse capítulo está focado na `HttpServlet`, um tipo que gera aplicações Web baseadas no protocolo HTTP, mas vale lembrar que a API não foi criada somente para este protocolo, podendo ser estendida para outros protocolos também baseados em requisições e respostas.

5.3 MAPEANDO UMA SERVLET NO WEB.XML

Acabamos de definir uma Servlet, mas como vamos acessá-la pelo navegador? Qual o endereço podemos acessar para fazermos com que ela execute? O container não tem como saber essas informações, a não ser que digamos isso para ele. Para isso, vamos fazer um mapeamento de uma URL específica para uma servlet através do arquivo `web.xml`, que fica dentro do `WEB-INF`.

Uma vez que chamar a servlet pelo pacote e nome da classe acabaria criando URLs estranhas e complexas, é comum mapear, por exemplo, uma servlet como no exemplo, chamada `OiMundo` para o nome `primeiraServlet`.

Começamos com a definição da servlet em si, dentro da tag `<servlet>`:

```
<servlet>
    <servlet-name>primeiraServlet</servlet-name>
    <servlet-class>br.com.caelum.servlet.OiMundo</servlet-class>
</servlet>
```

Em seguida, mapeie nossa servlet para a URL `/oi`. Perceba que isso acontece dentro da tag `<servlet-mapping>` (mapeamento de servlets) e que você tem que indicar que está falando daquela servlet que definimos logo acima: passamos o mesmo `servlet-name` para o mapeamento.

```
<servlet-mapping>
    <servlet-name>primeiraServlet</servlet-name>
    <url-pattern>/oi</url-pattern>
</servlet-mapping>
```

Portanto, são necessários dois passos para mapear uma servlet para uma URL:

- Definir o nome e classe da servlet;
- Usando o nome da servlet, definir a URL.

A servlet pode ser acessada através da seguinte URL:

<http://localhost:8080/fj21-agenda/oi>

Assim que o arquivo `web.xml` e a classe da servlet de exemplo forem colocados nos diretórios corretos, basta configurar o Tomcat para utilizar o diretório de base como padrão para uma aplicação Web.

Mais sobre o url-pattern

A tag `<url-pattern>` também te dá a flexibilidade de disponibilizar uma servlet através de várias URLs de um caminho, por exemplo o código abaixo fará com que qualquer endereço acessado dentro de `/oi` seja interpretado pela sua servlet:

```
<servlet-mapping>
    <servlet-name>primeiraServlet</servlet-name>
    <url-pattern>/oi/*</url-pattern>
```

```
</servlet-mapping>
```

Você ainda pode configurar "extensões" para as suas servlets, por exemplo, o mapeamento abaixo fará com que sua servlet seja chamada por qualquer requisição que termine com .php :

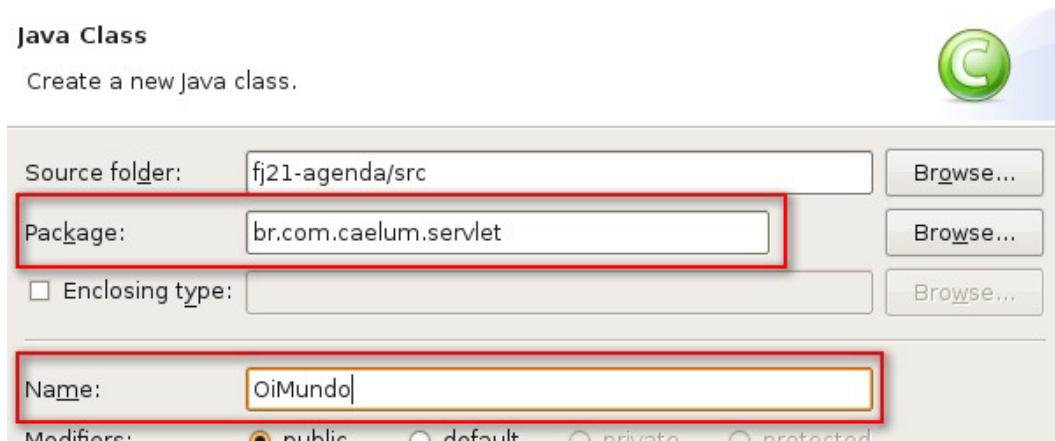
```
<servlet-mapping>
    <servlet-name>primeiraServlet</servlet-name>
    <url-pattern>*.php</url-pattern>
</servlet-mapping>
```

5.4 A ESTRUTURA DE DIRETÓRIOS

Repare que não criamos diretório nenhum na nossa aplicação (exceto o pacote para a nossa classe Servlet). Ou seja, o mapeamento da servlet não tem relação alguma com um diretório físico na aplicação. Esse mapeamento é apenas um nome atribuído, virtual, que é utilizado para acessarmos a aplicação.

5.5 EXERCÍCIOS: PRIMEIRA SERVLET

1. Crie a servlet `OiMundo` no pacote `br.com.caelum.servlet`. Escolha o menu **File, New, Class** (mais uma vez, aproveite para aprender teclas de atalho).



- Estenda `HttpServlet` :

```
public class OiMundo extends HttpServlet {
}
```

- Utilize o **CTRL+SHIFT+O** para importar `HttpServlet` .
- Para escrever a estrutura do método `service` , dentro da classe, escreva apenas `service` e dê **Ctrl+espaço**: o Eclipse gera pra você o método.

```

package br.com.caelum.servlet;

import javax.servlet.http.HttpServlet;

public class OiMundo extends HttpServlet {

    service
    }
    
```

◆ service(HttpServletRequest arg0, HttpServletResponse arg1) : void
● service(ServletRequest arg0, ServletResponse arg1) : void

ATENÇÃO: Cuidado para escolher corretamente a versão de service que recebe HttpServletRequest/Response .

A anotação @Override serve para notificar o compilador que estamos sobrescrevendo o método service da classe mãe. Se, por acaso, errarmos o nome do método ou trocarmos a ordem dos parâmetros, o compilador vai reclamar e você vai perceber o erro ainda em tempo de compilação.

O método gerado deve ser esse. **Troque os nomes dos parâmetros arg0 e arg1 como abaixo:**

```

@Override
protected void service(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
}

```

- Escreva dentro do método service sua implementação. Por enquanto, queremos apenas que nossa Servlet monte uma página HTML simples para testarmos.

Cuidado em tirar a chamada ao super.service antes e repare que a declaração do método já foi feita no passo anterior.

```

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

    PrintWriter out = response.getWriter();

    out.println("<html>");
    out.println("<head>");
    out.println("<title>Primeira Servlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Olá mundo Servlet!</h1>");
    out.println("</body>");
    out.println("</html>");
}

```

2. Abra o arquivo web.xml e clique na aba Source na parte inferior do editor de código. Dentro da tag <web-app> , mapeie a URL /oi para a servlet OiMundo . Aproveite o autocompletar do Eclipse e cuidado ao escrever o nome da classe e do pacote.

```

<servlet>
    <servlet-name>servletOiMundo</servlet-name>
    <servlet-class>
        br.com.caelum.servlet.OiMundo
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>servletOiMundo</servlet-name>
    <url-pattern>/oi</url-pattern>
</servlet-mapping>

```

3. Reinicie o Tomcat clicando no botão verde na aba Servers.



4. Teste a url <http://localhost:8080/fj21-agenda/oi>

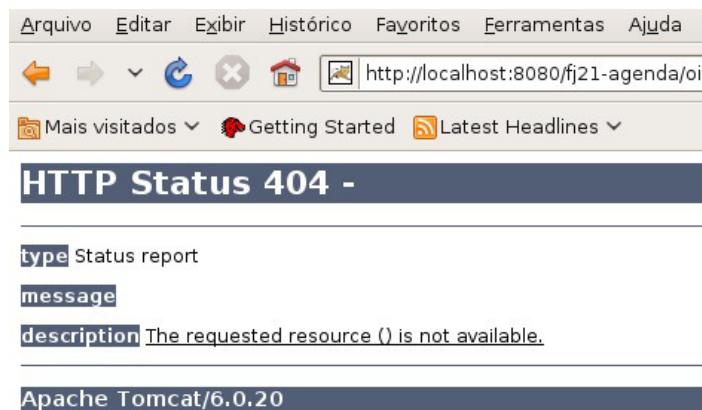


5.6 ERROS COMUNS

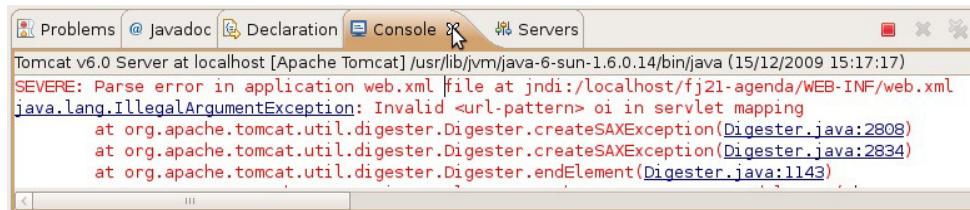
Existem diversos erros comuns nos exercícios anteriores. Aqui vão alguns deles:

- Esquecer da barra inicial no URL pattern:

```
<url-pattern>oi</url-pattern>
```

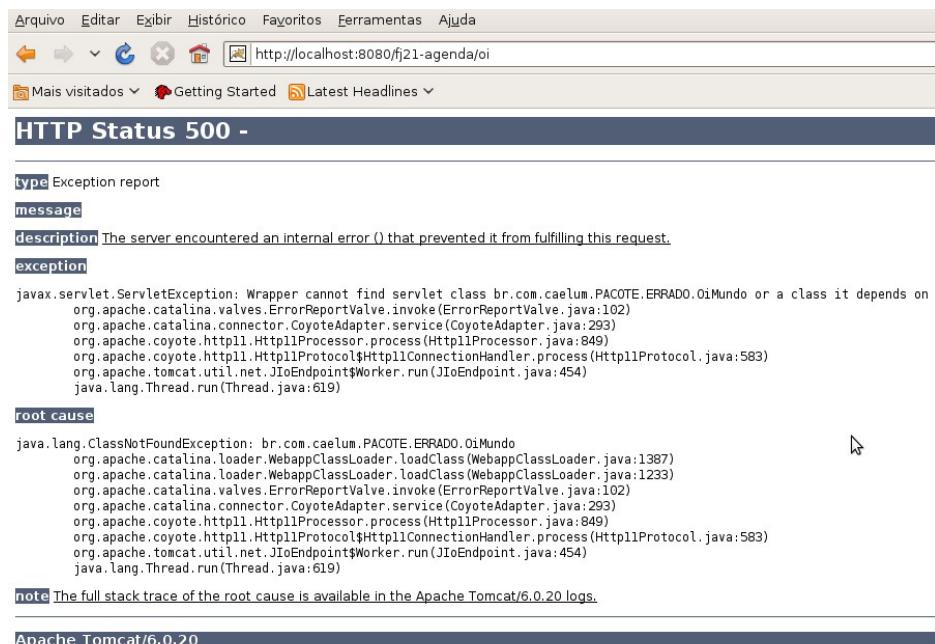


Nesse caso, uma exceção acontecerá no momento em que o tomcat for inicializado:



- Digitar errado o nome do pacote da sua servlet:

```
<servlet-class>br.caelum.servlet.OiMundo</servlet-class>
```



- Esquecer de colocar o nome da classe no mapeamento da servlet:

```
<servlet-class>br.com.caelum.servlet</servlet-class>
```

5.7 FACILIDADES DAS SERVLETS 3.0

Como foi visto no exercício anterior, criar Servlets usando o Java EE 5 é um processo muito trabalhoso. Um dos grandes problemas é que temos que configurar cada um de nossas Servlets no **web.xml** e se quisermos acessar essa servlet de maneiras diferentes, temos que criar vários mapeamentos para a mesma servlet, o que pode com o tempo tornar-se um problema devido a difícil manutenção.

Na nova especificação Servlets 3.0, que faz parte do Java EE 6, podemos configurar a maneira como vamos acessar a nossa Servlet de maneira programática, utilizando anotações simples.

De modo geral, não é mais preciso configurar as nossas Servlets no web.xml, sendo suficiente usar a anotação `@WebServlet` apenas:

```
@WebServlet("/oi")
public class OiServlet3 extends HttpServlet {
    ...
}
```

Isso é equivalente a configurar a Servlet acima com a **url-pattern** configurada como `/oi`.

Na anotação `@WebServlet`, podemos colocar ainda um parâmetro opcional chamado `name` que define um nome para a Servlet (equivalente ao `servlet-name`). Se não definirmos esse atributo, por padrão, o nome da Servlet é o nome completo da classe da sua Servlet, também conhecido como *Fully Qualified Name*.

Se quisermos que nossa Servlet seja acessado através de apenas uma URL, recomenda-se definir a URL diretamente no atributo `value` como no exemplo acima. Mas se precisarmos definir mais de uma URL para acessar a Servlet, podemos utilizar o atributo `urlPatterns` e passar um vetor de URLs:

```
@WebServlet(name = "MinhaServlet3", urlPatterns = {"/oi", "/ola"})
public class OiServlet3 extends HttpServlet{
    ...
}
```

É bom reforçar que, mesmo a Servlet estando anotado com `@WebServlet()`, ele deve obrigatoriamente realizar um *extends* a classe `javax.servlet.http.HttpServlet`.

ARQUIVO WEB.XML

Dentro da tag `<web-app>` no `web.xml`, existe um novo atributo chamado `metadata-complete`. Nesse atributo podemos configurar se nossas classes anotadas com `@WebServlet` serão procuradas pelo servidor de aplicação. Se definirmos como `true` as classes anotadas serão ignoradas.

Se não definirmos ou definirmos como `false` as classes que estiverem no `WEB-INF/classes` ou em algum `.jar` dentro `WEB-INF/lib` serão examinadas pelo servidor de aplicação.

5.8 PARA SABER MAIS: WEB SERVLET E INITPARAM ANNOTATION

Mesmo sendo uma prática questionada por alguns desenvolvedores, podemos passar parâmetros programaticamente para as Servlets na sua inicialização e sem precisar do arquivo `web.xml`. Basta usar a anotação `@WebInitParam()`, para declarar cada parâmetro no padrão chave/valor e depois passá-los dentro de um vetor para a propriedade `initParams` da anotação `@WebServlet()`:

```
@WebServlet(
```

```

        name = "OiServlet3",
        urlPatterns = {"/oi"},
        initParams = {
            @WebInitParam(name = "param1", value = "value1"),
            @WebInitParam(name = "param2", value = "value2")}
        }
    public class OiServlet3 {
        ...
    }

```

Para recuperá-los dentro da Servlet temos três estratégias:

- Usando a sobrecarga do método `init()` das Servlets:

```

// código omitido
private String parametro1;
private String parametro2;

@Override
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    this.parametro1 = config.getInitParameter("param1");
    this.parametro2 = config.getInitParameter("param2");
}

```

- Em qualquer outro método da Servlet por meio de um objeto da classe `ServletConfig`:

```

public void service(HttpServletRequest request,
                     HttpServletResponse response) throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<h2>Exemplo com InitParam Servlet</h2>");

    ServletConfig config = getServletConfig();

    String parametro1= config.getInitParameter("param1");
    out.println("Valor do parâmetro 1: " + parametro1);

    String parametro2 = config.getInitParameter("param2");
    out.println("<br>Valor do parâmetro 1: " + parametro2);

    out.close();
}

```

- Ou usando o método `getServletConfig()` com `getInitParameter()` direto na opção de saída:

```

out.println("Valor do parâmetro 1: "
+ getServletConfig().getInitParameter("param1"));

```

5.9 ENVIANDO PARÂMETROS NA REQUISIÇÃO

Ao desenvolver uma aplicação Web, sempre precisamos realizar operações no lado do servidor, com dados informados pelo usuário, seja através de formulários ou seja através da URL.

Por exemplo, para gravarmos um contato no banco de dados, precisamos do nome, e-mail, endereço e a data de nascimento dele. Temos uma página com um formulário que o usuário possa preencher e ao clicar em um botão esses dados devem, de alguma forma, ser passados para um Servlet. Já sabemos que a Servlet responde por uma determinada URL (através do *url-pattern/v2.5* ou do *urlPatterns/v3.0*), portanto, só precisamos indicar que ao clicar no botão devemos enviar uma requisição para essa Servlet.

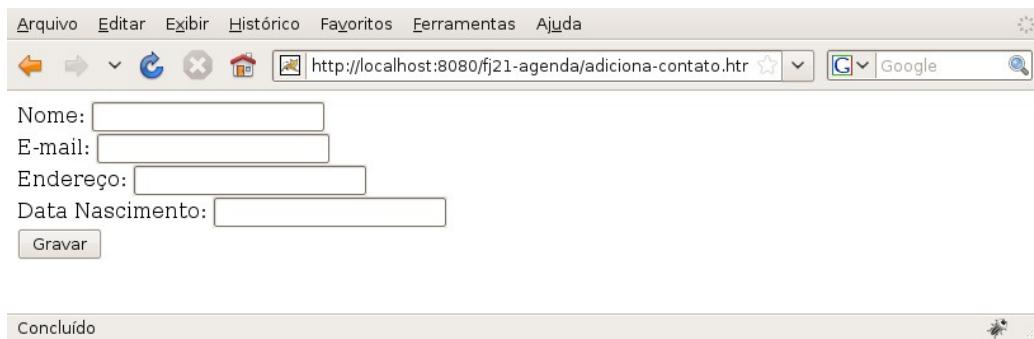
Para isso, vamos criar uma página HTML, chamada `adiciona-contato.html`, contendo um formulário para preenchermos os dados dos contatos:

```
<html>
    <body>
        <form action="adicionaContato">
            Nome: <input type="text" name="nome" /><br />
            E-mail: <input type="text" name="email" /><br />
            Endereço: <input type="text" name="endereco" /><br />
            Data Nascimento: <input type="text" name="dataNascimento" /><br />

            <input type="submit" value="Gravar" />
        </form>
    </body>
</html>
```

Esse código possui um formulário, determinado pela tag `<form>`. O atributo `action` indica qual endereço deve ser chamado ao submeter o formulário, ao clicar no botão *Gravar*. Nesse caso, estamos apontando o `action` para um endereço que será uma `Servlet` que já vamos criar.

Ao acessar a página `adiciona-contato.html`, o resultado deverá ser similar à figura abaixo:



5.10 PEGANDO OS PARÂMETROS DA REQUISIÇÃO

Para recebermos os valores que foram preenchidos na tela e submetidos, criaremos uma `Servlet`, cuja função será receber de alguma maneira esses dados e convertê-los, se necessário.

Dentro do método `service` da nossa `Servlet` para adição de contatos, vamos buscar os dados que foram enviados na **requisição**. Para buscarmos esses dados, precisamos utilizar o parâmetro `request` do método `service` chamando o método `getParameter("nomeDoParametro")`, onde o nome do parâmetro é o mesmo nome do `input` que você quer buscar o valor. Isso vai retornar uma `String` com o valor do parâmetro. Caso não exista o parâmetro, será retornado `null`:

```
String valorDoParametro = request.getParameter("nomeDoParametro");
```

O fato de ser devolvida uma `String` nos traz um problema, pois, a data de nascimento do contato está criada como um objeto do tipo `Calendar`. Então, o que precisamos fazer é **converter** essa `String` em um objeto `Calendar`. Mas a API do Java para trabalhar com datas não nos permite fazer isso diretamente. Teremos que converter antes a `String` em um objeto do tipo `java.util.Date` com auxílio da classe `SimpleDateFormat`, utilizando o método `parse`, da seguinte forma:

```
String dataEmTexto = request.getParameter("dataNascimento");
Date date = new SimpleDateFormat("dd/MM/yyyy").parse(dataEmTexto);
```

Repare que indicamos também o *pattern* (formato) com que essa data deveria chegar para nós, através do parâmetro passado no construtor de `SimpleDateFormat` com o valor `dd/MM/yyyy`. Temos que tomar cuidado pois o método `parse` lança uma exceção do tipo `ParseException`. Essa exceção indica que o que foi passado na data não pôde ser convertido ao pattern especificado. Com o objeto do tipo `java.util.Date` que foi devolvido, queremos criar um `Calendar`. Para isso vamos usar o método `setTime` da classe `Calendar`, que recebe um `Date`.

```
dataNascimento = Calendar.getInstance();
dataNascimento.setTime(date);
```

Vamos utilizar também o nosso DAO para gravar os contatos no banco de dados. No final, a nossa Servlet ficará da seguinte forma:

```
@WebServlet("/adicionaContato")
public class AdicionaContatoServlet extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
                           throws IOException, ServletException {

        PrintWriter out = response.getWriter();

        // pegando os parâmetros do request
        String nome = request.getParameter("nome");
        String endereco = request.getParameter("endereco");
        String email = request.getParameter("email");
        String dataEmTexto = request.getParameter("dataNascimento");
        Calendar dataNascimento = null;

        // fazendo a conversão da data
        try {
            Date date = new SimpleDateFormat("dd/MM/yyyy")
                .parse(dataEmTexto);
            dataNascimento = Calendar.getInstance();
            dataNascimento.setTime(date);
        } catch (ParseException e) {
            out.println("Erro de conversão da data");
            return; //para a execução do método
        }

        // monta um objeto contato
        Contato contato = new Contato();
        contato.setNome(nome);
        contato.setEndereco(endereco);
        contato.setEmail(email);
        contato.setDataNascimento(dataNascimento);
```

```

// salva o contato
ContatoDao dao = new ContatoDao();
dao.adiciona(contato);

// imprime o nome do contato que foi adicionado
out.println("<html>");
out.println("<body>");
out.println("Contato " + contato.getNome() +
" adicionado com sucesso");
out.println("</body>");
out.println("</html>");
}
}

```

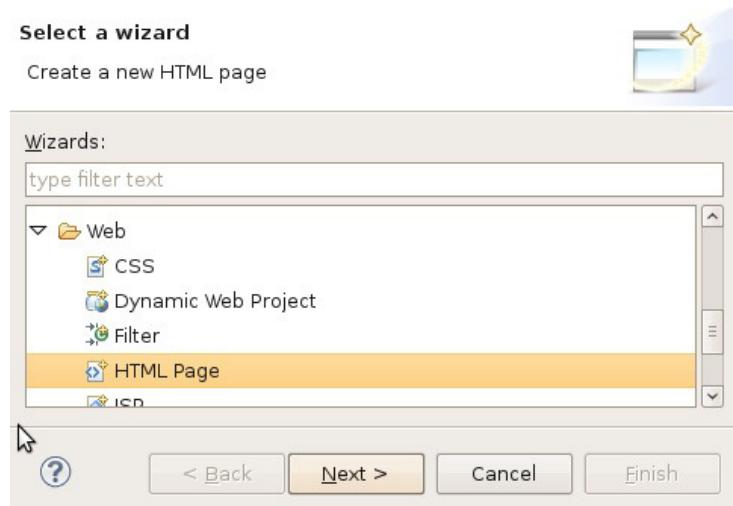
5.11 EXERCÍCIOS: CRIANDO FUNCIONALIDADE PARA GRAVAR CONTATOS

- Como vamos precisar gravar contatos, precisaremos das classes para trabalhar com banco de dados que criamos no capítulo de JDBC. Para isso, deixamos disponível um arquivo zip contendo as classes necessárias que criamos anteriormente.
 - No Eclipse, selecione o projeto **fj21-agenda** e vá no menu **File -> Import**
 - Dentro da janela de Import, escolha **General -> Archive File** e clique em **Next**:
 - No campo **From archive file** clique em **Browse**, selecione o arquivo **Desktop/21/dao-modelo.zip** e clique em **Finish**

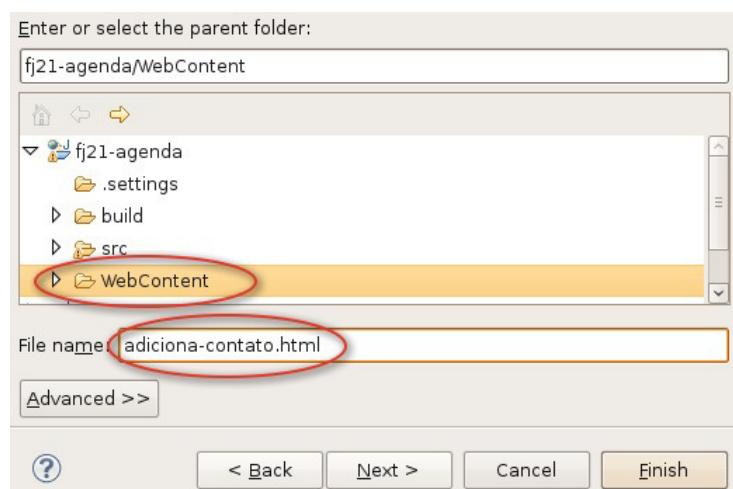
EM CASA

Caso você esteja fazendo em casa, você pode usar exatamente as mesmas classes criadas durante os exercícios do capítulo de JDBC, mas terá que fazer uma mudança na classe ConnectionFactory (volte ao capítulo 2, no box sobre o Class.forName. Não esqueça de copiar também o Driver do MySQL.

- Temos que criar a página que permitirá aos usuários cadastrar os contatos
 - Vá no menu **File -> New -> Other**.
 - Escolha **Web -> HTML Page ou HTML File** e clique **Next**:



- Chame o arquivo de **adiciona-contato.html** e clique em **Finish** (garanta que o arquivo esteja dentro do diretório *WebContent*):



- Esse arquivo HTML deverá ter o seguinte conteúdo (**cuidado com o nome dos inputs**):

```

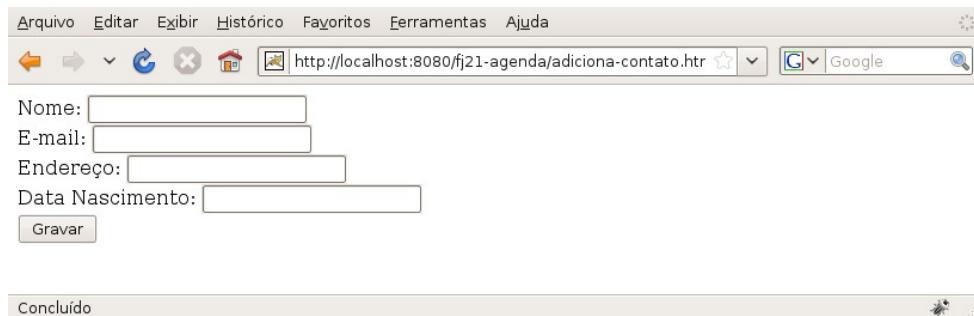
<html>
  <body>
    <h1>Adiciona Contatos</h1>
    <hr />
    <form action="adicionaContato">
      Nome: <input type="text" name="nome" /><br />
      E-mail: <input type="text" name="email" /><br />
      Endereço: <input type="text" name="endereco" /><br />
      Data Nascimento:
        <input type="text" name="dataNascimento" /><br />

        <input type="submit" value="Gravar" />
    </form>
  </body>
</html>

```

- Acesse no navegador o endereço:

<http://localhost:8080/fj21-agenda/adiciona-contato.html>



3. Precisamos criar a `Servlet` que gravará o contato no banco de dados:

- Crie uma nova `Servlet` no pacote `br.com.caelum.agenda.servlet` chamado `AdicionaContatoServlet` com o seguinte código.

Cuidado ao implementar essa classe, que é grande e complicada.

Use o `Ctrl+Shift+O` para ajudar nos imports. A classe `Date` deve ser de `java.util` e a classe `ParseException`, de `java.text`.

```
@WebServlet("/adicionaContato")
public class AdicionaContatoServlet extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
                           throws IOException, ServletException {
        // busca o writer
        PrintWriter out = response.getWriter();

        // buscando os parâmetros no request
        String nome = request.getParameter("nome");
        String endereco = request.getParameter("endereco");
        String email = request.getParameter("email");
        String dataEmTexto = request
            .getParameter("dataNascimento");
        Calendar dataNascimento = null;

        // fazendo a conversão da data
        try {
            Date date =
                new SimpleDateFormat("dd/MM/yyyy")
                .parse(dataEmTexto);
            dataNascimento = Calendar.getInstance();
            dataNascimento.setTime(date);
        } catch (ParseException e) {
            out.println("Erro de conversão da data");
            return; //para a execução do método
        }

        // monta um objeto contato
        Contato contato = new Contato();
        contato.setNome(nome);
```

```

        contato.setEndereco(endereco);
        contato.setEmail(email);
        contato.setDataNascimento(dataNascimento);

        // salva o contato
        ContatoDao dao = new ContatoDao();
        dao.adiciona(contato);

        // imprime o nome do contato que foi adicionado
        out.println("<html>");
        out.println("<body>");
        out.println("Contato " + contato.getNome() +
                   " adicionado com sucesso");
        out.println("</body>");
        out.println("</html>");
    }
}

```

UTILIZANDO A SERVLET V2.5

Se ainda estivéssemos utilizando a versão 2.5 da Servlet, precisaríamos fazer o seguinte mapeamento no **web.xml**:

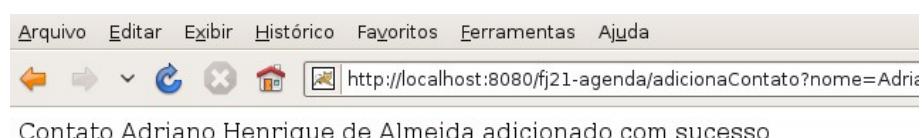
```

<servlet>
    <servlet-name>AdicionaContato</servlet-name>
    <servlet-class>
        br.com.caelum.agenda.servlet.AdicionaContatoServlet
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>AdicionaContato</servlet-name>
    <url-pattern>/adicionaContato</url-pattern>
</servlet-mapping>

```

- Reinicie o servidor, para que a nova Servlet seja reconhecido
- Acesse novamente no navegador a URL <http://localhost:8080/fj21-agenda/adiciona-contato.html>
- Preencha o formulário e clique em Gravar. O resultado deve ser semelhante à imagem a seguir:



- Verifique no banco de dados se o dado realmente foi adicionado com sucesso.

5.12 GET, POST E MÉTODOS HTTP

Repare que no exercício anterior, ao clicarmos no botão salvar, todos os dados que digitamos no formulário aparecem na URL da página de sucesso. Isso acontece porque não definimos no nosso formulário a forma com que os dados são enviados para o servidor, através do atributo `method` para o `<form>` da seguinte forma:

```
<form action="adicionaContato" method="POST">
```

Como não tínhamos definido, por padrão então é usado o método GET, que indica que os valores dos parâmetros são passados através da URL junto dos nomes dos mesmos, separados por &, como em:
`nome=Adriano&email=adriano@caelum.com.br`

Podemos também definir o método para `POST` e, dessa forma, os dados são passados dentro do corpo do protocolo HTTP, sem aparecer na URL que é mostrada no navegador.

Podemos, além de definir no formulário como os dados serão passados, também definir quais métodos HTTP nossa servlet aceitará.

O método `service` aceita todos os métodos HTTP, portanto, tanto o método GET quanto o POST. Para especificarmos como trataremos cada método, temos que escrever os métodos `doGet` e/ou `doPost` na nossa servlet:

```
void doGet(HttpServletRequest req, HttpServletResponse res);  
void doPost(HttpServletRequest req, HttpServletResponse res);
```

OUTROS MÉTODOS HTTP

Além do GET e do POST, o protocolo HTTP possui ainda mais 7 métodos: PUT, DELETE, HEAD, TRACE, CONNECT, OPTIONS e PATCH.

Muitas pessoas conhecem apenas o GET e POST, pois, são os únicos que HTML 4 suporta.

5.13 TRATANDO EXCEÇÕES DENTRO DA SERVLET

O que será que vai acontecer se algum SQL do nosso DAO contiver erro de sintaxe e o comando não puder ser executado? Será que vai aparecer uma mensagem agradável para o usuário?

Na verdade, caso aconteça um erro dentro da nossa `Servlet` a `stacktrace` da exceção ocorrida será mostrada em uma tela padrão do container. O problema é que para o usuário comum, a mensagem de erro do Java não fará o menor sentido. O ideal seria mostrarmos uma página de erro dizendo: "Um erro ocorreu" e com informações de como notificar o administrador.

Para fazermos isso, basta configurarmos nossa aplicação dizendo que, caso aconteça uma *Exception*, uma página de erro deverá ser exibida. Essa configuração é feita no **web.xml**, com a seguinte declaração:

```
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/erro.html</location>
</error-page>
```

Além de tratarmos as exceções que podem acontecer na nossa aplicação, podemos também tratar os códigos de erro HTTP, como por exemplo, 404, que é o erro dado quando se acessa uma página inexistente. Para isso basta fazermos a declaração no web.xml:

```
<error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
</error-page>
```

WRAPPING EM SERVLETException

Caso aconteça uma exceção que seja do tipo checada (não filha de `RuntimeException`), também teríamos que repassá-la para container. No entanto, o método `service` só nos permite lançar `ServletException` e `IOException`.

Para podermos lançar outra exceção checked, precisamos escondê-la em uma `ServletException`, como a seguir:

```
try {
    // código que pode lançar SQLException
} catch (SQLException e) {
    throw new ServletException(e);
}
```

Essa técnica é conhecida como *wrapping de exceptions*. O container, ao receber a `ServletException`, vai desembrulhar a exception interna e tratá-la.

5.14 EXERCÍCIO: TRATANDO EXCEÇÕES E CÓDIGOS HTTP

1. Vamos criar uma página para mostrar a mensagem genérica de tratamento:

- Crie um novo HTML chamado **erro.html** com o seguinte conteúdo:

```
<html>
    <body>
        Um erro ocorreu!
    </body>
</html>
```

- **Adicione** a declaração da página de erro no **web.xml**:

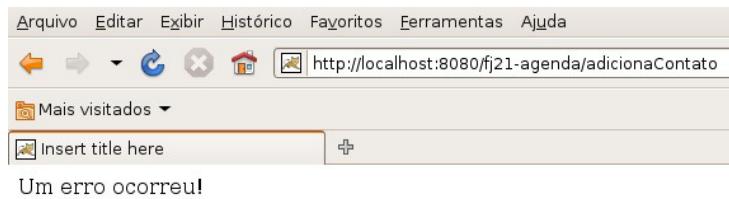
```
<error-page>
```

```

<exception-type>java.lang.Exception</exception-type>
<location>/erro.html</location>
</error-page>

```

- Altere o usuário de acesso ao banco na classe ConnectionFactory de root para algum outro usuário que não exista, por exemplo, toor .
- Reinicie o servidor, para que as alterações tenham efeito
- Acesse no navegador a URL <http://localhost:8080/fj21-agenda/adiciona-contato.html>
- Preencha o formulário e clique em Gravar, o resultado deve ser semelhante à imagem a seguir:



Altere novamente o usuário de acesso ao banco na classe ConnectionFactory para root .

2. Vamos criar uma página para ser exibida quando o usuário acessar algo inexistente:

- Crie um novo HTML chamado **404.html** com o seguinte conteúdo:

```

<html>
  <body>
    A página acessada não existe.
  </body>
</html>

```

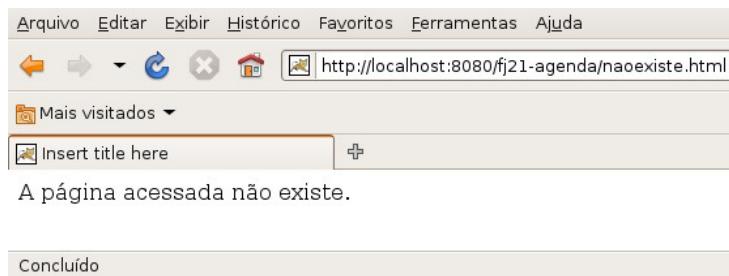
- **Adicione** a declaração da página no `web.xml` :

```

<error-page>
  <error-code>404</error-code>
  <location>/404.html</location>
</error-page>

```

- Reinicie novamente o servidor;
- Acesse no navegador uma URL inexistente no projeto, por exemplo, <http://localhost:8080/fj21-agenda/naoexiste.html>:



5.15 INIT E DESTROY

Toda Servlet deve possuir um construtor sem argumentos para que o container possa criá-lo. Após a criação, o *servlet container* inicializa a Servlet com o método `init(ServletConfig config)` e o usa durante todo o seu período ativo, até que vai desativá-lo através do método `destroy()`, para então liberar o objeto.

É importante perceber que a sua Servlet será instanciado uma única vez pelo container e esse único objeto será usado para atender a todas as requisições de todos os clientes em *threads* separadas. Aliás, é justo isso que traz uma melhoria em relação aos CGI comuns que disparavam diversos processos.

Na inicialização de uma Servlet, quando parâmetros podem ser lidos e variáveis comuns a todas as requisições devem ser inicializadas, é um bom momento, por exemplo, para carregar arquivos diversos de configurações da aplicação:

```
void init (ServletConfig config);
```

Na finalização, devemos liberar possíveis recursos que estejamos segurando:

```
void destroy();
```

Os métodos `init` e `destroy`, quando reescritos, são obrigados a chamar o `super.init()` e `super.destroy()` respectivamente. Isso acontece pois um método é diferente de um construtor. Quando estendemos uma classe e criamos o nosso próprio construtor da classe filha, ela chama o construtor da classe pai sem argumentos, preservando a garantia da chamada de um construtor. O mesmo não acontece com os métodos.

Supondo que o método `init` (ou `destroy`) executa alguma tarefa fundamental em sua classe pai, se você esquecer de chamar o `super`, terá problemas.

O exemplo a seguir mostra uma Servlet implementando os métodos de inicialização e finalização. Os métodos `init` e `destroy` podem ser bem simples (lembre-se que são opcionais):

```
@WebServlet("/minhaServlet")
public class MinhaServlet extends HttpServlet {

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        log("Iniciando a servlet");
    }

    public void destroy() {
        super.destroy();
        log("Destruindo a servlet");
    }

    protected void service(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException {
        //código do seu método service
    }
}
```

```
}
```

5.16 UMA ÚNICA INSTÂNCIA DE CADA SERVLET

De acordo com a especificação de Servlets, por padrão, existe uma única instância de cada Servlet declarada. Ao chegar uma requisição para a Servlet, uma nova Thread é aberta sobre aquela instância que já existe.

Isso significa que, se colocássemos em nossa Servlet uma variável de instância, ela seria compartilhada entre todas as threads que acessam essa Servlet! Em outras palavras, seria compartilhado entre todas as requisições e todos os clientes enxergariam o mesmo valor. Provavelmente não é o que queremos fazer.

Um exemplo simples para nos auxiliar enxergar isso é uma Servlet com uma variável para contar a quantidade de requisições:

```
@WebServlet("/contador")
public class Contador extends HttpServlet {
    private int contador = 0; //variavel de instancia

    protected void service(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        contador++; // a cada requisicao a mesma variavel é incrementada

        // recebe o writer
        PrintWriter out = response.getWriter();

        // escreve o texto
        out.println("<html>");
        out.println("<body>");
        out.println("Contador agora é: " + contador);
        out.println("</body>");
        out.println("</html>");
    }
}
```

Quando a Servlet for inicializada, o valor do contador é definido para 0 (zero). Após isso, a cada requisição que é feita para essa Servlet, devido ao fato da instância ser sempre a mesma, a variável utilizada para incrementar será sempre a mesma, e por consequência imprimirá o número atual para o contador.

Sabemos que compartilhar variáveis entre múltiplas Threads pode nos trazer problemas graves de concorrência. Se duas threads (no caso, duas requisições) modificarem a mesma variável ao "mesmo tempo", podemos ter perda de informações mesmo em casos simples como o do contador acima.

Há duas soluções para esse problema. A primeira seria impedir que duas threads accessem ao mesmo tempo o mesmo objeto crítico; para isso, podemos sincronizar o método service. Mas isso traria muitos problemas de escalabilidade (apenas uma pessoa por vez poderia requisitar minha página). A outra solução, mais simples, é apenas não compartilhar objetos entre threads.

Quando se fala de Servlets, a boa prática diz para **evitar usar atributos compartilhados**.

5.17 EXERCÍCIOS OPCIONAIS

1. Implemente os códigos das seções anteriores sobre ciclo de vida e concorrência em Servlets. Faça a classe `Contador` e use também os métodos `init` e `destroy`. O objetivo é ver na prática os conceitos discutidos.

5.18 DISCUSSÃO: CRIANDO PÁGINAS DENTRO DE UMA SERVLET

Imagine se quiséssemos listar os nossos contatos, como poderíamos fazer? Como até o momento só conhecemos `Servlet`, provavelmente nossa sugestão seria criarmos uma `Servlet` que faça toda a listagem através de `out.println()`. Mas, será que a manutenção disso seria agradável? E se um dia precisarmos adicionar uma coluna nova na tabela? Teríamos que recompilar classes, e colocarmos a atualização no ar.

Com o decorrer do curso aprenderemos que essa não é a melhor forma de fazermos essa funcionalidade.

JAVASERVER PAGES

"O maior prazer é esperar pelo prazer." -- Gotthold Lessing

Nesse capítulo, você aprenderá:

- O que é JSP;
- Suas vantagens e desvantagens;
- Escrever arquivos JSP com scriptlets;
- Usar Expression Language.

6.1 COLOCANDO O HTML NO SEU DEVIDO LUGAR

Até agora, vimos que podemos escrever conteúdo dinâmico através de `Servlets`. No entanto, se toda hora criarmos `Servlets` para fazermos esse trabalho, teremos muitos problemas na manutenção das nossas páginas e também na legibilidade do nosso código, pois sempre aparece código Java misturado com código HTML. Imagine todo um sistema criado com `Servlets` fazendo a geração do HTML.

Para não termos que criar todos os nossos conteúdos dinâmicos dentro de classes, misturando fortemente HTML com código Java, precisamos usar uma tecnologia que podemos usar o HTML de forma direta, e que também vá possibilitar a utilização do Java. Algo similar ao ASP e PHP.

Essa tecnologia é o **JavaServer Pages** (JSP). O primeiro arquivo JSP que vamos criar é chamado **bemvindo.jsp**. Esse arquivo poderia conter simplesmente código HTML, como o código a seguir:

```
<html>
  <body>
    Bem vindo
  </body>
</html>
```

Assim, fica claro que uma página JSP nada mais é que um arquivo baseado em HTML, com a extensão `.jsp`.

Dentro de um arquivo JSP podemos escrever também código Java, para que possamos adicionar comportamento dinâmico em nossas páginas, como declaração de variáveis, condicionais (`if`), loops (`for`, `while`) entre outros.

Portanto, vamos escrever um pouco de código Java na nossa primeira página. Vamos declarar uma variável do tipo `String` e inicializá-la com algum valor.

```
<%
String mensagem = "Bem vindo!";
%>
```

Para escrever código Java na sua página, basta escrevê-lo entre as tags `<%` e `%>`. Esse tipo de código é chamado de **scriptlet**.

SCRIPTLET

Scriptlet é o código escrito entre `<%` e `%>`. Esse nome é composto da palavra *script* (pedaço de código em linguagem de script) com o sufixo *let*, que indica algo pequeno.

Como você já percebeu, a Sun possui essa mania de colocar o sufixo *let* em seus produtos como os *scriptlets*, *servlets*, *portlets*, *midlets*, *applets* etc...

Podemos avançar mais um pouco e utilizar uma das variáveis já implícitas no JSP: todo arquivo JSP já possui uma variável chamada `out` (do tipo `JspWriter`) que permite imprimir para o `response` através do método `println`:

```
<% out.println(nome); %>
```

A variável `out` é um objeto implícito na nossa página JSP e existem outras de acordo com a especificação. Repare também que sua funcionalidade é semelhante ao `out` que utilizávamos nas *Servlets* mas sem precisarmos declará-lo antes.

Existem ainda outras possibilidades para imprimir o conteúdo da nossa variável: podemos utilizar um atalho (muito parecido, ou igual, a outras linguagens de *script* para a Web):

```
<%= nome %><br />
```

Isso já é o suficiente para que possamos escrever o nosso primeiro JSP.

COMENTÁRIOS

Os comentários em uma página JSP devem ser feitos como o exemplo a seguir:

```
<%-- comentário em jsp --%>
```

6.2 EXERCÍCIOS: PRIMEIRO JSP

- Crie o arquivo **WebContent/bemvindo.jsp** com o seguinte conteúdo:

```
<html>
    <body>
        <%-- comentário em JSP aqui: nossa primeira página JSP --%>

        <%
            String mensagem = "Bem vindo ao sistema de agenda do FJ-21!";
        %>
        <% out.println(mensagem); %>

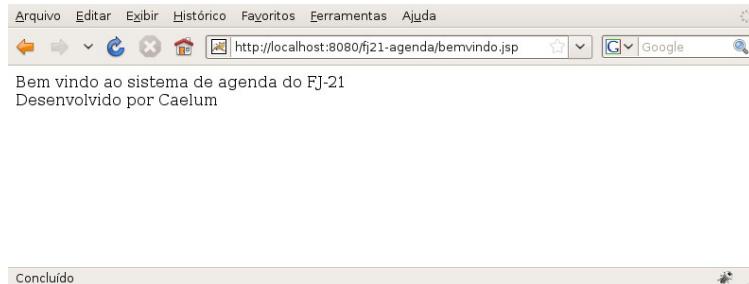
        <br />

        <%
            String desenvolvido = "Desenvolvido por (SEU NOME AQUI)";
        %>
        <%= desenvolvido %>

        <br />

        <%
            System.out.println("Tudo foi executado!");
        %>
    </body>
</html>
```

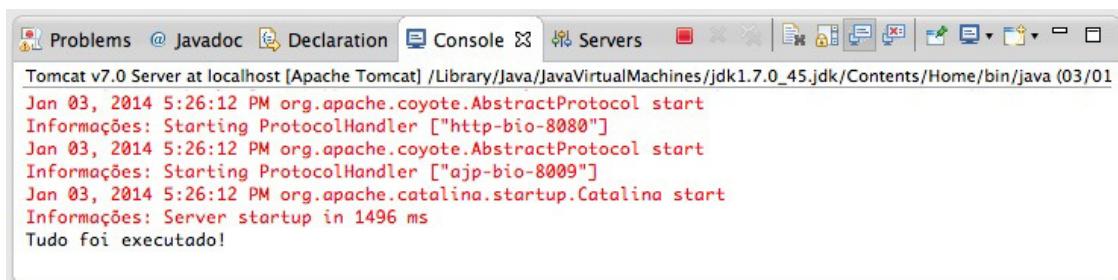
- Acesse a URL <http://localhost:8080/fj21-agenda/bemvindo.jsp> no navegador



- Onde apareceu a mensagem "Tudo foi executado!"?

É muito importante você se lembrar que o código Java é interpretado no servidor, portanto apareceu no console do seu Tomcat.

Verifique o console do seu Tomcat.



6.3 LISTANDO OS CONTATOS COM SCRIPTLET

Uma vez que podemos escrever qualquer código Java como scriptlet, não fica difícil criar uma listagem de todos os contatos do banco de dados.

Temos todas as ferramentas necessárias para fazer essa listagem uma vez que já fizemos isso no capítulo de JDBC.

Basicamente, o código utilizará o `ContatoDao` que criamos anteriormente para imprimir a lista de `Contato`:

```
<%
ContatoDao dao = new ContatoDao();
List<Contato> contatos = dao.getLista();

for (Contato contato : contatos ) {
%>
    <li><%=contato.getNome()%>, <%=contato.getEmail()%>:
        <%=contato.getEndereco()%></li>

<%
}
%>
```

Nesse código ainda falta, assim como no Java puro, importar as classes dos pacotes corretos.

Para fazermos o import das classes, precisamos declarar que aquela página precisa de acesso à outras classes Java. Para isso, utilizamos diretivas, que possuem a seguinte sintaxe: `<%@ ... %>`.

Repare que ela se parece muito com scriptlet, com a diferença que possui uma `@` logo na abertura. Essa diretiva é uma diretiva de página, ou seja, uma configuração específica de uma página.

Para isso, utilizamos `<%@ page %>`. Basta dizermos qual configuração queremos fazer nessa página, que no nosso caso é importar uma classe para utilizá-la:

```
<%@ page import="br.com.caelum.agenda.dao.ContatoDao" %>
```

O atributo `import` permite que seja especificado qual o pacote a ser importado. Para importar diversos pacotes, podemos separá-los por vírgulas (vide o exercício).

6.4 EXERCÍCIOS OPCIONAIS: LISTA DE CONTATOS COM SCRIPTLET

1. Crie o arquivo `WebContent/lista-contatos-scriptlet.jsp` e siga:

- Importe os pacotes necessários. Use o Ctrl+Espaço do Eclipse para ajudar a escrever os pacotes.

```
<%@ page import="java.util.*,
br.com.caelum.agenda.dao.*,
br.com.caelum.agenda.modelo.*" %>
```

- Coloque o código para fazer a listagem. Use bastante o Ctrl+Espaço do Eclipse.

```

<html>
    <body>
        <table>
            <%
                ContatoDao dao = new ContatoDao();
                List<Contato> contatos = dao.getLista();

                for (Contato contato : contatos ) {
            %>
                    <tr>
                        <td><%=contato.getNome() %></td>
                        <td><%=contato.getEmail() %></td>
                        <td><%=contato.getEndereco() %></td>
                        <td><%=contato.getDataNascimento().getTime() %></td>
                    </tr>
            <%
                }
            %>
        </table>
    </body>
</html>

```

- Teste a url <http://localhost:8080/fj21-agenda/lista-contatos-scriptlet.jsp>
2. Repare que a data apareceu de uma forma complicada de ler. Tente mostrá-la formatada utilizando a classe `SimpleDateFormat`.
 3. Coloque cabeçalhos para as colunas da tabela, descrevendo o que cada coluna significa.
 4. Tente utilizar o quadro a seguir para definir a página padrão de seu site.

WELCOME-FILE-LIST

O arquivo web.xml abaixo diz que os arquivos chamados "bemvindo.jsp" devem ser chamados quando um cliente tenta acessar um diretório web qualquer.

O valor desse campo costuma ser "index.html" em outras linguagens de programação.

Como você pode ver pelo arquivo gerado automaticamente pelo WTP, é possível indicar mais de um arquivo para ser o seu welcome-file! Mude-o para:

```

<welcome-file-list>
    <welcome-file>bemvindo.jsp</welcome-file>
</welcome-file-list>

```

Reinic peace o tomcat e acesse a URL: <http://localhost:8080/fj21-agenda/>

6.5 MISTURANDO CÓDIGO JAVA COM HTML

Abrimos o capítulo dizendo que não queríamos misturar código Java com código HTML nas nossas `Servlets`, pois, prejudicava a legibilidade do nosso código e afetava a manutenção.

Mas é justamente isso que estamos fazendo agora, só que no JSP, através de Scriptlets.

É complicado ficar escrevendo Java em seu arquivo JSP, não é?

Primeiro, fica tudo mal escrito e difícil de ler. O Java passa a atrapalhar o código HTML em vez de ajudar. Depois, quando o responsável pelo design gráfico da página quiser alterar algo, terá que conhecer Java para entender o que está escrito lá dentro. Hmm... não parece uma boa solução.

E existe hoje em dia no mercado muitas aplicações feitas inteiramente utilizando scriptlets e escrevendo código Java no meio dos HTMLs.

Com o decorrer do curso, vamos evoluir nosso código até um ponto em que não faremos mais essa mistura.

6.6 EL: EXPRESSION LANGUAGE

Para remover um pouco do código Java que fica na página JSP, a Sun desenvolveu uma linguagem chamada **Expression Language** que é interpretada pelo servlet container.

Nosso primeiro exemplo com essa linguagem é utilizá-la para mostrar parâmetros que o cliente envia através de sua requisição.

Por exemplo, se o cliente chama a página `testaparam.jsp?idade=24`, o programa deve mostrar a mensagem que o cliente tem 24 anos.

Como fazer isso? Simples, existe uma variável chamada `param` que, na expression language, é responsável pelos parâmetros enviados pelo cliente. Para ler o parâmetro chamado **idade** basta usar `${param.idade}`. Para ler o parâmetro chamado dia devemos usar `${param.dia}`.

A expression language ainda vai ser utilizada para muitos outros casos, não apenas para pegar parâmetros que vieram do request. Ela é a forma mais elegante hoje em dia para trabalhar no JSP e será explorada novamente durante os capítulos posteriores.

6.7 EXERCÍCIOS: PARÂMETROS COM A EXPRESSION LANGUAGE

1. Crie uma página chamada **WebContent/digita-idade.jsp** com o conteúdo:

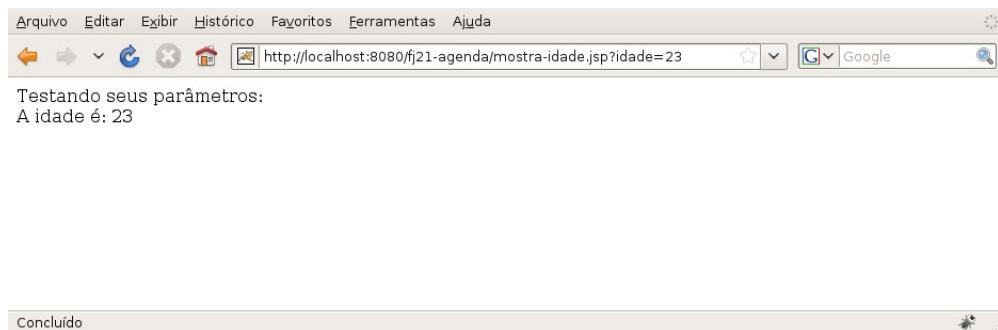
```
<html>
  <body>
    Digite sua idade e pressione o botão:<br />

    <form action="mostra-idade.jsp">
      Idade: <input type="text" name="idade"/> <input type="submit"/>
    </form>
  </body>
</html>
```

2. Crie um arquivo chamado **WebContent/mostra-idade.jsp** e coloque o código de expression language que mostra a idade que foi enviada como parâmetro para essa página:

```
<html>
  <body>
    Testando seus parâmetros:<br />
    A idade é ${param.idade}.
  </body>
</html>
```

3. Teste o sistema acessando a página <http://localhost:8080/fj21-agenda/digita-idade.jsp>.



6.8 PARA SABER MAIS: COMPILANDO OS ARQUIVOS JSP

Os arquivos JSPs não são compilados dentro do Eclipse, por esse motivo na hora que estamos escrevendo o JSP no Eclipse não precisamos das classes do driver.

Os JSPs são transformados em uma Servlet, que vimos anteriormente, por um compilador JSP (o Tomcat contém um compilador embutido). Esse compilador JSP pode gerar um código Java que é então compilado para gerar bytecode diretamente para a servlet.

Então, somente durante a execução de uma página JSP, quando ele é transformado em uma servlet, que seu código Java é compilado e necessitamos das classes do driver que são procuradas no diretório lib.

USANDO TAGLIBS

"Saber é compreendermos as coisas que mais nos convém." -- Friedrich Nietzsche

Nesse capítulo, você aprenderá o que são Taglibs e JSTL. E também terá a chance de utilizar algumas das principais tags do grupo `core` e `fmt`.

7.1 TAGLIBS

No capítulo anterior, começamos a melhorar nossos problemas com relação à mistura de código Java com HTML através da `Expression Language`. No entanto, ela sozinha não pode nos ajudar muito, pois ela não nos permite, por exemplo, instanciar objetos, fazer verificações condicionais (`if else`), iterações como em um `for` e assim por diante.

Para que possamos ter esse comportamento sem impactar na legibilidade do nosso código, teremos que escrever esse código nos nossos JSPs numa forma parecida com o que já escrevemos lá, que é HTML, logo, teremos que escrever código baseado em **Tags**.

Isso mesmo, uma **tag!** A Sun percebeu que os programadores estavam abusando do código Java no JSP e tentou criar algo mais "natural" (um ponto um tanto quanto questionável da maneira que foi apresentada no início), sugerindo o uso de tags para substituir trechos de código.

O resultado final é um conjunto de tags (uma **tag library**, ou **taglib**) padrão, que possui, entre outras tags, a funcionalidade de instanciar objetos através do construtor sem argumentos.

7.2 INSTANCIANDO POJOS

Como já foi comentado anteriormente, os Javabeans devem possuir o construtor público sem argumentos (um típico *Plain Old Java Object*: POJO), getters e setters.

Instanciá-los na nossa página JSP não é complicado. Basta utilizarmos a tag correspondente para essa função, que no nosso caso é a `<jsp:useBean>`.

Para utilizá-la, basta indicarmos qual a classe queremos instanciar e como se chamará a variável que será atribuída essa nova instância.

```
<jsp:useBean id="contato" class="br.com.caelum.agenda.modelo.Contato"/>
```

Podemos imprimir o nome do contato (que está em branco, claro...):

```
 ${contato.nome}
```

Mas, onde está o `getNome()`? A expression language é capaz de perceber sozinha a necessidade de chamar um método do tipo *getter*, por isso o padrão *getter/setter* do POJO é tão importante hoje em dia.

Desta maneira, classes como `Contato` são ferramentas poderosas por seguir esse padrão pois diversas bibliotecas importantes estão baseadas nele: Hibernate, Struts, VRaptor, JSF, EJB etc.

ATENÇÃO

Na Expression Language `${contato.nome}` chamará o método `getNome` por padrão. Para que isso sempre funcione, devemos colocar o parâmetro em letra minúscula. Ou seja, `${contato.Nome}` não funciona.

7.3 JSTL

Seguindo a ideia de melhorar o código Java que precisa de uma maneira ou outra ser escrito na página JSP, a Sun sugeriu o uso da **JavaServer Pages Standard Tag Library**, a JSTL.

OBSERVAÇÃO

Antes de 2005, JSTL significava *JavaServer Pages Standard Template Library*.

A **JSTL** é a API que encapsulou em tags simples toda a funcionalidade que diversas páginas Web precisam, como controle de laços (`fors`), controle de fluxo do tipo `if else`, manipulação de dados XML e a internacionalização de sua aplicação.

Antigamente, diversas bibliotecas foram criadas por vários grupos com funcionalidades similares ao JSTL (principalmente ao Core), culminando com a aparição da mesma, em uma tentativa da Sun de padronizar algo que o mercado vê como útil.

Existem ainda outras partes da JSTL, por exemplo aquela que acessa banco de dados e permite escrever códigos SQL na nossa página, mas se o designer não comprehende Java o que diremos de SQL? O uso de tal parte da JSTL é desencorajado.

A JSTL foi a forma encontrada de padronizar o trabalho de milhares de programadores de páginas JSP.

Antes disso, muita gente programava como nos exemplos que vimos anteriormente, somente com JSPs e Javabeans, o chamado Modelo 1, que na época fazia parte dos Blueprints de J2EE da Sun (boas práticas) e nós vamos discutir mais para frente no curso.

As empresas hoje em dia

Muitas páginas JSP no Brasil ainda possuem grandes pedaços de scriptlets espalhados dentro delas.

Recomendamos a todos os nossos alunos que optarem pelo JSP como camada de visualização, que utilizem a JSTL e outras bibliotecas de tag para evitar o código incompreensível que pode ser gerado com scriptlets.

O código das scriptlets mais confunde do que ajuda, tornando a manutenção da página JSP cada vez mais custosa para o programador e para a empresa.

7.4 INSTALAÇÃO

Para instalar a implementação mais famosa da **JSTL** basta baixar a mesma no site <http://jstl.java.net/>.

Ao usar o JSTL em alguma página precisamos primeiro definir o cabeçalho. Existem quatro APIs básicas e vamos aprender primeiro a utilizar a biblioteca chamada de **core**.

7.5 CABEÇALHO PARA A JSTL CORE

Sempre que vamos utilizar uma taglib devemos primeiro escrever um cabeçalho através de uma tag JSP que define qual taglib vamos utilizar e um nome, chamado *prefixo*.

Esse prefixo pode ter qualquer valor mas no caso da taglib core da JSTL o padrão da Sun é a letra **c**. Já a URI (que não deve ser decorada) é mostrada a seguir e não implica em uma requisição pelo protocolo http e sim uma busca entre os arquivos .jar no diretório lib.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

7.6 FOREACH

Usando a JSTL core, vamos reescrever o arquivo que lista todos contatos.

O cabeçalho já é conhecido da seção anterior:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Depois, precisamos instanciar e declarar nosso DAO. Ao revisar o exemplo da lista através de scriptlets, queremos executar o seguinte:

- classe: br.com.caelum.jdbc.dao.ContatoDao ;
- construtor: sem argumentos;
- variável: DAO.

Já vimos a tag **jsp:useBean**, capaz de instanciar determinada classe através do construtor sem argumentos e dar um nome (id) para essa variável.

Portanto vamos utilizar a tag `useBean` para instanciar nosso `ContatoDao` :

```
<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDao"/>
```

Como temos a variável `dao` , desejamos chamar o método `getLista` e podemos fazer isso através da EL:

```
${dao.lista}
```

Desejamos executar um loop para cada `contato` dentro da coleção retornada por esse método:

- array ou coleção: `dao.lista`;
- variável temporária: `contato`.

No nosso exemplo com scriptlets, o que falta é a chamada do método `getLista` e a iteração:

```
<%
// ...
List<Contato> contatos = dao.getLista();

for (Contato contato : contatos ) {
%>
    <%=contato.getNome()%>, <%=contato.getEmail()%>,
    <%=contato.getEndereco()%>, <%=contato.getDataNascimento() %>
<%
}
%>
```

A JSTL core disponibiliza uma tag chamada `c:forEach` capaz de iterar por uma coleção, exatamente o que precisamos. No `c:forEach` , precisamos indicar a coleção na qual vamos iterar, através do atributo `items` e também como chamará o objeto que será atribuído para cada iteração no atributo `var` . O exemplo a seguir mostra o uso de *expression language* de uma maneira muito mais elegante:

```
<c:forEach var="contato" items="${dao.lista}">
    ${contato.nome}, ${contato.email},
    ${contato.endereco}, ${contato.dataNascimento}
</c:forEach>
```

Mais elegante que o código que foi apresentado usando scriptlets, não?

FOREACH E VARSTATUS

É possível criar um contador do tipo `int` dentro do seu laço `foreach`. Para isso, basta definir o atributo chamado `varStatus` para a variável desejada e utilizar a propriedade `count` dessa variável.

```
<table border="1">
    <c:foreach var="contato" items="${dao.lista}" varStatus="id">
        <tr bgcolor="#${id.count % 2 == 0 ? 'aaee88' : 'ffffff'}">
            <td>${id.count}</td><td>${contato.nome}</td>
        </tr>
    </c:foreach>
</table>
```

7.7 EXERCÍCIOS: FOREACH

1. Precisamos primeiro colocar os JARs da JSTL em nossa aplicação.
 - Primeiro, vá ao Desktop, e entre no diretório `21/jars-jstl`
 - Haverá dois jars, `javax.servlet.jsp.jstl-x.x.x.jar` e `javax.servlet.jsp.jstl-api-x.x.x.jar`
 - Copie-os (CTRL+C) e cole-os (CTRL+V) dentro de **workspace/fj21-agenda/WebContent/WEB-INF/lib**
 - No Eclipse, dê um F5 no seu projeto ou clique com o botão direito do mouse sobre o nome do projeto e escolha a opção *Refresh*.

EM CASA

Caso você esteja em casa, pode fazer o download da JSTL API e da implementação em:
<https://jstl.java.net/download.html>

2. Liste os contatos de `ContatoDao` usando `jsp:useBean` e JSTL.

- Crie o arquivo `lista-contatos.jsp`, dentro da pasta **WebContent/** usando o atalho de novo JSP no Eclipse;
- Antes de escrevermos nosso código, precisamos importar a taglib JSTL Core. Isso é feito com a diretiva `<%@ taglib %>` no topo do arquivo. Usando o recurso de autocompletar do Eclipse (inclusive na URL), declare a taglib no topo do arquivo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

- Localize a tag `<body>` no arquivo e implemente o conteúdo da nossa página **dentro do body**. Vamos usar uma tabela HTML e as tags `<jsp:useBean/>` e `<c:forEach/>` que vimos antes:

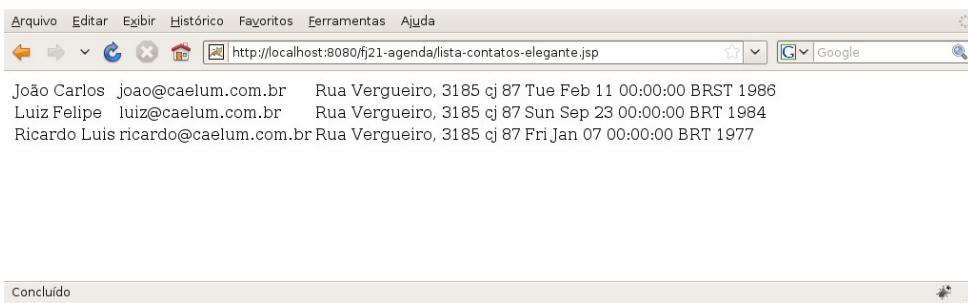
```

<!-- cria o DAO -->
<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDao"/>

<table>
    <!-- percorre contatos montando as linhas da tabela -->
    <c:forEach var="contato" items="${dao.lista}">
        <tr>
            <td>${contato.nome}</td>
            <td>${contato.email}</td>
            <td>${contato.endereco}</td>
            <td>${contato.dataNascimento.time}</td>
        </tr>
    </c:forEach>
</table>

```

- Acesse <http://localhost:8080/fj21-agenda/lista-contatos.jsp>



Repare que após criar uma nova página JSP não precisamos reiniciar o nosso container!

3. Scriptlets ou JSTL? Qual dos dois é mais fácil entender?

7.8 EXERCÍCIOS OPCIONAIS

1. Coloque um cabeçalho nas colunas da tabela com um título dizendo à que se refere a coluna.
2. Utilize uma variável de status no seu `c:forEach` para colocar duas cores diferentes em linhas pares e ímpares. (Utilize como auxílio o `box` imediatamente antes do exercício sobre `forEach`)

7.9 EVOLUINDO NOSSA LISTAGEM

A listagem dos nossos contatos funciona perfeitamente, mas o nosso cliente ainda não está satisfeito. Ele quer um pouco mais de facilidade nessa tela, e sugere que caso o usuário tenha e-mail cadastrado, coloquemos um link no e-mail que quando clicado abra o software de e-mail do computador do usuário para enviar um novo e-mail para esse usuário. Como podemos fazer essa funcionalidade?

Vamos analisar o problema com calma. Primeiro, percebemos que vamos precisar criar um link para envio de e-mail. Isso é facilmente conseguido através da tag do HTML `<a>` com o parâmetro

`href="mailto:email@email.com"` . Primeiro problema resolvido facilmente, mas agora temos outro. Como faremos a verificação se o e-mail está ou não preenchido?

7.10 FAZENDO IFS COM A JSTL

Para que possamos fazer essa verificação precisaremos fazer um `if` para sabermos se o e-mail está preenchido ou não. Mas, novamente, não queremos colocar código Java na nossa página e já aprendemos que estamos mudando isso para utilizar tags. Para essa finalidade, existe a tag `c:if` , na qual podemos indicar qual o teste lógico deve ser feito através do atributo `test` . Esse teste é informado através de *Expression Language*.

Para verificarmos se o e-mail está preenchido ou não, podemos fazer o seguinte:

```
<c:if test="${not empty contato.email}">
    <a href="mailto:${contato.email}">${contato.email}</a>
</c:if>
```

Podemos também, caso o e-mail não tenha sido preenchido, colocar a mensagem "e-mail não informado", ao invés de nossa tabela ficar com um espaço em branco. Repare que esse é justamente o caso contrário que fizemos no nosso `if` , logo, é equivalente ao `else` .

O problema é que não temos a tag `else` na JSTL, por questões estruturais de XML. Uma primeira alternativa seria fazermos outro `<c:if>` com a lógica invertida. Mas isso não é uma solução muito elegante. No Java, temos outra estrutura condicional que consegue simular um `if/else` , que é o `switch/case` .

Para simularmos `switch/case` com JSTL, utilizamos a tag `c:choose` e para cada caso do switch fazemos `c:when` . O `default` do switch pode ser representado através da tag `c:otherwise` , como no exemplo a seguir:

```
<c:choose>
    <c:when test="${not empty contato.email}">
        <a href="mailto:${contato.email}">${contato.email}</a>
    </c:when>
    <c:otherwise>
        E-mail não informado
    </c:otherwise>
</c:choose>
```

7.11 EXERCÍCIOS: LISTA DE CONTATOS COM CONDICIONAIS

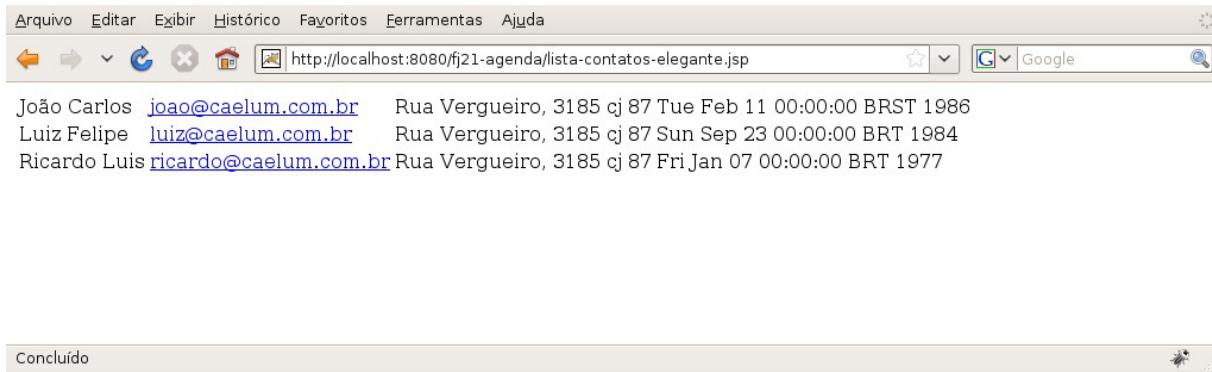
1. Vamos colocar em nossa listagem um link para envio de e-mail caso o mesmo tenha sido informado.
 - Abra o arquivo `lista-contatos.jsp` no Eclipse;
 - No momento de imprimir o e-mail do contato, adicione uma verificação para saber se o e-mail está preenchido e, caso esteja, adicione um link para envio de e-mail:

```

<c:forEach var="contato" items="#${dao.lista}">
    <tr>
        <td>${contato.nome}</td>
        <td>
            <c:if test="#{not empty contato.email}">
                <a href="mailto:${contato.email}">${contato.email}</a>
            </c:if>
        </td>
        <td>${contato.endereco}</td>
        <td>${contato.dataNascimento.time}</td>
    </tr>
</c:forEach>

```

- Acesse a página no navegador pelo endereço <http://localhost:8080/fj21-agenda/lista-contatos.jsp>



2. Vamos colocar a mensagem "E-mail não informado" caso o e-mail não tenha sido informado

- Abaixo do novo `if` que fizemos no item anterior, vamos colocar mais um `if`, dessa vez com a verificação contrária, ou seja, queremos saber se está vazio:

```

<c:forEach var="contato" items="#${dao.lista}">
    <tr>
        <td>${contato.nome}</td>
        <td>
            <c:if test="#{not empty contato.email}">
                <a href="mailto:${contato.email}">${contato.email}</a>
            </c:if>

            <c:if test="#{empty contato.email}">
                E-mail não informado
            </c:if>
        </td>
        <td>${contato.endereco}</td>
        <td>${contato.dataNascimento.time}</td>
    </tr>
</c:forEach>

```

- Caso você não possua nenhum contato sem e-mail, cadastre algum.
- Acesse a lista-contatos.jsp pelo navegador e veja o resultado final.

João Carlos	joao@caelum.com.br	Rua Vergueiro, 3185 cj 87 Tue Feb 11 00:00:00 BRST 1986
Luiz Felipe	lui@caelum.com.br	Rua Vergueiro, 3185 cj 87 Sun Sep 23 00:00:00 BRT 1984
Ricardo Luis	ricardo@caelum.com.br	Rua Vergueiro, 3185 cj 87 Fri Jan 07 00:00:00 BRT 1977
Adriano Henrique	E-mail não informado	Rua Vergueiro, 3185 cj 87 Sun Mar 27 00:00:00 BRT 1988

Concluído

3. (Opcional) Ao invés de utilizar dois `ifs`, use a tag `c:choose`

7.12 IMPORTANDO PÁGINAS

Um requisito comum que temos nas aplicações Web hoje em dia é colocar cabeçalhos e rodapé nas páginas do nosso sistema. Esses cabeçalhos e rodapés podem ter informações da empresa, do sistema e assim por diante. O problema é que, na grande maioria das vezes, **todas** as páginas da nossa aplicação precisam ter esse mesmo cabeçalho e rodapé. Como poderíamos resolver isso?

Uma primeira alternativa e talvez a mais inocente é colocarmos essas informações em todas as páginas da nossa aplicação, copiando e colando todo o cabeçalho várias vezes.

Mas o que aconteceria se precisássemos mudar o logotipo da empresa? Teríamos que mudar todas as páginas, o que não é um trabalho agradável.

Uma alternativa melhor seria isolarmos esse código que se repete em todas as páginas em uma outra página, por exemplo, `cabecalho.jsp` e todas as páginas da nossa aplicação, apenas dizem que precisam dessa outra página nela, através de uma tag nova, a `c:import`.

Para utilizá-la, podemos criar uma pagina com o cabeçalho do sistema, por exemplo, a `cabecalho.jsp`:

```
 Nome da empresa
```

E uma página para o rodapé, por exemplo, `rodape.jsp`:

```
Copyright 2010 - Todos os direitos reservados
```

Bastaria que, em todas as nossas páginas, por exemplo, na `lista-contatos.jsp`, colocássemos ambas as páginas, através da `c:import` como abaixo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<body>

<c:import url="cabecalho.jsp" />
```

```

<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDao"/>
<table>
    <!-- for -->
    <c:forEach var="contato" items="${dao.lista}">
        <tr>
            <td>${contato.nome}</td>
            <td>${contato.email}</td>
            <td>${contato.endereco}</td>
            <td>${contato.dataNascimento.time}</td>
        </tr>
    </c:forEach>
</table>

<c:import url="rodape.jsp" />

</body>
</html>

```

7.13 EXERCÍCIOS: CABEÇALHOS E RODAPÉS

- Vamos primeiro criar o nosso cabeçalho, utilizando o logotipo da Caelum.

- Vá no Desktop, entre na pasta 21 e copie o diretório `imagens` para dentro do diretório `WebContent` do seu projeto. Esse diretório possui o logotipo da Caelum.

Ou você pode usar o que se encontra em: <http://www.caelum.com.br/imagens/base/caelum-ensino-inovacao.png>

- Crie dentro de `WebContent` um arquivo chamado `cabecalho.jsp` com o logotipo do sistema:

```


<h2>Agenda de Contatos do(a) (Seu nome aqui)</h2>
<hr />

```

- Crie também a página `rodape.jsp`:

```

<hr />
Copyright 2010 - Todos os direitos reservados

```

- Podemos importar as duas páginas (`cabecalho.jsp` e `rodape.jsp`), dentro da nossa `listatodos.jsp` usando a tag `c:import` que vimos:

```

<html>
<body>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<c:import url="cabecalho.jsp" />

<!-- cria a lista -->
<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDao"/>
<table>
    <!-- for -->
    <c:forEach var="contato" items="${dao.lista}">
        <tr>
            <td>${contato.nome}</td>

```

```

<td>
<c:if test="${not empty contato.email}">
    <a href="mailto:${contato.email}">${contato.email}</a>
</c:if>
<c:if test="${empty contato.email}">
    E-mail não informado
</c:if>
</td>

<td>${contato.endereco}</td>
<td>${contato.dataNascimento.time}</td>
</tr>
</c:forEach>
</table>

<c:import url="rodape.jsp" />
</body>
</html>

```

- Visualize o resultado final acessando no navegador <http://localhost:8080/fj21-agenda/lista-contatos.jsp>

Arquivo Editar Exibir Histórico Favoritos Ferramentas Ajuda

http://localhost:8080/fj21-agenda/lista-contatos-elegante.jsp

Caelum
Ensino e Inovação

Agenda de Contatos da Caelum

João Carlos	joao@caelum.com.br	Rua Vergueiro, 3185 cj 87 Tue Feb 11 00:00:00 BRST 1986
Luiz Felipe	luiz@caelum.com.br	Rua Vergueiro, 3185 cj 87 Sun Sep 23 00:00:00 BRT 1984
Ricardo Luis	ricardo@caelum.com.br	Rua Vergueiro, 3185 cj 87 Fri Jan 07 00:00:00 BRT 1977
Adriano Henrique	E-mail não informado	Rua Vergueiro, 3185 cj 87 Sun Mar 27 00:00:00 BRT 1988

Copyright 2010 - Todos os direitos reservados

Concluído

7.14 FORMATAÇÃO DE DATAS

Apesar da nossa listagem de contatos estar bonita e funcional, ela ainda possui problemas, por exemplo, a visualização da data de nascimento. Muitas informações aparecem na data, como horas, minutos e segundos, coisas que não precisamos.

Já aprendemos anteriormente que uma das formas que podemos fazer essa formatação em código Java é através da classe `SimpleDateFormat`, mas não queremos utilizá-la aqui, pois não queremos código Java no nosso JSP.

Para isso, vamos utilizar outra Taglib da JSTL que é a taglib de formatação, a `fmt`.

Para utilizarmos a taglib `fmt`, precisamos importá-la, da mesma forma que fizemos com a tag `core`, através da diretiva de `taglib`, como abaixo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
```

Dentro da taglib `fmt`, uma das tags que ela possui é a `formatDate`, que faz com que um determinado objeto do tipo `java.util.Date` seja formatado para um dado `pattern`. Então, podemos utilizar a tag `fmt:formatDate` da seguinte forma:

```
<fmt:formatDate value="${contato.dataNascimento.time}"  
pattern="dd/MM/yyyy" />
```

Repare que, na *Expression Language* que colocamos no `value`, há no final um `.time`. Isso porque o atributo `value` só aceita objetos do tipo `java.util.Date`, e nossa data de nascimento é um `java.util.Calendar` que possui um método `getTime()` para chegarmos ao seu respectivo `java.util.Date`.

COMO FAZER PATTERNS MAIS COMPLICADOS?

Podemos no atributo `pattern` colocar outras informações com relação ao objeto `java.util.Date` que queremos mostrar, por exemplo:

- `m` - Minutos
- `s` - Segundos
- `H` - Horas (0 - 23)
- `D` - Dia no ano, por exemplo, 230

Sugerimos que quando precisar de formatações mais complexas, leia a documentação da classe `SimpleDateFormat`, aonde se encontra a descrição de alguns caracteres de formatação.

7.15 EXERCÍCIOS: FORMATANDO A DATA DE NASCIMENTO DOS CONTATOS

1. Vamos fazer a formatação da data de nascimento dos nossos contatos.

- Na `lista-contatos.jsp`, importe a taglib `fmt` no topo do arquivo. Use o autocompletar do Eclipse para te ajudar a escrever:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
```

- Troque a Expression Language que mostra a data de nascimento para passar pela tag `formatDate` abaixo:

```
<fmt:formatDate value="${contato.dataNascimento.time}"  
pattern="dd/MM/yyyy" />
```

- Acesse a página pelo navegador e veja o resultado final:

Arquivo Editar Exibir Histórico Favoritos Ferramentas Ajuda

http://localhost:8080/fj21-agenda/lista-contatos-elegante.jsp

Caelum
Ensino e Inovação

Agenda de Contatos da Caelum

João Carlos	joao@caelum.com.br	Rua Vergueiro, 3185 cj 87 11/02/1986
Luiz Felipe	luiz@caelum.com.br	Rua Vergueiro, 3185 cj 87 23/09/1984
Ricardo Luis	ricardo@caelum.com.br	Rua Vergueiro, 3185 cj 87 07/01/1977
Adriano Henrique	E-mail não informado	Rua Vergueiro, 3185 cj 87 27/03/1988

Copyright 2010 - Todos os direitos reservados

Concluído

7.16 PARA SABER MAIS: LINKS COM

Muitas vezes trabalhar com links em nossa aplicação é complicado. Por exemplo, no arquivo `cabecalho.jsp` incluímos uma imagem chamada `caelum.png` que está na pasta `imagens`.

Incluímos esta imagem com a tag HTML `` utilizando o caminho `imagens/caelum.png` que é um caminho relativo, ou seja, se o `cabecalho.jsp` estiver na raiz do projeto, o arquivo `caelum.png` deverá estar em uma pasta chamada `imagens` também na raiz do projeto.

Utilizar caminhos relativos muitas vezes é perigoso. Por exemplo, se colocarmos o arquivo `cabecalho.jsp` em um diretório chamado `arquivos_comuns`, a imagem que tínhamos adicionado será procurada em um diretório `imagens` dentro do diretório `arquivos_comuns`. Resultado, a imagem não será exibida.

Poderíamos resolver isso utilizando um caminho absoluto. Ao invés de adicionarmos a imagem utilizando o caminho `imagens/caelum.png`, poderíamos usar `/imagens/caelum.png`. Só que utilizando a `/` ele não vai para a raiz da minha aplicação e sim para a raiz do tomcat, ou seja, ele iria procurar a imagem `caelum.png` em um diretório `imagens` na raiz do tomcat, quando na verdade o diretório nem mesmo existe.

Poderíamos resolver isso utilizando o caminho `/fj21-tarefas/imagens/caelum.png`. Nossa problema seria resolvido, entretanto, se algum dia mudássemos o contexto da aplicação para `tarefas` a imagem não seria encontrada, pois deixamos fixo no nosso jsp qual era o contexto da aplicação.

Para resolver este problema, podemos utilizar a tag `<c:url>` da JSTL. O uso dela é extremamente simples. Para adicionarmos o logotipo da caelum no `cabecalho.jsp`, faríamos da seguinte maneira:

```
<c:url value="/imagens/caelum.png" var="imagem"/>

```

Ou de uma forma ainda mais simples:

```
<img src="" />
```

O HTML gerado pelo exemplo seria: .

7.17 EXERCÍCIOS OPCIONAIS: CAMINHO ABSOLUTO

1.

- Abra o arquivo `cabecalho.jsp` e **altere-o** adicionando a tag `<c:url>` :

```
<img src="" />
```

Como vamos usar a JSTL também nesse novo arquivo de cabeçalho, não deixe de incluir a declaração da taglib no início do arquivo. (o comando `<%@ taglib ... %>` semelhante ao usado na listagem)

- Visualize o resultado acessando no navegador `http://localhost:8080/fj21-agenda/lista-contatos.jsp`
- A imagem `caelum.png` continua aparecendo normalmente

Se, algum dia, alterarmos o contexto da nossa aplicação, o cabeçalho continuaria exibindo a imagem da maneira correta. Usando a tag `<c:url>` ficamos livres para utilizar caminhos absolutos.

7.18 PARA SABER MAIS: OUTRAS TAGS

A JSTL possui além das tags que vimos aqui, muitas outras, e para diversas finalidades. Abaixo está um resumo com algumas das outras tags da JSTL:

- **c:catch** - bloco do tipo try/catch
- **c:forTokens** - for em tokens (ex: "a,b,c" separados por vírgula)
- **c:out** - saída
- **c:param** - parâmetro
- **c:redirect** - redirecionamento
- **c:remove** - remoção de variável
- **c:set** - criação de variável

Leia detalhes sobre as taglibs da JSTL (JavaDoc das tags e classes) em:
<http://java.sun.com/products/jsp/jstl/reference/api/index.html>

TAGS CUSTOMIZADAS COM TAGFILES

"Eu apenas invento e espero que outros apareçam precisando do que inventei" -- R. Buckminster Fuller

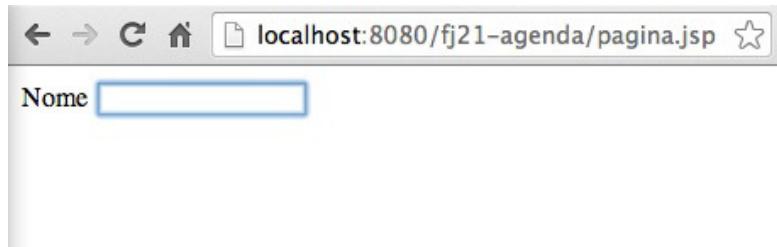
Ao término desse capítulo, você será capaz de:

- criar sua própria taglib através de tagfiles;
- encapsular códigos repetitivos em tags;
- conhecer outras taglibs existentes no mercado.

8.1 PORQUE EU PRECISARIA DE OUTRAS TAGS ALÉM DA JSTL?

É muito comum, no momento em que estamos desenvolvendo nossos JSPs, cairmos em situações em que fazemos muita repetição de código. Por exemplo, sempre que criamos uma caixa de texto, podemos associá-la com um `label`, através do seguinte código:

```
<label for="nomeContato">Nome</label>
<input type="text" id="nomeContato" name="nome" />
```



Esse código faz com que, se você clicar na palavra Nome, passe o foco para o campo de texto. Mas repare que temos algo que se repete nesse trecho, que é o `id` do `input` e o atributo `for` do `label`. Se mudarmos o `id` do `input` teremos que refletir essa alteração no `for` do `label`. Além disso, temos que sempre escrever todo esse código.

Não seria mais simples escrevermos `<campoTexto id="nomeContato" name="nome" label="Nome:>` e todo aquele código ser gerado para nós?

Um outro exemplo poderia ser quando utilizamos componentes que dependem de *Javascript*, como um campo que, ao ganhar o foco, mostra um calendário. Toda vez que temos que usar esse componente, precisamos escrever um pedaço de código JavaScript. Por que não encapsular tudo isso em novas tags

customizadas?

8.2 CALENDÁRIOS COM JQUERY

Vamos pegar um exemplo prático de possível uso de JavaScript no nosso sistema. Temos no cadastro de contatos, um campo para data de nascimento. Mas até o momento, o usuário precisa digitar a data na mão seguindo o formato que especificamos.

E se mostrássemos um calendário em JavaScript para o usuário escolher a data apenas clicando em um componente já pronto?

Para conseguirmos esse comportamento, podemos utilizar por exemplo os componentes visuais do **jQuery**, que é uma biblioteca com alguns componentes *JavaScript*.

Entre os diversos componentes que o *jQuery* possui, um deles é o campo com calendário, também conhecido por **datepicker**. Para utilizarmos o *datepicker* do *jQuery*, precisamos criar um `input` de texto simples e associá-lo com uma função *JavaScript*, como no código seguinte:

```
<input id="dataNascimento" type="text">

<script>
  $("#dataNascimento").datepicker();
</script>
```

A função *JavaScript* faz com que o `input` cujo `id` é `dataNascimento` seja um calendário.

Imagine se toda hora que fossemos criar um campo de data tivéssemos que escrever esse código *JavaScript*? As chances de errarmos esse código é razoável, mas ainda assim, o pior ponto ainda seria perder tempo escrevendo um código repetitivo, sendo que poderíamos obter o mesmo resultado apenas escrevendo:

```
<campoData id="dataNascimento" />
```

Qual abordagem é mais simples?

8.3 CRIANDO MINHAS PRÓPRIAS TAGS COM TAGFILES

Para que possamos ter a tag `<campoData>` precisaremos criá-la. Uma das formas que temos de criar nossas próprias taglibs é criando um arquivo contendo o código que nossa Taglib gerará. Esses arquivos contendo o código das tags são conhecidos como **tagfiles**.

Tagfiles nada mais são do que pedaços de JSP, com a extensão `.tag`, contendo o código que queremos que a nossa tag gere.

Vamos criar um arquivo chamado **campoData.tag** com o código que queremos gerar.

```
<input id="dataNascimento" name="dataNascimento">
```

```
<script>
    $("#" + id).datepicker();
</script>
```

Mas essa nossa tag está totalmente inflexível, pois o `id` e o nome do campo estão escritos diretamente dentro da tag. O que aconteceria se tivéssemos dois calendários dentro do mesmo formulário? Ambos teriam o mesmo `name`.

Precisamos fazer com que a nossa tag receba *parâmetros*. Podemos fazer isso adicionando a diretiva `<%@ attribute %>` na nossa tag, com os parâmetros `name` representando o nome do atributo e o parâmetro `required` com os valores `true` ou `false`, indicando se o parâmetro é obrigatório ou não.

```
<%@ attribute name="id" required="true" %>
```

Repare como é simples. Basta declarar essa diretiva de atributo no começo do nosso tagfile e temos a capacidade de receber um valor quando formos usar a tag. Imagine usar essa tag nova no nosso formulário:

```
<campoData id="dataNascimento" />
```

Já que nossa tag sabe receber parâmetros, basta usarmos esse parâmetro nos lugares adequados através de *expression language*, como no código abaixo:

```
<%@ attribute name="id" required="true" %>

<input id="${id}" name="${id}" type="text">
<script>
    $("#" + ${id}).datepicker();
</script>
```

Para usar nossa tag, assim como nas outras taglibs, precisamos importar nossos tagfiles. Como não estamos falando de taglibs complexas como a JSTL, mas sim de pequenos arquivos de tags do nosso próprio projeto, vamos referenciar diretamente a pasta onde nossos arquivos `.tag` estão salvos.

Nossos tagfiles devem ficar na pasta **WEB-INF/tags/** dentro do projeto e, no momento de usar as tags, vamos importar com a mesma diretiva `<%@ taglib %>` que usamos antes. A diferença é que, além de receber o **prefixo** da nova taglib, indicamos a pasta **WEB-INF/tags/** como localização das tags:

```
<%@taglib tagdir="/WEB-INF/tags" prefix="caelum" %>
```

Aqui decidimos importar a nova taglib sob o prefixo **caelum**, mas poderia ser qualquer nome. Assim, podemos usar a tag como:

```
<caelum:campoData id="dataNascimento" />
```

Repare que o nome da nossa nova Tag, é o mesmo nome do arquivo que criamos, ou seja, `campoData.tag` será utilizando como `<caelum:campoData>`.

UTILIZANDO BIBLIOTECAS JAVASCRIPT

Para podermos utilizar bibliotecas *Javascript* em nossa aplicação precisamos importar o arquivo `.js` que contém a biblioteca.

Para fazermos essa importação, basta que no cabeçalho da página que queremos utilizar o *Javascript*, ou seja, na Tag `head` , declaremos o seguinte:

```
<head>
    <script src="js/arquivo.js"></script>
</head>
```

Para saber mais sobre JavaScript, temos os cursos de Web da Caelum:

<http://www.caelum.com.br/cursos-web-front-end/>

UTILIZANDO ESTILOS EM CSS

Para que possamos construir uma interface agradável para o usuário, na maioria das vezes somente HTML não é suficiente.

Podemos também utilizar CSS (*Cascading Stylesheet*), que nada mais é que um arquivo contendo as definições visuais para sua página. Esses arquivos são distribuídos com a extensão `.css` e para que possamos usá-los, precisamos também importá-los dentro da tag `head` da nossa página que vai utilizar esses estilos. Como abaixo:

```
<head>
    <link href="css/meuArquivo.css" rel="stylesheet">
</head>
```

A Caelum oferece os treinamentos WD-43 e WD-47 para quem quer dominar as melhores técnicas de desenvolvimento Web com semântica perfeita, estilos CSS poderosos e JavaScripts corretos e funcionais.

<http://www.caelum.com.br/cursos-web-front-end/>

8.4 EXERCÍCIOS: CRIANDO NOSSA PRÓPRIA TAG PARA CALENDÁRIO

1. Vamos criar nossa tag para o campo de calendário com **datepicker**. Para isso vamos utilizar a biblioteca javascript jQuery.
 - Vá ao Desktop, e entre na pasta `21` ;

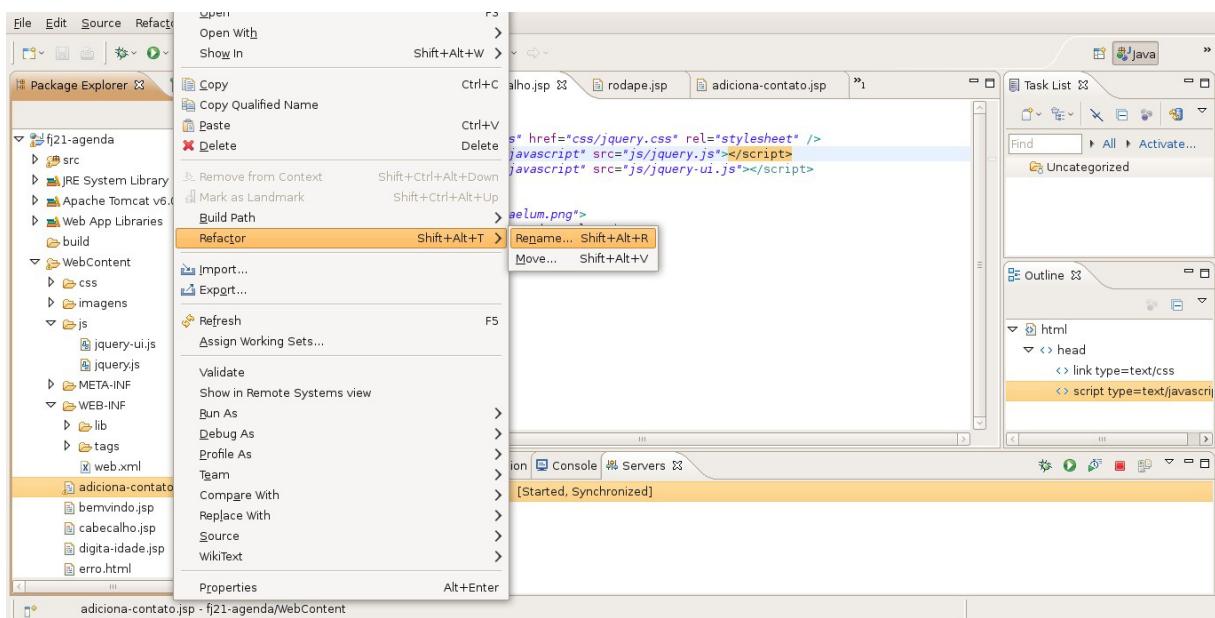
- Copie os diretórios `js` e `css` e cole-os dentro de `WebContent` no seu projeto;

EM CASA

Se você estiver fazendo de casa, pode achar esses dois arquivos no pacote do JQueryUI em: <http://jqueryui.com/download>

2. Queremos usar JSTL na página de adicionar contatos, a `adiciona-contato.html`. Mas temos um problema. Não podemos utilizar as tags ou expression language em uma página `HTML`. Para resolvemos isso, vamos trocar sua extensão para `.jsp`

- Clique com o botão direito em cima do arquivo `adiciona-contato.html` e escolha a opção *Rename* e troque a extensão de `html` para `jsp`, dessa forma o arquivo se chamará `adiciona-contato.jsp`.



3. Também vamos importar o cabeçalho e o rodapé nessa página, portanto, vamos usar o taglib JSTL `core`.

- Adicione no topo da página:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

- Vamos importar o cabeçalho, depois da tag `<body>` adicione:

```
<c:import url="cabecalho.jsp" />
```

- Falta o rodapé. Antes de fechar o tag body (</body>) adicione:

```
<c:import url="rodape.jsp" />
```

Cuidado para não apagar o formulário, precisaremos dele mais para frente.

- Acesse <http://localhost:8080/fj21-agenda/adiciona-contato.jsp> e verifique o resultado com o cabeçalho e rodapé.

- Precisamos importar o jQuery na página `adiciona-contato.jsp`. Para isso adicione depois da tag `<html>` e antes da tag `<body>`, os estilos CSS e os *Javascripts*

```
<html>
    <head>
        <link href="css/jquery.css" rel="stylesheet">
        <script src="js/jquery.js"></script>
        <script src="js/jquery-ui.js"></script>
    </head>

    <body>

        <!-- Restante da página aqui -->
```

- Vamos criar a nossa tag para o calendário.

- Dentro de `WEB-INF` crie um diretório chamado **tags**.
- Crie um arquivo chamado **campoData.tag** dentro de `WEB-INF/tags/`:

```
<%@ attribute name="id" required="true" %>

<input type="text" id="${id}" name="${id}" />
<script>
    $("#" + ${id}).datepicker({dateFormat: 'dd/mm/yy'});
</script>
```

- Para utilizá-la, vamos voltar para o `adiciona-contato.jsp` e junto à declaração da Taglib `core`, vamos importar nossa nova tag:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@taglib tagdir="/WEB-INF/tags" prefix="caelum" %>
```

- Basta trocarmos o `input` da data de nascimento pela nossa nova tag:

```
<form action="adicionaContato">
    Nome: <input type="text" name="nome" /><br />
    E-mail: <input type="text" name="email" /><br />
    Endereço: <input type="text" name="endereco" /><br />
    Data Nascimento: <caelum:campoData id="dataNascimento" /><br />

    <input type="submit" value="Gravar"/>
</form>
```

- Recarregue a página `adiciona-contato.jsp` e clique no campo da data de nascimento. O calendário deve ser aberto, escolha uma data e faça a gravação do contato.



- Verifique o código fonte gerado, e repare que o *Javascript* do calendário e o input foram gerados para nós através de nossa Taglib.

6. (opcional) Consulte a documentação do componente **datepicker** do *jQuery UI* para ver mais opções. Você pode acessá-la em: <http://jqueryui.com/demos/datepicker/>

Consulte, por exemplo, as opções `changeYear` e `changeMonth` para mostrar menus mais fáceis para escolher o ano e o mês. Há muitas outras opções também.

8.5 PARA SABER MAIS: OUTRAS TAGLIBS NO MERCADO

Muitos dos problemas do dia a dia que nós passamos, alguém já passou antes, portanto, muitas das vezes, pode acontecer de já existir uma Taglib que resolva o mesmo problema que você pode estar passando. Portanto, recomendamos que antes de criar a sua Taglib, sempre verifique se já não existe alguma que resolva seu problema.

O mercado atualmente possui várias Taglibs disponíveis, e para diversas funções, algumas das mais conhecidas são:

- Displaytag:** É uma Taglib para geração fácil de tabelas, e pode ser encontrada em <http://displaytag.sf.net>
- Cewolf:** É uma Taglib para geração de gráficos nas suas páginas e também pode ser encontrada em: <http://cewolf.sourceforge.net/new/>
- Waffle Taglib:** Tags para auxiliar na criação de formulários HTML que é encontrada em:

<http://waffle.codehaus.org/taglib.html>

8.6 DESAFIO: COLOCANDO DISPLAYTAG NO PROJETO

1. Adicione a displaytag no seu projeto para fazer a listagem dos contatos.

MVC - MODEL VIEW CONTROLLER

"Ensinar é aprender duas vezes." -- Joseph Joubert

Nesse capítulo, você aprenderá:

- O padrão arquitetural MVC;
- A construir um framework MVC simples.

9.1 SERVLET OU JSP?

Colocar todo HTML dentro de uma Servlet realmente não nos parece a melhor ideia. O que acontece quando precisamos mudar o design da página? O designer não vai saber Java para editar a Servlet, recompilá-la e colocá-la no servidor.

Imagine usar apenas JSP. Ficaríamos com muito *scriptlet*, que é muito difícil de dar manutenção.

Uma ideia mais interessante é usar o que é bom de cada um dos dois.

O JSP foi feito apenas para apresentar o resultado, e ele não deveria fazer acessos a banco de dados e nem fazer a instanciação de objetos. Isso deveria estar em código Java, na Servlet.

O ideal então é que a Servlet faça o trabalho árduo, a tal da **lógica de negócio**. E o JSP apenas apresente visualmente os resultados gerados pela Servlet. A Servlet ficaria então com a lógica de negócios (ou regras de negócio) e o JSP tem a **lógica de apresentação**.

Imagine o código do método da servlet `AdicionaContatoServlet` que fizemos antes:

```
protected void service(HttpServletRequest request,
    HttpServletResponse response) {

    // log
    System.out.println("Tentando criar um novo contato...");

    // acessa o bean
    Contato contato = new Contato();
    // chama os setters
    ...

    // adiciona ao banco de dados
    ContatoDao dao = new ContatoDao();
    dao.adiciona(contato);
```

```

// ok.... visualização
out.println("<html>");
out.println("<body>");
out.println("Contato " + contato.getNome() +
    " adicionado com sucesso");
out.println("</body>");
out.println("</html>");

}

```

Repare que, no final do nosso método, misturamos o código HTML com Java. O que queremos extrair do código acima é justamente essas últimas linhas.

Seria muito mais interessante para o programador e para o designer ter um arquivo JSP chamado contato-adicionado.jsp apenas com o HTML:

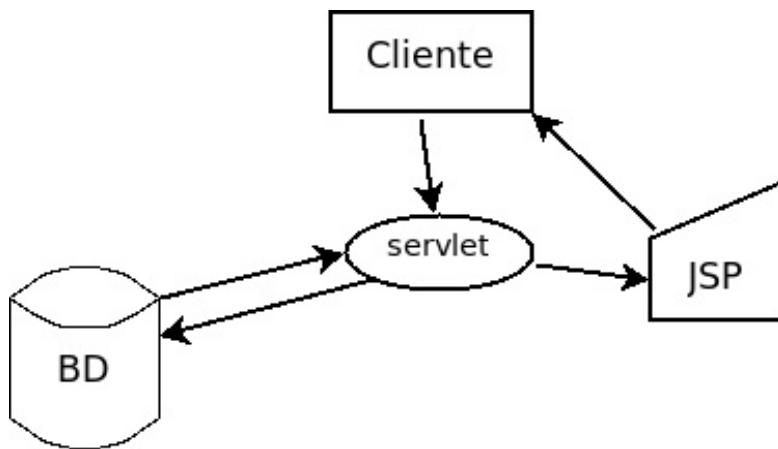
```

<html>
    <body>
        Contato ${param.nome} adicionado com sucesso
    </body>
</html>

```

Buscamos uma forma de redirecionar as requisições, capaz de encaminhar essa requisição para um outro recurso do servidor: por exemplo indo de uma servlet para um JSP.

Para isso, fazemos o **dispatch das requisições**, para que o JSP só seja renderizado depois que suas regras de negócio, dentro de uma servlet por exemplo, foram executadas.



9.2 REQUEST DISPATCHER

Poderíamos melhorar a nossa aplicação se trabalhássemos com o código Java na servlet e o HTML apenas no JSP.

A API de Servlets nos permite fazer tal redirecionamento. Basta conhecermos a URL que queremos acessar e podemos usar um objeto `RequestDispatcher` para acessar outro recurso Web, seja esse recurso uma página JSP ou uma servlet:

```
RequestDispatcher rd = request
```

```
.getRequestDispatcher("/ contato- adicionado.jsp");
rd.forward(request, response);
```

Podemos facilmente executar a lógica de nossa aplicação Web em uma servlet e então redirecionar para uma página JSP, onde você possui seu código HTML e tags que irão manipular os dados trazidos pela servlet.

FORWARD E INCLUDE

O método `forward` só pode ser chamado quando nada foi ainda escrito para a saída. No momento que algo for escrito, fica impossível redirecionar o usuário, pois o protocolo HTTP não possui meios de voltar atrás naquilo que já foi enviado ao cliente.

Existe outro método da classe `RequestDispatcher` que representa a inclusão de página e não o redirecionamento. Esse método se chama `include` e pode ser chamado a qualquer instante para acrescentar ao resultado de uma página os dados de outra.

9.3 EXERCÍCIOS: REQUESTDISPATCHER

Vamos evoluir nossa adição de contatos antes puramente usando Servlets para usar o `RequestDispatcher`.

1. Seguindo a separação aprendida nesse capítulo, queremos deixar em um JSP separado a responsabilidade de montar o HTML a ser devolvido para o usuário.

Crie então um novo arquivo **contato-adicionado.jsp** na pasta *WebContent*:

```
<html>
  <body>
    Contato ${param.nome} adicionado com sucesso
  </body>
</html>
```

2. Altere sua servlet `AdicionaContatoServlet` para que, após a execução da lógica de negócios, o fluxo da requisição seja redirecionado para nosso novo JSP.

Remova no fim da classe o código que monta a saída HTML (as chamadas de `out.println`). Vamos substituir por uma chamada ao `RequestDispatcher` e exibir o mesmo resultado usando o JSP que criamos. A chamada fica no final de nossa servlet:

```
RequestDispatcher rd = request
  .getRequestDispatcher("/ contato- adicionado.jsp");
rd.forward(request, response);
```

3. Teste a URL: <http://localhost:8080/fj21-agenda/adiciona-contato.jsp>

The image contains two screenshots of a web browser window. Both screenshots show the same URL: <http://localhost:8080/fj21-agenda/adicionaContato>.
The top screenshot shows the browser's status bar indicating "Contato José Eduardo adicionado com sucesso" (Contact José Eduardo added successfully).
The bottom screenshot shows the Caelum logo and the title "Agenda de Contatos da Caelum". It displays a form with fields for Nome (Name), E-mail (Email), Endereço (Address), and Data Nascimento (Birth Date), followed by a "Gravar" (Save) button. Below the form, a copyright notice reads "Copyright 2010 - Todos os direitos reservados".

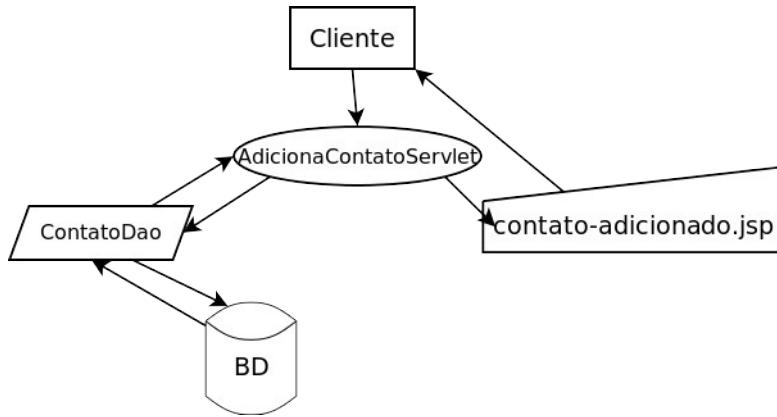
Resultado

Perceba que já atingimos um resultado que não era possível anteriormente.

Muitos projetos antigos que foram escritos em Java utilizavam somente JSP ou servlets e o resultado era assustador: diferentes linguagens misturadas num único arquivo, tornando difícil a tarefa de manutenção do código. Com o conteúdo mostrado até esse momento, é possível escrever um código com muito mais qualidade: cada tecnologia com a sua responsabilidade.

9.4 MELHORANDO O PROCESSO

Aqui temos várias servlets acessando o banco de dados, trabalhando com os DAOs e pedindo para que o JSP apresente esses dados, o diagrama a seguir mostra a representação do AdicionaContatoServlet após a modificação do exercício anterior.



Temos o problema de ter muitas servlets. Para cada lógica de negócios, teríamos uma servlet diferente, que significa várias portas de entradas, algo abominável em um projeto de verdade. Imagine dez classes de modelo, cinco lógicas diferentes, isso totaliza cinquenta formas diferentes de acesso.

E se quiséssemos fazer um *Log* das chamadas dessas lógicas? Teríamos que espalhar o código para esse *Log* por toda nossa aplicação.

Sabemos da existência de ferramentas para gerar tal código automaticamente, mas isso não resolve o problema da complexidade de administrar tantas servlets.

Utilizaremos uma ideia que diminuirá bastante o número de portas de entradas em nossa aplicação: *colocar tudo em uma Servlet só* e, de acordo com os parâmetros que o cliente usar, decidimos o que executar. Teríamos aí uma Servlet para controlar essa parte, como o esboço abaixo:

```

// Todas as lógicas dentro de uma Servlet
@WebServlet("/sistema")
public class SistemaTodoServlet extends HttpServlet {

    protected void service(HttpServletRequest request,
                           HttpServletResponse response) {

        String acao = request.getParameter("logica");
        ContatoDao dao = new ContatoDao();

        if (acao.equals("AdicionaContato")) {
            Contato contato = new Contato();
            contato.setNome(request.getParameter("nome"));
            contato.setEndereco(request.getParameter("endereco"));
            contato.setEmail(request.getParameter("email"));
            dao.adiciona(contato);

            RequestDispatcher rd =
                request.getRequestDispatcher("/contato-adicionado.jsp");
            rd.forward(request, response);
        } else if (acao.equals("ListaContatos")) {
            // busca a lista no DAO
            // despacha para um jsp
        } else if (acao.equals("RemoveContato")) {
            // faz a remoção e redireciona para a lista
        }
    }
}

```

```
}
```

Poderíamos acessar no navegador algo como `http://localhost:8080/<contexto>/sistema?logica=AdicionaContato`.

Mas para cada ação teríamos um `if / else if`, tornando a Servlet muito grande, com toda regra de negócio do sistema inteiro.

Podemos melhorar fazendo *refactoring* de extrair métodos. Mas continuariam com uma classe muito grande.

Seria melhor colocar cada regra de negócio (como inserir contato, remover contato, fazer relatório etc) em uma classe separada. Cada ação (regra de negócio) em nossa aplicação estaria em uma classe.

Então vamos extrair a nossa lógica para diferentes classes, para que nossa Servlet pudesse ter um código mais enxuto como esse:

```
if (acao.equals("AdicionaContato")) {  
    new AdicionaContato().executa(request, response);  
} else if (acao.equals("ListaContato")) {  
    new ListaContatos().executa(request, response);  
}
```

E teríamos classes `AdicionaContato`, `ListaContatos`, etc com um método (digamos, `executa`) que faz a lógica de negócios apropriada.

Porém, a cada lógica nova, lógica removida, alteração etc, temos que alterar essa servlet. Isso é trabalhoso e muito propenso a erros.

Repare dois pontos no código acima. Primeiro que ele possui o mesmo comportamento de `switch!` E `switch` em Java quase sempre pode ser substituído com vantagem por polimorfismo, como veremos a seguir. Outra questão é que recebemos como parâmetro justamente o nome da classe que chamamos em seguida.

Vamos tentar generalizar então, queremos executar o seguinte código:

```
String nomeDaClasse = request.getParameter("logica");  
new nomeDaClasse().executa(request, response);
```

Queremos pegar o nome da classe a partir do parâmetro e instanciá-la. Entretanto não podemos, pois `nomeDaClasse` é o nome de uma variável e o código acima não é válido. O nosso problema é que só sabemos o que vamos instanciar em tempo de execução (quando o parâmetro chegar) e não em tempo de compilação.

Mas a partir do nome da classe nós podemos recuperar um objeto que representará as informações contidas dentro daquela classe, como por exemplo atributos, métodos e construtores. Para que consigamos esse objeto, basta utilizarmos a classe `Class` invocando o método `forName` indicando de qual classe queremos uma representação. Isso nos retornará um objeto do tipo `Class` representando a

classe. Como abaixo:

```
String nomeDaClasse = "br.com.caelum.mvc.logica." +
    request.getParameter("logica");
Class classe = Class.forName(nomeDaClasse);
```

Ótimo, podemos ter uma representação de `AdicionaContato` ou de `ListaContato` e assim por diante. Mas precisamos de alguma forma instanciar essas classes.

Já que uma das informações guardadas pelo objeto do tipo `Class` é o construtor, nós podemos invocá-lo para instanciar a classe através do método `newInstance`.

```
Object objeto = classe.newInstance();
```

E como chamar o método `executa`? Repare que o tipo declarado do nosso objeto é `Object`. Dessa forma, não podemos chamar o método `executa`. Uma primeira alternativa seríamos fazer novamente `if/else` para sabermos qual é a lógica que está sendo invocada, como abaixo:

```
String nomeDaClasse = "br.com.caelum.mvc." +
    request.getParameter("logica");
Class classe = Class.forName(nomeDaClasse);

Object objeto = classe.newInstance();

if (nomeDaClasse.equals("br.com.caelum.mvc.AdicionaContato")) {
    ((AdicionaContato) objeto).executa(request, response);
} else if (nomeDaClasse.equals("br.com.caelum.mvc.ListaContatos")) {
    ((ListaContatos) objeto).executa(request, response);
} //e assim por diante
```

Mas estamos voltando para o `if/else` que estávamos fugindo no começo. Isso não é bom. Todo esse `if/else` é ocasionado por conta do tipo de retorno do método `newInstance` ser `Object` e nós tratarmos cada uma de nossas lógicas através de um tipo diferente.

Repare que, tanto `AdicionaContato` quanto `ListaContatos`, são consideradas `Logicas` dentro do nosso contexto. O que podemos fazer então é tratar ambas como algo que siga o contrato de `Logica` implementando uma interface de mesmo nome que declare o método `executa`:

```
public interface Logica {
    void executa(HttpServletRequest req,
                  HttpServletResponse res)
        throws Exception;
}
```

Podemos simplificar nossa `Servlet` para executar a lógica de forma polimórfica e, tudo aquilo que fazíamos em aproximadamente 8 linhas de código, podemos fazer em apenas 2:

```
Logica logica = (Logica) classe.newInstance();
logica.executa(request, response);
```

Dessa forma, uma lógica simples para logar algo no console poderia ser equivalente a:

```
public class PrimeiraLogica implements Logica {
    public void executa(HttpServletRequest req,
```

```

        HttpServletResponse res)
        throws Exception {
    System.out.println("Executando a logica e redirecionando...");
    RequestDispatcher rd = req
        .getRequestDispatcher("primeira-logica.jsp");
    rd.forward(req, res);
}
}

```

Alguém precisa controlar então que ação será executada para cada requisição, e que JSP será utilizado. Podemos usar uma servlet para isso, e então ela passa a ser a servlet controladora da nossa aplicação, chamando a ação correta e fazendo o dispatch para o JSP desejado.

Melhorando ainda mais nossa servlet controladora, poderíamos deixar nela a responsabilidade de nos redirecionar para uma página JSP ou para qualquer outra lógica ao final da execução das lógicas, bastando que o método `executa` retorne um simples `String`, eliminando toda a repetição de código `RequestDispatcher`.

Começaríamos alterando a assinatura do método `executa` da interface `Logica` que era `void` e agora retornará `String`:

```

public interface Logica {
    String executa(HttpServletRequest req, HttpServletResponse res)
        throws Exception;

}

```

Depois faríamos as lógicas retornarem um `String` com o nome do `.jsp` que deve ser chamado ao final das suas execuções.

```

public class PrimeiraLogica implements Logica {
    public String executa(HttpServletRequest req,
        HttpServletResponse res)
        throws Exception {
    System.out.println("Executando a logica e redirecionando...");
    return "primeira-logica.jsp";
}
}

```

Por fim, a servlet controladora deve receber esse `String` e implementar o código de `RequestDispatcher`:

```

@.WebServlet("/sistema")
public class ControllerServlet extends HttpServlet {

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

    String parametro = request.getParameter("logica");
    String nomeDaClasse = "br.com.caelum.mvc.logica." + parametro;

    try {
        Class<?> classe = Class.forName(nomeDaClasse);
        Logica logica = (Logica) classe.newInstance();
    }
}

```

```

    // Recebe o String após a execução da lógica
    String pagina = logica.executa(request, response);

    // Faz o forward para a página JSP
    request.getRequestDispatcher(pagina).forward(request, response);

} catch (Exception e) {
    throw new ServletException(
        "A lógica de negócios causou uma exceção", e);
}
}
}
}

```

9.5 RETOMANDO O DESIGN PATTERN FACTORY

Note que o método `forName` da classe `Class` retorna um objeto do tipo `Class`, mas esse objeto é novo? Foi reciclado através de um cache desses objetos?

Repare que não sabemos o que acontece exatamente dentro do método `forName`, mas ao invocá-lo e a execução ocorrer com sucesso, sabemos que a classe que foi passada em forma de `String` foi lida e inicializada dentro da virtual machine.

Na primeira chamada a `Class.forName` para determinada classe, ela é inicializada. Já em uma chamada posterior, `Class.forName` devolve a classe que já foi lida e está na memória, tudo isso sem que afete o nosso código.

Esse exemplo do `Class.forName` é ótimo para mostrar que qualquer código que isola a instanciação através de algum recurso diferente do construtor é uma **factory**.

9.6 EXERCÍCIOS: CRIANDO NOSSAS LÓGICAS E A SERVLET DE CONTROLE

- Crie a sua interface no pacote `br.com.caelum.mvc.logica`:

```

public interface Logica {
    String executa(HttpServletRequest req,
                    HttpServletResponse res) throws Exception;
}

```

- Crie uma implementação da interface `Logica`, nossa classe `PrimeiraLogica`, também no pacote `br.com.caelum.mvc.logica`:

```

public class PrimeiraLogica implements Logica {
    public String executa(HttpServletRequest req,
                          HttpServletResponse res) throws Exception {

        System.out.println("Executando a logica ...");

        System.out.println("Retornando o nome da página JSP ...");
        return "primeira-logica.jsp";
    }
}

```

```
}
```

3. Faça um arquivo JSP chamado `primeira-logica.jsp` dentro do diretório `WebContent` :

```
<html>
    <body>
        <h1> Página da nossa primeira lógica </h1>
    </body>
</html>
```

4. Vamos escrever nossa Servlet que coordenará o fluxo da nossa aplicação.

Crie sua servlet chamada `ControllerServlet` no pacote `br.com.caelum.mvc.servlet` :

```
@WebServlet("/mvc")
public class ControllerServlet extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, IOException {

        String parametro = request.getParameter("logica");
        String nomeDaClasse = "br.com.caelum.mvc.logica." + parametro;

        try {
            Class classe = Class.forName(nomeDaClasse);

            Logica logica = (Logica) classe.newInstance();
            String pagina = logica.executa(request, response);

            request.getRequestDispatcher(pagina).forward(request, response);
        } catch (Exception e) {
            throw new ServletException(
                "A lógica de negócios causou uma exceção", e);
        }
    }
}
```

5. Teste a url <http://localhost:8080/fj21-agenda/mvc?logica=PrimeiraLogica>



9.7 EXERCÍCIOS: CRIANDO UMA LÓGICA PARA REMOVER CONTATOS

1. Crie uma nova classe chamada `RemoveContatoLogic` no mesmo pacote `br.com.caelum.mvc.logica`. Devemos implementar a interface `Logica` e durante sua execução receberemos um `id` pelo `request` e removeremos o contato no banco a partir deste `id`.

```
public class RemoveContatoLogic implements Logica {
```

```

public String executa(HttpServletRequest req, HttpServletResponse res)
    throws Exception {

    long id = Long.parseLong(req.getParameter("id"));

    Contato contato = new Contato();
    contato.setId(id);

    ContatoDao dao = new ContatoDao();
    dao.exclui(contato);

    System.out.println("Excluindo contato... ");

    return "lista-contatos.jsp";
}

```

2. Na página `lista-contatos.jsp`, vamos acrescentar uma coluna na tabela que lista os contatos com um link chamando a lógica de remoção e passando o `id` do contato:

```

<!-- código omitido -->

<c:forEach var="contato" items="${dao.lista}">
<tr>

    <!-- código omitido -->

    <td>
        <a href="mvc?logica=RemoveContatoLogic&id=${contato.id}">Remover</a>
    </td>
</tr>
</c:forEach>

<!-- código omitido -->

```

3. Teste a logica de remoção acessando `http://localhost:8080/fj21-agenda/lista-contatos.jsp` e clicando em algum link **Remover**.

9.8 FAZENDO A LÓGICA PARA LISTAR OS CONTATOS

Agora que todo nosso processamento está passando pela Servlet controladora e conseguimos organizar nosso código em camadas bem definidas, nos deparamos com uma situação que ainda está um pouco distante do ideal.

Se olharmos a página `lista-contatos.jsp` veremos que para fazer a listagem dos contatos funcionar estamos criando uma instância da classe `ContatoDao` para utilizá-la depois no `<c:forEach>` recuperando uma lista de contatos.

```

<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDao" />

<table>
    <c:forEach var="contato" items="${dao.lista}">
        <tr>
            <td>${contato.nome}</td>
            <!-- código omitido -->

```

```

        </tr>
    </c:forEach>
</table>
```

Instanciar objetos da camada *Model* na camada *View* não é considerada uma boa prática na arquitetura MVC (*antipattern*).

Podemos resolver facilmente isso transferindo essa responsabilidade de montar a lista de contatos para uma lógica `ListaContatosLogic` e depois passá-la pronta direto para o JSP pelo *request*.

Para guardarmos algo na requisição, precisamos invocar o método `.setAttribute()` no *request*. Passamos para esse método uma identificação para o objeto que estamos guardando na requisição e também passamos o próprio objeto para ser guardado no *request*.

```

public class ListaContatosLogic implements Logica {

    public String executa(HttpServletRequest req, HttpServletResponse res)
        throws Exception {

        // Monta a lista de contatos
        List<Contato> contatos = new ContatoDao().getLista();

        // Guarda a lista no request
        req.setAttribute("contatos", contatos);

        return "lista-contatos.jsp";
    }
}
```

Agora é só ajustar a página `lista-contatos.jsp` para não instanciar mais o `ContatoDao`, removendo a linha `<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDao" />`, e depois fazer com que o `<c:forEach>` use a lista de contatos que foi colocada no *request*:

```
<c:forEach var="contato" items="${contatos}">
```

9.9 EXERCÍCIOS: LÓGICA PARA LISTAR CONTATOS

- Crie uma nova classe chamada `ListaContatosLogic` no mesmo pacote `br.com.caelum.mvc.logica`. Devemos implementar nela a interface `Logica` e, durante sua execução vamos criar uma lista de contatos através de uma instância da classe `ContatoDao`, guardá-la no *request* e retornar para a servlet controladora:

```

public class ListaContatosLogic implements Logica {

    public String executa(HttpServletRequest req, HttpServletResponse res)
        throws Exception {

        List<Contato> contatos = new ContatoDao().getLista();

        req.setAttribute("contatos", contatos);

        return "lista-contatos.jsp";
    }
}
```

2. Agora, vamos modificar a página `lista-contatos.jsp` para não instanciar mais `ContatoDao` na `View`, removendo a linha `<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDao" />`, e alterar o `<c:forEach>` para usar a lista de contatos que foi colocada pela lógica no `request` ao invés de `${dao.lista}` :

```
<c:forEach var="contato" items="${contatos}">
```

3. Agora podemos testar chamando: <http://localhost:8080/fj21-agenda/mvc?logica=ListaContatosLogic>

4. Depois dessas alterações, será necessário alterar o retorno da classe `RemoveContatoLogic` pois agora a chamada direta do `lista-contatos.jsp` não é mais possível. Devemos agora chamar a lógica que lista os contatos:

```
public class RemoveContatoLogic implements Logica {

    public String executa(HttpServletRequest req, HttpServletResponse res)
        throws Exception {

        // código omitido

        return "mvc?logica=ListaContatosLogic";
    }

}
```

9.10 ESCONDENDO NOSSAS PÁGINAS

Como alteramos nossa listagem para ser acessada pela lógica `ListaContatosLogic`, se acessarmos a jsp `lista-contatos.jsp` diretamente pelo navegador, a página não mostrará nenhum contato. Precisamos então sempre passar pela lógica, que por sua vez disponibilizará a listagem para a página.

Portanto, não devemos permitir que o usuário acesse diretamente nossa página. Para impossibilitar este acesso direto, colocaremos nossas páginas dentro do diretório **WEB-INF/jsp**.

Agora que estamos usando MVC, uma boa prática é não deixarmos os usuários acessarem nossas páginas diretamente, e sim passando sempre por uma lógica.

Nossa lógica de listagem ficará da seguinte forma:

```
public class ListaContatosLogic implements Logica {

    public String executa(HttpServletRequest req, HttpServletResponse res)
        throws Exception {

        List<Contato> contatos = new ContatoDao().getLista();

        req.setAttribute("contatos", contatos);

        return "/WEB-INF/jsp/lista-contatos.jsp";
    }
}
```

Além disso, se quisermos aplicar este mesmo conceito para as demais jsps, precisaremos alterar as demais lógicas correspondentes acrescentando o diretório `WEB-INF/jsp` antes do nome da página.

9.11 EXERCÍCIOS OPCIONAIS

1. Crie uma lógica chamada `AlteraContatoLogic` e teste a mesma através de um link na listagem da `lista-contatos.jsp`. Lembre-se, antes de chamar essa lógica é preciso criar uma outra lógica que mostre os dados do contato em uma nova página, permitindo assim a alteração dos dados, e só depois, no clique de um botão, que a alteração será de fato efetivada.
2. Crie a lógica de adicionar contatos (`AdicionaContatoLogic`). Repare que ela é bem parecida com a `AlteraContatoLogic`. Crie um formulário de adição de novo contato. Coloque um link para adicionar novos contatos dentro do `lista-contatos.jsp`.
3. **Desafio:** As lógicas de adição e de alteração ficaram muito parecidas. Tente criar uma versão de uma dessas lógicas que faça as duas. Dica: A única diferença entre as duas é a presença ou não do parâmetro `Id`.
4. **Desafio:** Altere seu projeto para que nenhuma jsp seja acessível diretamente, colocando-as no diretório `WEB-INF/jsp`. Modifique também suas lógicas de acordo. OBS: Deverá ser criada uma nova lógica para a visualização do formulário de adição de contatos.

9.12 MODEL VIEW CONTROLLER

Generalizando o modelo acima, podemos dar nomes a cada uma das partes dessa nossa arquitetura. Quem é responsável por apresentar os resultados na página web é chamado de Apresentação (**View**).

A servlet (e auxiliares) que faz os dispatches para quem deve executar determinada tarefa é chamada de Controladora (**Controller**).

As classes que representam suas entidades e as que te ajudam a armazenar e buscar os dados são chamadas de Modelo (**Model**).

Esses três formam um padrão arquitetural chamado de **MVC**, ou **Model View Controller**. Ele pode sofrer variações de diversas maneiras. O que o MVC garante é a separação de tarefas, facilitando assim a reescrita de alguma parte, e a manutenção do código.

O famoso **Struts** ajuda você a implementar o **MVC**, pois tem uma controladora já pronta, com uma série de ferramentas para te auxiliar. O **Hibernate** pode ser usado como **Model**, por exemplo. E como **View** você não precisa usar só **JSP**, pode usar a ferramenta **Velocity**, por exemplo.

9.13 LISTA DE TECNOLOGIAS: CAMADA DE CONTROLE

Há diversas opções para a camada de controle no mercado. Veja um pouco sobre algumas delas:

- **Struts Action** - o controlador mais famoso do mercado Java, é utilizado principalmente por ser o mais divulgado e com tutoriais mais acessíveis. Possui vantagens características do MVC e desvantagens que na época ainda não eram percebidas. É o controlador pedido na maior parte das vagas em Java hoje em dia. É um projeto que não terá grandes atualizações pois a equipe dele se juntou com o WebWork para fazer o Struts 2, nova versão do Struts incompatível com a primeira e totalmente baseada no WebWork.
- **VRaptor** - desenvolvido inicialmente por profissionais da Caelum e baseado em diversas ideias dos controladores mencionados acima, o VRaptor usa o conceito de favorecer Convenções em vez de Configurações para minimizar o uso de XML e anotações em sua aplicação Web.
- **JSF** - JSF é uma especificação Java para frameworks MVC. Ele é baseado em componentes e possui várias facilidades para desenvolver a interface gráfica. Devido ao fato de ser um padrão oficial, ele é bastante adotado. O JSF é ensinado no curso FJ-22 e em detalhes no nosso curso FJ-26.
- **Spring MVC** - é uma parte do *Spring Framework* focado em implementar um controlador MVC. É fácil de usar em suas últimas versões e tem a vantagem de se integrar a toda a estrutura do Spring com várias tecnologias disponíveis.

9.14 LISTA DE TECNOLOGIAS: CAMADA DE VISUALIZAÇÃO

Temos também diversas opções para a camada de visualização. Um pouco sobre algumas delas:

- **JSP** - como já vimos, o JavaServer Pages, temos uma boa ideia do que ele é, suas vantagens e desvantagens. O uso de *taglibs* (a JSTL por exemplo) e expression language é muito importante se você escolher JSP para o seu projeto. É a escolha do mercado hoje em dia.
- **Velocity** - um projeto antigo, no qual a EL do JSP se baseou, capaz de fazer tudo o que você precisa para a sua página de uma maneira extremamente compacta. Indicado pela Caelum para conhecer um pouco mais sobre outras opções para camada de visualização.
- **Freemarker** - similar ao Velocity e com ideias do JSP - como suporte a taglibs - o freemarker vem sendo cada vez mais utilizado, ele possui diversas ferramentas na hora de formatar seu texto que facilitam muito o trabalho do designer.
- **Sitemesh** - não é uma alternativa para as ferramentas anteriores mas sim uma maneira de criar templates para seu site, com uma ideia muito parecida com o *struts tiles*, porém genérica: funciona inclusive com outras linguagens como PHP etc.

Em pequenas equipes, é importante uma conversa para mostrar exemplos de cada uma das tecnologias acima para o designer, afinal quem vai trabalhar com as páginas é ele. A que ele preferir,

você usa, afinal todas elas fazem o mesmo de maneiras diferentes. Como em um projeto é comum ter poucos designers e muitos programadores, talvez seja proveitoso facilitar um pouco o trabalho para aqueles.

9.15 DISCUSSÃO EM AULA: OS PADRÕES COMMAND E FRONT CONTROLLER

RECURSOS IMPORTANTES: FILTROS

"A arte nunca está terminada, apenas abandonada." -- Leonardo Da Vinci

Ao término desse capítulo, você será capaz de:

- criar classes que filtram a requisição e a resposta;
- guardar objetos na requisição;
- descrever o que é injeção de dependências;
- descrever o que é inversão de controle;

10.1 REDUZINDO O ACOPLAMENTO COM FILTROS

Em qualquer aplicação surgem requisitos que não são diretamente relacionados com a regra de negócio. Um exemplo clássico desses requisitos não funcionais é a auditoria (*Logging*). Queremos logar as chamadas de uma lógica da nossa aplicação. Outros exemplos são autorização, tratamento de erro ou criptografia. Existem vários outros, mas o que todos eles tem em comum é que não são relacionados com as regras de negócios.

A pergunta como implementar estas funcionalidades, nos vem a cabeça. A primeira ideia seria colocar o código diretamente na classe que possui a lógica. A classe seguinte mostra através de pseudocódigo as chamadas para fazer auditoria e autorização:

```
public class RemoveContatoLogic implements Logica {

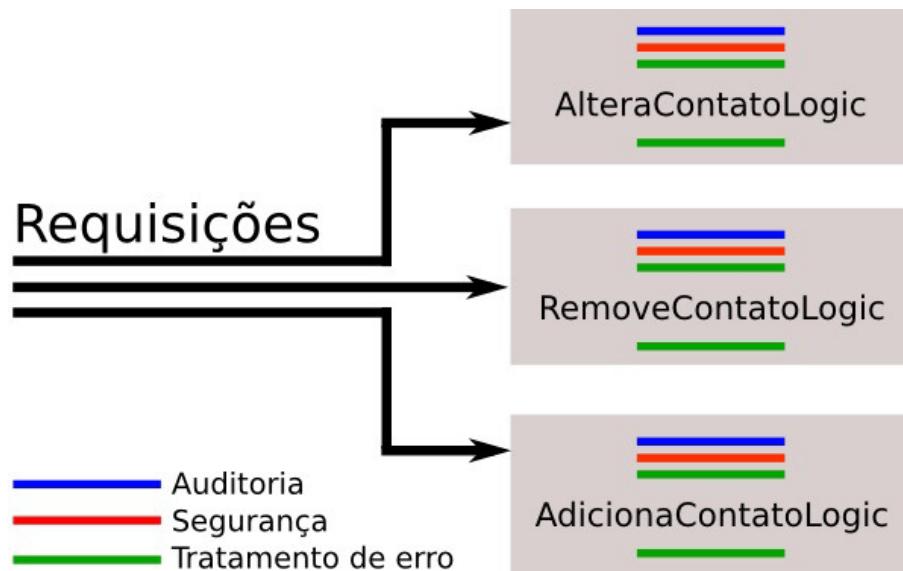
    public void executa(HttpServletRequest request,
                         HttpServletResponse response)
        throws Exception {

        // auditoria
        Logger.info("acessando remove contato logic");

        // autorização
        if(!usuario.ehCliente()) {
            request.getRequestDispatcher("/acessoNegado.jsp")
                .forward(request, response);
        }

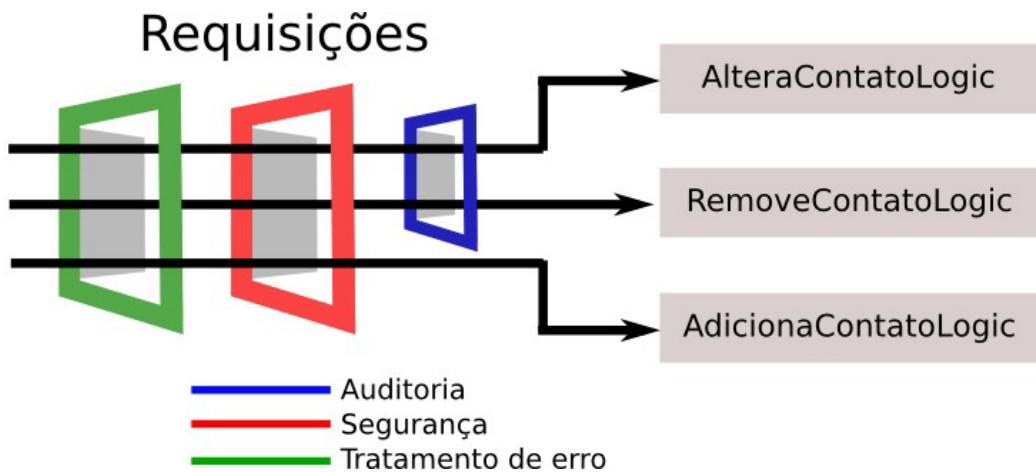
        // toda lógica para remover o contato aqui
        // ...
    }
}
```

Podemos ver que além da lógica é preciso implementar os outros requisitos, mas não só apenas na lógica que altera o contato, também é necessário colocar o mesmo código nas lógicas que adiciona, remove ou faz a listagem dos contatos. Isso pode piorar pensando que existem muito mais requisitos não funcionais, como resultado o código aumenta em cada lógica. Desse jeito criamos um acoplamento muito forte entre a logica e a implementação dos requisitos não funcionais. O grande problema é que os mesmos ficam espalhados em todas as lógicas. A imagem abaixo ilustra esse acoplamento:



A API de Servlets nos provê um mecanismo para tirar esse acoplamento e isolar esse comportamento, que são os **Filtros**. Filtros são classes que permitem que executemos código antes da requisição e também depois que a resposta foi gerada.

Uma boa analogia é pensar que as lógicas são quartos em uma casa. Para acessar um quarto é preciso passar por várias portas. As portas são os filtros, onde você passa na ida e na volta. Cada filtro encapsula apenas uma responsabilidade, ou seja um filtro para fazer auditoria, outro para fazer a segurança etc. Então é possível usar vários filtros em conjunto. Uma porta também pode ficar fechada, caso o usuário não possua acesso a lógica, ou seja, o filtro pode negar a execução de uma lógica. Veja a imagem seguinte que mostra os filtros aplicados no exemplo anterior:



A grande vantagem é que cada requisito fica em um lugar só e conseguimos desacoplar nossas lógicas.

Para criarmos filtros utilizando a API de Servlets do Java EE 5, temos as mesmas dificuldades que temos quando vamos definir Servlets . Para cada filtro é necessário criarmos a classe que implementa a interface `javax.servlet.Filter` e depois declararmos o filtro no `web.xml` , além de termos que declarar para quais URL's aquele filtro será aplicado.

Configuração de um filtro no `web.xml` :

```
<filter>
    <filter-name>meuFiltro</filter-name>
    <filter-class>br.com.caelum.filtro.Meufiltro</filter-class>
</filter>

<filter-mapping>
    <filter-name>meuFiltro</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Mas para definirmos um filtro usando a nova API de Servlets do Java EE 6, basta apenas criarmos uma classe que implementa a interface `javax.servlet.Filter` e anotarmos a classe com `@WebFilter` . Podemos aproveitar e passar como parâmetro na anotação o padrão de requisições que serão filtradas:

```
@WebFilter("/oi")
public class Meufiltro implements Filter {
    public void doFilter(ServletRequest req,
        ServletResponse res, FilterChain chain) {
        // ...
    }
}
```

Desta forma indicamos que todas as requisições vindas a partir de `/oi` serão filtradas e, portanto, o filtro será aplicado em cada requisição.

É possível usar um filtro mais específico. Por exemplo, podemos filtrar todas as requisições para

paginas JSPs:

```
@WebFilter("*.jsp")
public class MeuFiltro implements Filter {
    public void doFilter(ServletRequest req,
        ServletResponse res, FilterChain chain) {
        // ...
    }
}
```

Ou um filtro mais amplo, filtrando TODAS as requisições da aplicação:

```
@WebFilter("/*")
public class MeuFiltro implements Filter {
    public void doFilter(ServletRequest req,
        ServletResponse res, FilterChain chain) {
        // ...
    }
}
```

Ao implementar a interface `Filter`, temos que implementar 3 métodos: `init`, `destroy` e `doFilter`.

Os métodos `init` e `destroy` possuem a mesma função dos métodos de mesmo nome da `Servlet`, ou seja, executar algo quando o seu filtro é carregado pelo container e quando é descarregado pelo container.

O método que fará todo o processamento que queremos executar é o `doFilter`, que recebe três parâmetros: `ServletRequest`, `ServletResponse` e `FilterChain`.

```
@WebFilter("/*")
public class FiltroTempoDeExecucao implements Filter {

    // implementação do init e destroy

    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        // todo o processamento vai aqui
    }
}
```

Perceba a semelhança desse método com o método `service` da classe `Servlet`. A ideia do filtro é também processar requests, mas ele poderá fazer isso de maneira mais genérica para vários tipos de requests. Mas um filtro vai além de um servlet, com um filtro podemos também fechar "a porta". Esse poder vem do argumento `FilterChain` (a cadeia de filtros). Ele nos permite indicar ao container que o request deve prosseguir seu processamento. Isso é feito com uma chamada do método `doFilter` da classe `FilterChain`:

```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {

    // passa pela porta
}
```

```
        chain.doFilter(request, response);
    }
```

Um filtro não serve para processar toda a requisição. A ideia é ele *interceptar* vários *requests* semelhantes, executar algo, mas depois permitir que o processamento normal do *request* prossiga através das Servlets e JSPs normais.

Qualquer código colocado antes da chamada `chain.doFilter(request, response)` será executado na ida, qualquer código depois na volta. Com isso podemos fazer um verificação de acesso antes da lógica, ou abrir um recurso (conexão ou transação) antes e na volta fechar o mesmo. Um filtro é ideal para fazer tratamento de error ou medir o tempo de execução.

A única coisa que precisamos fazer para que o nosso filtro funcione é registrá-lo, para que o container saiba que ele precisa ser executado. Fazemos isso de forma simples, usando atributos da anotação `@WebFilter`. Através dos atributos, declaramos o filtro e quais URLs ou Servlets serão filtradas.

O atributo `filterName` define um nome (ou *alias*) para o filtro. Se não definirmos o atributo `filterName` para nosso filtro, o nome dele será o nome completo da classe, da mesma forma que acontece com as `Servlets`. Para definirmos quais URL passarão pelo nosso filtro, podemos fazê-lo de maneira similar à anotação `@WebServlet`. Se quisermos definir aquele filtro para apenas uma URL, passamos através do parâmetro na anotação `@WebFilter` como foi feito no exemplo acima - `@WebFilter("/oi")`. Mas, se quisermos definir que mais de uma URL será filtrada, podemos usar o atributo `urlPatterns`:

```
@WebFilter(filterName = "MeuFiltro", urlPatterns = {"/oi", "/ola"})
public class MeuFiltro implements Filter {
    public void doFilter(ServletRequest req,
                         ServletResponse res, FilterChain chain) {
        // ...
    }
}
```

Podemos ainda configurar quais servlets serão filtrados por aquele filtro declarando seus nomes no atributo `servletNames`. Por exemplo:

```
@WebFilter(filterName = "MeuFiltro",
            servletNames = {"meuServlet", "outroServlet"})
public class MeuFiltro implements Filter {
    public void doFilter(HttpServletRequest req,
                         HttpServletResponse res, FilterChain chain) {
        // ...
    }
}
```

OUTRAS ANOTAÇÕES

Existem outras anotações presentes na API Servlets 3.0:

- **@WEBLISTENER** - Utilizada para definir Listeners de eventos que podem ocorrer em vários pontos da sua aplicação; equivale ao de hoje;
- **@WEBINITPARAM** - Utilizada para especificar parâmetros de inicialização que podem ser passados para Servlets e Filtros . Pode ser passada como parâmetro para as anotações `@WebServlet` e `@WebFilter` ; equivale ao de hoje;
- **@MULTIPARTCONFIG** - Utilizada para definir que um determinado Servlet receberá uma requisição do tipo `mime/multipart` .

10.2 EXERCÍCIOS OPCIONAIS: FILTRO PARA MEDIR O TEMPO DE EXECUÇÃO

1. Vamos criar o nosso filtro para medir o tempo de execução de uma requisição.

- Crie uma nova classe chamada `FiltroTempoDeExecucao` no pacote `br.com.caelum.agenda.filtro` e faça ela implementar a interface `javax.servlet.Filter`
- Anote-a com `@WebFilter` e diga que TODAS as requisições para a nossa aplicação devem ser filtradas (`@WebFilter("/*")`).
- Deixe os métodos `init` e `destroy` vazios e implemente o `doFilter` :

```
@WebFilter("/*")
public class FiltroTempoDeExecucao implements Filter {
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        long tempoInicial = System.currentTimeMillis();

        chain.doFilter(request, response);

        long tempoFinal = System.currentTimeMillis();
        String uri = ((HttpServletRequest)request).getRequestURI();
        String parametros = ((HttpServletRequest) request)
            .getParameter("logica");
        System.out.println("Tempo da requisicao de " + uri
            + "?logica="
            + parametros + " demorou (ms): "
            + (tempoFinal - tempoInicial));

    }
    // métodos init e destroy omitidos
}
```

- Reinicie o servidor e acesse <http://localhost:8080/fj21-agenda/mvc?logica=ListaContatosLogic>

Procure a saída no console.

10.3 PROBLEMAS NA CRIAÇÃO DAS CONEXÕES

Nossa aplicação de agenda necessita, em vários momentos, de uma conexão com o banco de dados e, para isso, o nosso `DAO` invoca em seu construtor a `ConnectionFactory` pedindo para a mesma uma nova conexão. Mas em qual lugar ficará o fechamento da conexão?

```
public class ContatoDao {  
    private Connection connection;  
  
    public ContatoDao() {  
        this.connection = new ConnectionFactory().getConnection();  
    }  
  
    // métodos adiciona, remove, getLista etc  
    // onde fechamos a conexão?  
}
```

Até o momento, não estamos nos preocupando com o fechamento das conexões com o banco de dados. Isso é uma péssima prática para uma aplicação que vai para produção. Estamos deixando conexões abertas e sobrecarregando o servidor de banco de dados, que pode aguentar apenas um determinado número de conexões abertas, o que fará com que sua aplicação pare de funcionar subitamente.

Outro problema que temos ao seguir essa estratégia de adquirir conexão no construtor dos `DAOs` é quando queremos usar dois `DAOs` diferentes, por exemplo `ContatoDao` e `FornecedorDao`. Ao instanciarmos ambos os `DAOs`, vamos abrir duas conexões, enquanto poderíamos abrir apenas uma conexão para fazer as duas operações. De outra forma fica impossível realizar as operações numa única transação.

10.4 TENTANDO OUTRAS ESTRATÉGIAS

Já percebemos que não é uma boa ideia colocarmos a criação da conexão no construtor dos nossos `DAOs`. Mas qual é o lugar ideal para criarmos essa conexão com o banco de dados?

Poderíamos criar a conexão dentro de cada método do `DAO`, mas nesse caso, se precisássemos usar dois métodos diferentes do `DAO` em uma mesma operação, novamente abriríamos mais de uma conexão. Dessa forma, abrir e fechar as conexões dentro dos métodos dos `DAOs` também não nos parece uma boa alternativa.

Precisamos, de alguma forma, criar a conexão e fazer com que essa mesma conexão possa ser usada por todos os seus `DAOs` em uma determinada requisição. Assim, podemos criar nossas conexões com o banco de dados dentro de nossas `Servlets` (ou lógicas, no caso do nosso framework MVC visto no

capítulo anterior) e apenas passá-las para o DAO que vamos utilizar. Para isso, nossos DAOs deverão ter um construtor que receba Connection .

Vejamos:

```
public class ContatoDao {  
    private Connection connection;  
  
    public ContatoDao(Connection connection) {  
        this.connection = connection;  
    }  
  
    // métodos adiciona, remove, getLista etc  
}  
  
public class AdicionaContatoLogic implements Logica {  
    public String executa(HttpServletRequest request,  
                          HttpServletResponse response) {  
        Contato contato = // contato montado com os dados do request  
  
        Connection connection = new ConnectionFactory()  
            .getConnection();  
  
        // passa conexão pro construtor  
        ContatoDao dao = new ContatoDao(connection);  
        dao.adiciona(contato);  
  
        connection.close();  
  
        // retorna para o JSP  
    }  
}
```

Isso já é uma grande evolução com relação ao que tínhamos no começo mas ainda não é uma solução muito boa. Acoplamos com a ConnectionFactory todas as nossas lógicas que precisam utilizar DAOs . E, em orientação a objetos, não é uma boa prática deixarmos nossas classes com acoplamento alto.

INJEÇÃO DE DEPENDÊNCIAS E INVERSÃO DE CONTROLE

Ao não fazermos mais a criação da conexão dentro do ContatoDao mas sim recebermos a Connection da qual dependemos através do construtor, dizemos que o nosso DAO não tem mais o controle sobre a criação da Connection e, por isso, estamos **Invertendo o Controle** dessa criação. A única coisa que o nosso DAO diz é que ele depende de uma Connection através do construtor e, por isso, o nosso DAO precisa que esse objeto do tipo Connection seja recebido, ou seja, ele espera que a **dependência seja injetada**.

Injeção de Dependências e Inversão de Controle são conceitos muito importantes nas aplicações atuais e nós as estudaremos com mais detalhes ainda no curso.

10.5 REDUZINDO O ACOPLAMENTO COM FILTROS

Não queremos também que a nossa lógica conheça a classe `ConnectionFactory` mas, ainda assim, precisamos que ela possua a conexão para que possamos repassá-la para o `DAO`.

Para diminuirmos esse acoplamento, queremos que, sempre que chegar uma requisição para a nossa aplicação, uma conexão seja aberta e, depois que essa requisição for processada, ela seja fechada. Também podemos adicionar o tratamento a transação aqui, se acharmos necessário. Precisamos então interceptar toda requisição para executar esses procedimentos.

Como já visto os **Filtros** permitem que executemos código antes da requisição e também depois que a resposta foi gerada. Ideal para abrir uma conexão antes e fechar na volta.

Vamos então implementar um filtro com esse comportamento:

```
@WebFilter("/*")
public class FiltroConexao implements Filter {
    // implementação do init e destroy, se necessário

    public void doFilter(ServletRequest request,
                         ServletResponse response, FilterChain chain)
                         throws IOException, ServletException {

        // abre uma conexão
        Connection connection = new ConnectionFactory()
            .getConnection();

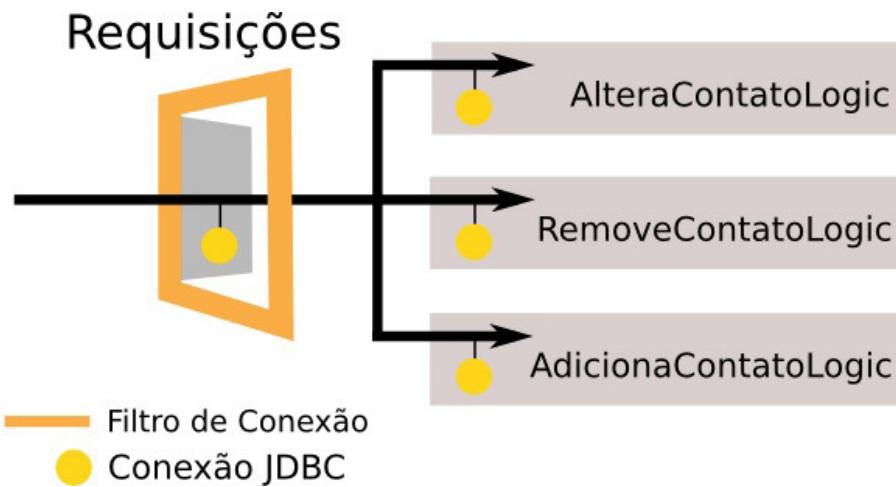
        // indica que o processamento do request deve prosseguir
        chain.doFilter(request, response);

        // fecha conexão
        connection.close();
    }
}
```

Com a conexão aberta, precisamos então fazer com que a requisição saia do nosso filtro e vá para o próximo passo, seja ele um outro filtro, ou uma Servlet ou um JSP. Dessa forma, nossa lógica de negócio pode executar normalmente. Para isso, usamos o argumento `FilterChain` que nos permite indicar ao container que o `request` deve prosseguir seu processamento. Isso é feito com uma chamada ao `doFilter` do `FilterChain`:

Até agora, conseguimos abrir uma conexão no começo dos requests, prosseguir o processamento do `request` normalmente e fechar a conexão após da execução. Mas nossas lógicas vão executar, digamos, manipulações de Contatos e vão precisar da conexão aberta no filtro. Mas como acessá-la? Como, dentro de uma Servlet, pegar um objeto criado dentro de um filtro, uma outra classe?

A ideia é associar (pendurar) de alguma forma a conexão criada ao `request` atual. Isso porque tanto o filtro quanto a Servlet estão no mesmo `request` e porque, como definimos, as conexões vão ser abertas por `requests`.



Para guardarmos algo na requisição, precisamos invocar o método `setAttribute` no `request`. Passamos para esse método uma identificação para o objeto que estamos guardando na requisição e também passamos o próprio objeto para ser guardado no `request`.

```
public void doFilter(ServletRequest request,
                     ServletResponse response, FilterChain chain) {

    Connection connection = new ConnectionFactory()
        .getConnection();

    // "pendura um objeto no Request"
    request.setAttribute("connection", connection);

    chain.doFilter(request, response);

    connection.close();
}
```

Ao invocarmos o `doFilter`, a requisição seguirá o seu fluxo normal levando o objeto `connection` junto. O que o usuário requisitou será então executado até a resposta ser gerada (por uma Servlet ou JSP). Após isso, a execução volta para o ponto imediatamente abaixo da invocação do `chain.doFilter`. Ou seja, através de um filtro conseguimos executar algo **antes** do `request` ser processado e **depois** da resposta ser gerada.

Pronto, nosso Filtro é o único ponto da nossa aplicação que criará conexões.

Repare como usamos o *wildcard* no parâmetro da anotação para indicar que todas as requisições serão filtradas e, portanto, terão uma conexão aberta já disponível.

Só falta na nossa lógica pegarmos a conexão que guardamos no `request`. Para isso basta invocarmos o método `getAttribute` no `request`. Nossa lógica ficará da seguinte maneira:

```
public class AdicionaContatoLogic implements Logica {
    public String executa (HttpServletRequest request,
                          HttpServletResponse response)
        throws Exception {
```

```

Contato contato = // contato montado com os dados do request

// buscando a conexão do request
Connection connection = (Connection) request
    .getAttribute("connection");

ContatoDao dao = new ContatoDao(connection);
dao.adiciona(contato);

// faz o return do JSP como de costume
}
}

```

Uma outra grande vantagem desse desacoplamento é que ele torna o nosso código mais fácil de se testar unitariamente, assunto que aprendemos e praticamos bastante no curso **FJ-22**.

10.6 EXERCÍCIOS: FILTROS

- Vamos criar o nosso filtro para abrir e fechar a conexão com o banco de dados

- Crie uma nova classe chamada `FiltroConexao` no pacote `br.com.caelum.agenda.filtro` e faça ela implementar a interface `javax.servlet.Filter`
- Anote-a com `@WebFilter("/*")` para registrar o filtro no *container* e fazer com que todas as requisições passem por ele;
- Deixe os métodos `init` e `destroy` vazios e implemente o `doFilter`:

```

public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {

    try {
        Connection connection = new ConnectionFactory()
            .getConnection();

        // pendurando a connection na requisição
        request.setAttribute("conexao", connection);

        chain.doFilter(request, response);

        connection.close();
    } catch (SQLException e) {
        throw new ServletException(e);
    }
}

```

- Crie um construtor no seu `ContatoDao` que receba `Connection` e armazene-a no atributo:

```

public class ContatoDao {
    private Connection connection;

    public ContatoDao(Connection connection) {
        this.connection = connection;
    }

    // outro construtor e métodos do DAO

```

}

3. Na sua `RemoveContatoLogic`, criada no capítulo anterior, busque a conexão no `request`, e repasse-a para o `DAO`. Procure na lógica a criação do `DAO` e faça as alterações:

```
public class RemoveContatoLogic implements Logica {  
    public String executa(HttpServletRequest request,  
                          HttpServletResponse response)  
        throws Exception {  
  
        // ...  
  
        // (procure o ContatoDao no código existente)  
        // busca a conexão pendurada na requisição  
        Connection connection = (Connection) request  
            .getAttribute("conexao");  
  
        // passe a conexão no construtor  
        ContatoDao dao = new ContatoDao(connection);  
  
        // ...  
    }  
}
```

Ou você pode fazer essa mesma modificação na nossa antiga `AdicionaContatoServlet`.

4. Remova um contato na sua aplicação e verifique que tudo continua funcionando normalmente.

SPRING MVC

"Um homem pinta com seu cérebro e não com suas mãos." -- Michelangelo

Nesse capítulo, você aprenderá:

- Porque utilizar frameworks;
- Como funciona o framework Spring MVC;
- As diferentes formas de se trabalhar com o Spring MVC.

11.1 PORQUE PRECISAMOS DE FRAMEWORKS MVC?

Quando estamos desenvolvendo aplicações, em qualquer linguagem, queremos nos preocupar com infraestrutura o mínimo possível. Isso não é diferente quando trabalhamos com uma aplicação Web. Imagine termos que lidar diretamente com o protocolo HTTP a todo momento que tivermos que desenvolver uma funcionalidade qualquer. Nesse ponto, os containers e a API de Servlets encapsulam o protocolo para que não precisemos lidar diretamente com ele, mas mesmo assim existe muito trabalho repetitivo que precisamos fazer para que possamos desenvolver nossa lógica.

Um exemplo desse trabalho repetitivo que fazíamos era a conversão da data. Como o protocolo HTTP sempre interpreta tudo como texto, é preciso transformar essa data em um objeto do tipo `Calendar`. Mas sempre que precisamos de uma data temos essa mesma conversão usando a `SimpleDateFormat`.

Outro exemplo é, que para gravarmos um objeto do tipo `Contato`, precisamos pegar na API de `Servlets` parâmetro por parâmetro para montar um objeto do tipo `Contato` invocando os setters adequados.

Não seria muito mais fácil que nossa lógica recebesse de alguma forma um objeto do tipo `Contato` já devidamente populado com os dados que vieram na requisição? Nosso trabalho seria apenas, por exemplo invocar o `ContatoDao` passando o `Contato` para ser adicionado.

O grande problema é que estamos atrelados a API de `Servlets` que ainda exige muito trabalho braçal para desenvolvermos nossa lógica. E, justamente para resolver esse problema, começaram a surgir os frameworks MVC, com o objetivo de diminuir o impacto da API de `Servlets` em nosso trabalho e fazer com que passemos a nos preocupar exclusivamente com a lógica de negócios, que é o código que possui valor para a aplicação.

11.2 UM POUCO DE HISTÓRIA

Logo se percebeu que o trabalho com Servlets e JSPs puros não era tão produtivo e organizado. A própria Sun começou a fomentar o uso do padrão MVC e de patterns como *Front Controller*. Era muito comum as empresas implementarem esses padrões e criarem soluções baseadas em mini-frameworks caseiros.

Mas logo se percebeu que o retrabalho era muito grande de projeto para projeto, de empresa para empresa. Usar MVC era bem interessante, mas reimplementar o padrão todo a cada projeto começou a ser inviável.

O Struts foi um dos primeiros frameworks MVC com a ideia de se criar um controlador reutilizável entre projetos. Ele foi lançado no ano 2000 com o objetivo de tornar mais simples a criação de aplicações Web com a linguagem Java ao disponibilizar uma série de funcionalidades já prontas.

Isso fez com que muitas pessoas o utilizassem para desenvolver suas aplicações, tornando-o rapidamente a principal solução MVC no mercado Java. Uma das consequências disso é que hoje em dia ele é um dos mais utilizados no mercado.

No entanto, hoje, ele é visto como um framework que demanda muito trabalho, justamente por ter sido criado há muito tempo, quando muitas das facilidades da linguagem Java ainda não existiam.

Por isso surgiram outros frameworks MVC. A comunidade do Struts, por exemplo, uniu forças com a de outro framework que começava a ganhar espaço no mercado, que era o WebWork. Ambas as comunidades se fundiram e desenvolveram o **Struts 2**, que vem a ser uma versão mais simples de se trabalhar do que o Struts 1, e com ainda mais recursos e funcionalidades.

Um dos frameworks mais famosos no mercado é o Spring MVC. Spring é um framework que inicialmente não foi criado para o desenvolvimento web. Na essência o Spring é um container leve que visa fornecer serviços para sua aplicação como por exemplo o gerenciamento de objetos ou transação. Mas com o tempo a comunidade Spring entendeu que o Struts era ultrapassado e começou criar um framework MVC próprio. O Spring MVC é um framework moderno que usa os recursos atuais da linguagem além de usar todo poder do container Spring. Nesse capítulo veremos as funcionalidades desse framework poderoso.

STRUTS 1

Embora bastante antigo, o Struts 1 ainda é usado em muitas empresas. Os conceitos e fundamentos são muito parecidos entre as versões, mas a versão antiga é mais trabalhosa e possui formas particulares de uso.

STRUTS 2

Apesar do nome famoso, o Struts 2 nunca foi tão utilizado quanto o Struts 1. Enquanto o Struts 1 era pioneiro na época e se tornou o padrão no desenvolvimento web Java, o Struts 2 era uma escolha entre vários frameworks MVC que ofereciam as mesmas facilidades.

11.3 CONFIGURANDO O SPRING MVC

Para que possamos aprender o Spring MVC, vamos criar um sistema de *lista de tarefas*. E o primeiro passo que precisamos dar é ter o Spring MVC para adicionarmos em nossa aplicação. Spring MVC vem junto com as bibliotecas do framework Spring que podemos encontrar no site <http://springsource.org>. Lá, é possível encontrar diversas documentações e tutoriais, além dos JARs do projeto.

Uma vez que adicionamos os JARs do Spring MVC em nosso projeto dentro do diretório `WEB-INF/lib`, precisamos declarar um Servlet, que fará o papel de *Front Controller* da nossa aplicação, recebendo as requisições e as enviando às lógicas corretas. Para declararmos a Servlet do Spring MVC, basta adicionarmos no `web.xml` da nossa aplicação:

```
<servlet>
    <servlet-name>Spring MVC Dispatcher Servlet</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>Spring MVC Dispatcher Servlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Repare que é uma configuração normal de Servlet, com `servlet-class` e `url-pattern`, como as que fizemos antes. Tem apenas um elemento novo, o `init-param`. Este parâmetro é uma configuração que pode ser passada para o servlet pelo `web.xml`. Aqui definimos o nome do arquivo de configuração do framework Spring, o `spring-context.xml`. Quando o Servlet é carregado, ele vai procurar esse `spring-context.xml` dentro da pasta `WEB-INF`.

XML específico do Spring

O framework Spring possui sua própria configuração XML. O Spring, por ser muito mais do que um

controlador MVC, poderia ser utilizado em ambientes não Web, ou seja nem sempre o Spring pode se basear no `web.xml`. Por este motivo, mas não somente este, o Spring definiu o seu próprio XML com várias opções para configurar a aplicação.

A primeira coisa que faremos nesse arquivo é habilitar o uso de anotações do Spring MVC e configurar pacote base da aplicação web para o Spring achar as nossas classes:

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

Além disso, é preciso informar ao Spring o local onde colocaremos os arquivos JSP. Para isso Spring MVC oferece uma classe especial que recebe o nome da pasta dos JSPs e a extensão dos arquivos. Vamos criar todos os JSPs na pasta `/WEB-INF/views/`:

```
<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 começar com o Spring MVC. O arquivo completo, com todos os cabeçalhos, fica então como:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="br.com.caelum.tarefas" />
    <mvc:annotation-driven />

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

</beans>
```

11.4 CRIANDO AS LÓGICAS

No nosso framework MVC que desenvolvemos anteriormente, para criarmos nossas lógicas, criávamos uma classe que implementava uma interface. Existem diversas abordagens seguidas pelos frameworks para possibilitar a criação de lógicas. Passando por declaração em um arquivo XML, herdando de alguma classe do framework e, a partir do Java 5, surgiram as anotações, que são uma forma de introduzir metadados dentro de nossas classes.

O Spring MVC permite criar as lógicas de formas diferentes, usando convenções ou declarar tudo no XML, porém na versão 3 a maneira indicada é utilizando anotações.

11.5 A LÓGICA OLÁ MUNDO!

Para criarmos nossa primeira lógica, vamos criar uma classe chamada `olaMundoController`. Nela vamos colocar métodos que são as ações (`Action`). O sufixo `Controller` não é obrigatório para o Spring MVC porém é uma convenção do mercado. Como utilizaremos Spring MVC baseado em anotações, é **obrigatório** que o seu `Controller` esteja dentro do pacote `br.com.caelum.tarefas` ou em um subpacote. No nosso caso, criaremos a classe dentro do pacote: `br.com.caelum.tarefas.controller`.

Dentro dessa nossa nova classe, vamos criar um método que imprimirá algo no console e, em seguida, irá redirecionar para um JSP com a mensagem "Olá mundo!". A classe deve ser anotada com `@Controller`, uma anotação do Spring MVC. Ela indica ao Spring que os métodos dessa classe são ações(`Action`). Podemos criar um método de qualquer nome dentro dessa classe, desde que ele esteja com a anotação `@RequestMapping`. A anotação `@RequestMapping` recebe um atributo chamado `value` que indica qual será a URL utilizada para invocar o método, como esse atributo já é o padrão não precisamos definir. Portanto, se colocarmos o valor `olaMundoSpring` acessaremos o método dentro do nosso `@Controller` pela URL <http://localhost:8080/fj21-tarefas/olaMundoSpring>.

Vamos chamar o método `execute`, mas novamente poderia ser qualquer nome. Esse método deve retornar uma `String` que indica qual JSP deve ser executado após a lógica. Por exemplo, podemos retornar "ok" para enviar o usuário para uma página chamada `ok.jsp`. O método não deve retornar o sufixo da página, já que isso foi configurado no XML do Spring. Também lembrando que o Spring MVC procura as páginas JSP dentro da pasta `WEB-INF/views`.

Dessa forma, teríamos a seguinte classe:

```
@Controller
public class OlaMundoController {

    @RequestMapping("/olaMundoSpring")
    public String execute() {
        System.out.println("Executando a lógica com Spring MVC");
        return "ok";
    }
}
```

Um ponto importante a se notar é que podemos criar outros métodos que respondam por outras URL's, ou seja, vários ações dentro dessa classe (dentro do mesmo `@Controller`). Bastaria que nós utilizássemos novamente a anotação `@RequestMapping` esses métodos.

Por fim, só precisamos criar o JSP que mostrará a mensagem "Olá mundo!". Basta criar o arquivo `ok.jsp` dentro da pasta `WEB-INF/views/`, que mapeamos anteriormente no XML do Spring.

O JSP terá o seguinte conteúdo:

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

Podemos acessar nosso método pela URL <http://localhost:8080/fj21-tarefas/olaMundoSpring>. O que acontece é que após a execução do método o Spring MVC verifica qual foi o resultado retornado pelo seu método e procura despachar a requisição para a página indicada.

11.6 PARA SABER MAIS: CONFIGURANDO O SPRING MVC EM CASA

Caso você esteja em casa, faça o download do Spring Framework e use apenas os seguintes JARs na sua aplicação:

- commons-logging-1.x.x.jar
- log4j-1.2.x.jar
- mysql-connector-java-5.x.x.jar
- slf4j-api-1.7.x.jar
- slf4j-log4j12-1.7.x.jar
- spring-aspects-4.x.x.RELEASE.jar
- spring-aop-4.x.x.RELEASE.jar
- spring-beans-4.x.x.RELEASE.jar
- spring-context-4.x.x.RELEASE.jar
- spring-core-4.x.x.RELEASE.jar
- spring-expression-4.x.x.RELEASE.jar
- spring-jdbc-4.x.x.RELEASE
- spring-web-4.x.x.jar
- spring-webmvc-4.x.x.RELEASE.jar

11.7 EXERCÍCIOS: CONFIGURANDO O SPRING MVC E TESTANDO A CONFIGURAÇÃO

1. Vamos configurar o Spring MVC em um novo projeto.
 - Crie um novo projeto web: **File -> New -> Project... -> Dynamic Web Project** chamado **fj21-tarefas** .
 - Na aba **Servers**, clique com o botão direito no Tomcat e vá em **Add and Remove...**:
 - Basta selecionar o nosso projeto **fj21-tarefas** e clicar em **Add**:

- Vamos começar importando as classes que serão necessárias ao nosso projeto, como o modelo de Tarefas e o DAO .
- Clique com o botão direito no projeto `fj21-tarefas` e escolha a opção `Import` .
- Selecione General -> Archive File
- Escolha o arquivo `projeto-tarefas.zip` que está em `Desktop/21` e confirme a importação.
- Abra o arquivo `web.xml` para fazermos a declaração do servlet do Spring MVC:

```

<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>

```

2. Vamos fazer um simples Olá Mundo, para testarmos nossa configuração:

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

```

@Controller
public class OlaMundoController {

    @RequestMapping("/olaMundoSpring")
    public String execute() {
        System.out.println("Executando a lógica com Spring MVC");
        return "ok";
    }
}

```

Dica: Use `Ctrl+Shift+O` para importar as classes.

- Precisamos preparar a camada de visualização. 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 Spring MVC!</h2>
</body>
</html>

```

- Reinic peace o Tomcat e acesse no seu navegador o endereço <http://localhost:8080/fj21-tarefas/olaMundoSpring>. O resultado deve ser algo parecido com:



11.8 ADICIONANDO TAREFAS E PASSANDO PARÂMETROS

Vamos começar a criar o nosso sistema de tarefas. Guardaremos uma descrição da tarefa, e uma indicação informando se a tarefa já foi finalizada ou não e quando foi finalizada. Esse sistema é composto pelo seguinte modelo:

```

public class Tarefa {
    private Long id;
    private String descricao;
    private boolean finalizado;
    private Calendar dataFinalizacao;

    //getters e setters
}

```

Vamos criar a funcionalidade de adição de novas tarefas. Para isso, teremos uma tela contendo um formulário com campos para serem preenchidos. Queremos que, ao criarmos uma nova tarefa, a mesma venha por padrão como não finalizada e, consequentemente, sem a data de finalização definida. Dessa forma, nosso formulário terá apenas o campo `descricao`. Podemos criar um JSP chamado `formulario.jsp` contendo somente o campo para descrição:

```

<html>
    <body>
        <h3>Adicionar tarefas</h3>
        <form action="adicionaTarefa" method="post">
            Descrição: <br />

```

```

<textarea name="descricao" rows="5" cols="100"></textarea><br />

<input type="submit" value="Adicionar">
</form>
</body>
</html>

```

O nosso form, ao ser submetido, chama uma ação, um método dentro de um `@Controller` que responde pela URL `adicionaTarefa`. Esse método precisa receber os dados da requisição e gravar a tarefa que o usuário informou na tela. Vamos chamar esse método `adiciona` e colocar dentro da classe `TarefasController`:

```

@Controller
public class TarefasController {

    @RequestMapping("adicionaTarefa")
    public String adiciona() {
        JdbcTarefaDao dao = new JdbcTarefaDao();
        dao.adiciona(tarefa);
        return "tarefa-adicionada";
    }
}

```

Mas, como montaremos o objeto `tarefa` para passarmos ao nosso DAO? Dentro dessa nossa classe `TarefasController` em nenhum momento temos um `HttpServletRequest` para pegarmos os parâmetros enviados na requisição e montarmos o objeto `tarefa`.

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. E com o Spring MVC não é diferente.

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 com os dados que vieram da requisição. Portanto, vamos criar no nosso método um novo parâmetro chamado `tarefa`:

```

@Controller
public class TarefasController {

    @RequestMapping("adicionaTarefa")
    public String adiciona(Tarefa tarefa) {
        JdbcTarefaDao dao = new JdbcTarefaDao();
        dao.adiciona(tarefa);
        return "tarefa-adicionada";
    }
}

```

Queremos que o campo de texto que criamos no nosso formulário preencha a descrição dessa tarefa. Para fazermos isso, basta darmos o nome com o caminho da propriedade que queremos definir. Portanto, se dentro do objeto `tarefa` queremos definir a propriedade `descricao`, basta nomearmos o `input` com `descricao`. O Spring MVC cria o objeto `tarefa` para nós e preenche esse objeto com os dados da requisição, nesse caso com o parâmetro `descricao` da requisição. Como a classe `Tarefa` é um JavaBean e possui construtor sem argumentos e getters/setters isso não será problema.

Por fim, basta exibirmos a mensagem de confirmação de que a criação da tarefa foi feita com sucesso. Criamos o arquivo `tarefa-adicionada.jsp` com o seguinte conteúdo HTML:

```
<html>
<body>
    Nova tarefa adicionada com sucesso!
</body>
</html>
```

Melhorar a organização dos arquivos JSPs

A nossa aplicação terá várias páginas relacionadas com uma ou mais tarefas. Queremos organizar a nossa aplicação desde início e separar os arquivos JSPs relacionadas em subpastas. Ou seja, todas as páginas JSP relacionadas com o modelo tarefa ficarão na pasta `tarefa`. Por isso, vamos criar uma nova pasta `tarefa` dentro da pasta `WEB-INF/views` para os JSPs que o `TarefasController` vai usar. 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 `tarefa/adicionada`.

Por tanto, o código completo do método `adiciona` fica:

```
@Controller
public class TarefasController {

    @RequestMapping("adicionaTarefa")
    public String adiciona(Tarefa tarefa) {
        JdbcTarefaDao dao = new JdbcTarefaDao();
        dao.adiciona(tarefa);
        return "tarefa/adicionada";
    }
}
```

Na pasta `WEB-INF/views/tarefa` também deve ficar o formulário para adicionar uma tarefa. Como discutimos antes, todos os JSPs da tarefa na mesma subpasta. Porém aqui surge um novo problema: É preciso carregar o JSP no navegador, mas o acesso direto ao pasta `WEB-INF` é proibido pelo servlet-container e consequentemente não é possível acessar o formulário. Para resolver isso vamos criar uma nova ação (um novo método) dentro da classe `TarefasController` que tem a finalidade de chamar o formulário apenas. O método usa também a anotação `@RequestMappping` e retorna um `String` para chamar o formulário.

Abaixo o código completo do `TarefasController` com os dois métodos:

```
@Controller
public class TarefasController {

    @RequestMapping("novaTarefa")
    public String form() {
        return "tarefa/formulario";
    }

    @RequestMapping("adicionaTarefa")
```

```

public String adiciona(Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.adiciona(tarefa);
    return "tarefa/adicionada";
}
}

```

A estrutura das pasta `WEB-INF` fica como:

```

WEB-INF
-views
 -tarefa
   -formulario.jsp
   -adicionada.jsp

```

Para chamar o formulário usaremos a URL: <http://localhost:8080/fj21-tarefas/novaTarefa>

11.9 EXERCÍCIOS: CRIANDO TAREFAS

Vamos criar o formulário e nossa ação para fazer a gravação das tarefas.

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

```

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

```

- Agora precisamos um método (action) dentro de um `@Controller` para acessar o JSP. Crie uma nova classe no pacote `br.com.caelum.tarefas.controller` chamada `TarefasController`.

Nossa classe precisa ter um método para acessar o JSP. Vamos chamar o método `form()` e usar a anotação `@RequestMapping`:

```

@Controller
public class TarefasController {

    @RequestMapping("novaTarefa")
    public String form() {
        return "tarefa/formulario";
    }
}

```

Use `Ctrl+Shift+O` para importar as classes.

- Ainda falta o método que realmente adiciona a tarefa no banco de dados. Esse método é chamado

pelo nosso formulário e recebe uma tarefa como parâmetro. Ele novamente usa a anotação `@RequestMapping` para definir a URL.

Dentro da classe `TarefasController` crie o método `adiciona` que recebe uma tarefa. No método usamos a `JdbcTarefaDao` para persistir os dados. O retorno do método define o local e nome do JSP.

O código deve ficar:

```
@RequestMapping("adicionaTarefa")
public String adiciona(Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.adiciona(tarefa);
    return "tarefa/adicionada";
}
```

4. E, por fim, criamos o arquivo **adicionada.jsp** na pasta **tarefa** que mostrará uma mensagem de confirmação de que a tarefa foi efetivamente adicionada.

```
<html>
<body>
    Nova tarefa adicionada com sucesso!
</body>
</html>
```

5. Reinicie o Tomcat.

Acesse no seu navegador o endereço `http://localhost:8080/fj21-tarefas/novaTarefa` e adicione uma nova tarefa.





Caso aconteça uma exceção informando que a tabela não está criada, crie-a com o script abaixo e tente inserir novamente a tarefa. Abra o terminal e digite:

```
mysql -u root
use fj21;
create table tarefas (
    id BIGINT NOT NULL AUTO_INCREMENT,
    descricao VARCHAR(255),
    finalizado BOOLEAN,
    dataFinalizacao DATE,
    primary key (id)
);
```

11.10 INCLUINDO VALIDAÇÃO NO CADASTRO DE TAREFAS

Já conseguimos adicionar novas tarefas em nossa aplicação. Porém, o que impede algum usuário desatento incluir uma tarefa sem descrição? Até agora, nada. Nós queremos que um usuário não seja capaz de adicionar uma tarefa sem descrição, para isso precisamos incluir algum mecanismo de *validação* em nossa ação de adicionar tarefas. A validação deve ser executada no lado do servidor, afim de garantir que os dados serão validados em um lugar onde o usuário não consiga interferir.

Validando programaticamente

A maneira mais fácil de validar a tarefa é usar vários `if`s no método `adiciona` da classe `TarefasController` antes de chamar `dao.adiciona(tarefa)`, executando a validação programaticamente. O código seguinte mostra a ideia:

```
@RequestMapping("adicionaTarefa")
public String adiciona(Tarefa tarefa) {
    if(tarefa.getDescricao() == null || tarefa.getDescricao().equals(""))
        return "tarefa/formulario";
```

```

    }

JdbcTarefaDao dao = new JdbcTarefaDao();
dao.adiciona(tarefa);
return "tarefa/adicionada";
}

```

O problema aqui é quanto mais atributos na tarefa mais `if`s teremos. É provável também que vamos repetir um `if` ou outro quando validarmos a tarefa em métodos diferentes, por exemplo, para adicionar ou alterar a tarefa. Sabemos que copiar e colar código não é uma boa maneira de reaproveitar código. O que precisamos é de algum artifício que seja igual para qualquer método, algo que ajude na validação dos dados.

11.11 VALIDAÇÃO COM BEAN VALIDATION

A partir do Java EE 6 temos uma especificação que resolve este problema. A JSR 303, também conhecida como Bean Validation, define uma série de anotações e uma API para criação de validações para serem utilizadas em Java Beans, que podem ser validados agora em qualquer camada da aplicação.

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

```

public class Tarefa {

    private Long id;

    @Size(min=5)
    private String descricao;

    private boolean finalizado;
    private Calendar dataFinalizacao;

    //...
}

```

Pronto! Com essas anotações, qualquer objeto do tipo `Tarefa` pode ser validado na camada de controller. Só falta avisar o Spring MVC que realmente queremos executar a validação. Isso é feito pela anotação `Valid` que devemos usar na antes do parâmetro da ação:

```

@RequestMapping("adicionaTarefa")
public String adiciona(@Valid Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.adiciona(tarefa);
    return "tarefa/adicionada";
}

```

Como estamos falando de Spring MVC, antes da chamada do método é executada a validação, ou seja será verificado se a descrição da tarefa não está vazia. Se estiver, será lançada uma exceção do tipo `ConstraintViolationException` que possui a descrição do erro.

Não queremos mostrar uma exceção para o usuário e sim apenas voltar para o formulário para

mostrar uma mensagem que a validação falhou. O Spring MVC pode guardar o resultado (os erros de validação) em um objeto do tipo `BindingResult`. Assim não será lançado um exceção. Este objeto `BindingResult` se torna um parâmetro da ação. Então só é preciso perguntar para ele se existe um erro de validação e se existir, voltar para o formulário. Veja o código:

```
@RequestMapping("adicionaTarefa")
public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

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

    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.adiciona(tarefa);
    return "tarefa/adicionada";
}
```

No código acima verificamos se existe um de erro validação relacionado com o atributo `descricao` da tarefa. Também podemos conferir se existe algum erro de validação, mais genérico:

```
@RequestMapping("adicionaTarefa")
public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

    if(result.hasErrors()) {
        return "tarefa/formulario";
    }
}
```

Mostrando as mensagens de validação

Para exibir as mensagens de validação no JSP usamos um tag especial que o Spring MVC oferece. O tag se chama **form:errors**:

```
<form:errors path="tarefa.descricao" />
```

O atributo `path` indica com que atributo essa mensagem está relacionada.

Abaixo está o código completo do formulário `formulario.jsp` da pasta `tarefa`. Repare que é preciso importar o taglib do Spring MVC:

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<html>
<body>
    <h3>Adicionar tarefas</h3>
    <form action="adicionaTarefa" method="post">
        Descrição:
        <br/>
        <textarea rows="5" cols="100" name="descricao"></textarea>
        <br/>
        <form:errors path="tarefa.descricao" cssStyle="color:red"/>
        <br/>
        <input type="submit" value="Adicionar"/>
    </form>
</body>
</html>
```

Mensagens internacionalizadas

Para deixar as mensagens de nossa aplicação mais fáceis de alterar, é comum criar um arquivo separado do HTML que possui apenas mensagens. Este arquivo é normalmente chamado *mensagens.properties* ou *messages.properties*.

Na pasta *WEB-INF* do projeto podemos então criar este arquivo com o seguinte conteúdo:

```
tarefa.adicionada.com.sucesso=Tarefa adicionada com sucesso!
```

```
...
```

Repare que definimos as mensagens em um estilo de: <chave>=<valor> .

O Spring MVC pode carregar automaticamente este arquivo, desde que a linha abaixo seja incluída no arquivo *spring-context.xml*:

```
<bean id="messageSource" class=
    "org.springframework.context.support
     .ReloadableResourceBundleMessageSource">
    <property name="basename" value="/WEB-INF/mensagens" />
</bean>
```

Basta então usar a taglib *fmt* para mostra mensagens do arquivo *mensagens.properties* na página HTML:

```
<fmt:message key="tarefa.adicionada.com.sucesso"/>
```

Personalizando as mensagens de erros

Podemos ainda personalizar as mensagens de validação do Bean Validation, escrevendo nossas mensagens dentro do atributo *message* das anotações:

```
public class Tarefa {

    private Long id;

    @Size(min=5, message="Descrição deve ter pelo menos 5 caracteres")
    private String descricao;
    ...
}
```

Para não deixar as mensagens de validação espalhadas em nossas classes, podemos isolar estas mensagens no arquivo padrão de mensagens do Bean Validation, chamado **ValidationMessages.properties** :

```
tarefa.descricao.pequena=Descrição deve conter
    pelo menos {min} caracteres
    ...

```

O arquivo deve ficar dentro da pasta **src**. Depois podemos referenciar as chaves das mensagens dentro das anotações também pelo atributo *message* :

```
public class Tarefa {

    private Long id;
```

```

    @Size(min=5, message="{tarefa.descricao.pequena}")
    private String descricao;
}

```

11.12 EXERCÍCIOS: VALIDANDO TAREFAS

- Para configurar o framework *Bean Validation* é preciso copiar quatro jars.
 - Primeiro, vá ao Desktop, e entre no diretório `21/jars-hibernate-validator`.
 - Haverá quatro JARs: `classmate-x.x.x.jar`, `hibernate-validator-x.x.x.Final.jar`, `jboss-logging-x.x.x.GA.jar` e `validation-api-x.x.x.Final.jar`
 - Copie-os (CTRL+C) e cole-os (CTRL+V) dentro do *workspace* do eclipse na pasta `fj21-tarefas/WebContent/WEB-INF/lib`
- Abra a classe `Tarefa`. Nela é preciso definir as regras de validação através das anotações do framework *Bean validation*. A atributo **descricao** deve ter pelo menos 5 caracteres:

```

public class Tarefa {

    private Long id;

    @Size(min=5)
    private String descricao;
}

```

- Abra a classe `TarefasController` e procure o método `adiciona`. Coloque a anotação `@Valid` na frente do parâmetro `Tarefa tarefa` e adicione o parâmetro `BindingResult` na assinatura do método.

Além disso, no mesmo método, adicione a verificação se há erros de validação. O método completo fica:

```

@RequestMapping("adicionaTarefa")
public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

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

    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.adiciona(tarefa);
    return "tarefa/adicionada";
}

```

- Abra o JSP `formulario.jsp` (da pasta `WEB-INF/views/tarefa`). Adicione no início da página a declaração da taglib do Spring MVC:

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

Dentro do HTML adicione a tag `form:errors` acima do tag `form`. Adicione apenas o tag `form:errors`:

```
<form:errors path="tarefa.descricao"/>
```

```
<form action="adicionaTarefa" method="post">
```

5. Reinicie o Tomcat e acesse no seu navegador o endereço `http://localhost:8080/fj21-tarefas/novaTarefa`. Envie uma requisição SEM preencher a descrição da tarefa, a mensagem de validação deve aparecer.



11.13 LISTANDO AS TAREFAS E DISPONIBILIZANDO OBJETOS PARA A VIEW

Como já conseguimos adicionar tarefas em nossa aplicação, precisamos saber o que foi adicionado. Para isso precisamos criar uma nova funcionalidade que lista as tarefas. A função dessa ação será invocar o `JdbcTarefaDao` para conseguir a lista das tarefas que estão no banco de dados. Podemos adicionar um novo método dentro da classe `TarefasController`:

```
@RequestMapping("listaTarefas")
public String lista() {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    List<Tarefa> tarefas = dao.lista();
    return "tarefa/lista";
}
```

Essa lista de tarefas deverá ser disponibilizada para o JSP fazer sua exibição. Para que possamos disponibilizar um objeto para o JSP, temos que alterar o retorno do método `lista`. A ideia é que o Spring MVC não só recebe o nome da página JSP (`tarefa/lista`) quando chama o método `lista`, o Spring MVC também recebe os dados para o JSP. Os dados para a exibição na tela e o nome da página JSP foram encapsulados pelo Spring MVC em uma classe especial que se chama `ModelAndView`. Vamos criar um objeto do tipo `ModelAndView` e preencher esse modelo com nossa lista de tarefas e definir o nome da página JSP. O método `lista` deve retornar esse objeto, não mais apenas um `String`. Veja como fica o código:

```

@RequestMapping("listaTarefas")
public ModelAndView lista() {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    List<Tarefa> tarefas = dao.lista();

    ModelAndView mv = new ModelAndView("tarefa/lista");
    mv.addObject("tarefas", tarefas);
    return mv;
}

```

Dessa forma, será disponibilizado para o JSP um objeto chamado `tarefas` que pode ser acessado via *Expression Language* como `${tarefas}`. Poderíamos em seguida iterar sobre essa lista utilizando a Tag `forEach` da JSTL core.

Vendo o código da ação `lista` pode aparecer estranho instanciar uma classe do framework Spring (`ModelAndView`) dentro do nosso controlador. Isso até influencia negativamente a testabilidade do método. Por isso, O Spring MVC dá uma outra opção, oferece uma alternativa ao uso da classe `ModelAndView`. Na nossa ação podemos receber um objeto que representa o modelo para o nosso JSP. Spring MVC pode passar um parâmetro para o método do controlador que tem a função do modelo. Esse modelo podemos preencher com a lista de tarefas. Assim também continuaremos devolver uma String como retorno do método que indica o caminho para o JSP:

```

@RequestMapping("listaTarefas")
public String lista(Model model) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    List<Tarefa> tarefas = dao.lista();
    model.addAttribute("tarefas", tarefas);
    return "tarefa/lista";
}

```

Dessa maneira não é preciso instanciar o modelo, e sim apenas disponibilizar a lista.

11.14 EXERCÍCIOS: LISTANDO TAREFAS

1. Vamos criar a listagem das nossas tarefas mostrando se a mesma já foi finalizada ou não.
 - Na classe `TarefasController` adicione o método `lista` que recebe um `Model` como parâmetro:


```

@RequestMapping("listaTarefas")
public String lista(Model model) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    model.addAttribute("tarefas", dao.lista());
    return "tarefa/lista";
}

```
 - Para fazer a listagem, vamos precisar da JSTL (iremos fazer um `forEach`), portanto precisamos importá-la. Primeiro, vá ao Desktop e entre no diretório `21/jars-jstl`.
 - Haverão dois JARs, `javax.servlet.jsp.jstl-x.x.x.jar` e `javax.servlet.jsp.jstl-api-x.x.x.jar`.
 - Copie-os (CTRL+C) e cole-os (CTRL+V) dentro do *workspace* do eclipse na pasta `fj21-`

- No Eclipse, dê um F5 no seu projeto. Pronto, a JSTL já está em nosso projeto.
- Crie o JSP que fará a exibição das tarefas dentro da pasta WebContent/WEB-INF/views/tarefa . Chame-o de lista.jsp e adicione o seguinte conteúdo:

```

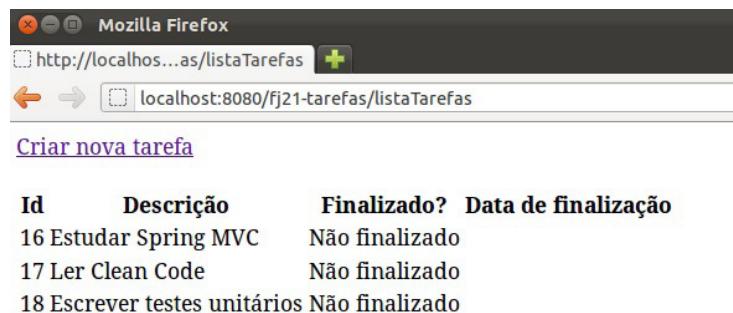
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<html>
<body>

    <a href="novaTarefa">Criar nova tarefa</a>

    <br /> <br />

    <table>
        <tr>
            <th>Id</th>
            <th>Descrição</th>
            <th>Finalizado?</th>
            <th>Data de finalização</th>
        </tr>
        <c:forEach items="${tarefas}" var="tarefa">
            <tr>
                <td>${tarefa.id}</td>
                <td>${tarefa.descricao}</td>
                <c:if test="${tarefa.finalizado eq false}">
                    <td>Não finalizado</td>
                </c:if>
                <c:if test="${tarefa.finalizado eq true}">
                    <td>Finalizado</td>
                </c:if>
                <td>
                    <fmt:formatDate
                        value="${tarefa.dataFinalizacao.time}"
                        pattern="dd/MM/yyyy"/>
                </td>
            </tr>
        </c:forEach>
    </table>
</body>
</html>
```

- Reinicie o Tomcat e acesse o endereço <http://localhost:8080/fj21-tarefas/listaTarefas> e veja o resultado.



Id	Descrição	Finalizado?	Data de finalização
16	Estudar Spring MVC	Não finalizado	
17	Ler Clean Code	Não finalizado	
18	Escrever testes unitários	Não finalizado	

11.15 REDIRECIONANDO A REQUISIÇÃO PARA OUTRA AÇÃO

Tarefas podem ser adicionadas por engano, ou pode ser que não precisemos mais dela, portanto, queremos removê-las. Para fazer essa remoção, criaremos um link na listagem que acabamos de desenvolver que, ao ser clicado, invocará a nova ação para a remoção de tarefas passando o código da tarefa para ser removida.

O link pode ser feito com HTML na lista de tarefas da seguinte forma:

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

Podemos desenvolver um método para fazer a remoção. A lógica não possui nenhuma novidade, basta recuperarmos o parâmetro como aprendemos nesse capítulo e invocarmos o DAO para fazer a remoção:

```
@RequestMapping("removeTarefa")
public String remove(Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.remove(tarefa);
    return "para onde ir??";
}
```

A questão é: Para qual lugar redirecionar o usuário após a exclusão?

Poderíamos criar um novo JSP com uma mensagem de confirmação da remoção, mas usualmente isso não costuma ser bom, porque precisaríamos navegar até a lista das tarefas novamente caso tenhamos que remover outra tarefa. Seria muito mais agradável para o usuário que ele fosse redirecionado direto para a lista das tarefas.

Uma das formas que poderíamos fazer esse redirecionamento é enviar o usuário diretamente para a página que lista as tarefas (`tarefa/lista.jsp`). Mas, essa não é uma boa abordagem, porque precisaríamos, outra vez, disponibilizar a lista das tarefas para o JSP, algo que já fazemos na ação de listar as tarefas, o método `lista` na classe `TarefasController`.

Já que o método `lista` faz esse trabalho, poderíamos, ao invés de redirecionar a execução para o JSP, enviá-la para essa ação. Para isso, o retorno do método deve ser um pouco modificado. Vamos continuar devolvendo uma `String` mas essa `String` deve indicar que queremos chamar uma outra ação. Podemos fazer um redirecionamento na lado do servidor (*forward*) ou pelo navegador, no lado do cliente (*redirect*).

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

```
@RequestMapping("removeTarefa")
public String remove(Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.remove(tarefa);
    return "forward:listaTarefas";
}
```

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

```
@RequestMapping("removeTarefa")
public String remove(Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.remove(tarefa);
    return "redirect:listaTarefas";
}
```

11.16 EXERCÍCIOS: REMOVENDO E ALTERANDO TAREFAS

1. Vamos fazer a funcionalidade de remoção de tarefas.

- Adicione no arquivo `lista.jsp` uma coluna com um link que ao ser clicado invocará a Action para remover tarefa.

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

- Crie um novo método **remove** na classe `TarefasController` com o código:

```
@RequestMapping("removeTarefa")
public String remove(Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.remove(tarefa);
    return "redirect:listaTarefas";
}
```

- Acesse a lista de tarefas em `http://localhost:8080/fj21-tarefas/listaTarefas` e remova algumas tarefas.

2. Criaremos a tela para fazer a alteração das tarefas, como por exemplo, marcá-la como finalizada e definirmos a data de finalização.

- Primeiro vamos criar um novo link na nossa listagem que enviará o usuário para a tela contendo os dados da tarefa selecionada:

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

- Vamos criar uma nova ação que dado um `id`, devolverá a `Tarefa` correspondente para um JSP, que mostrará os dados para que a alteração possa ser feita.

Crie um novo método `mostra` na classe `TarefasController`:

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

- Crie o JSP `mostra.jsp` dentro da pasta `views/tarefa` para mostrar a tarefa escolhida:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

```

<html>
<body>
    <h3>Alterar tarefa - ${tarefa.id}</h3>
    <form action="alteraTarefa" method="post">

        <input type="hidden" name="id" value="${tarefa.id}" />

        Descrição:<br />
        <textarea name="descricao" cols="100" rows="5"><%-
            -%>${tarefa.descricao}<%-
            -%></textarea>
        <br />

        Finalizado? <input type="checkbox" name="finalizado"
            value="true" ${tarefa.finalizado? 'checked' : '' }/> <br />

        Data de finalização: <br />
        <input type="text" name="dataFinalizacao"
            value=<fmt:formatDate
                value="${tarefa.dataFinalizacao.time}"
                pattern="dd/MM/yyyy" />"/>
        <br />

        <input type="submit" value="Alterar"/>
    </form>
</body>
</html>

```

- Para o Spring MVC saber converter automaticamente a data no formato brasileiro para um Calendar é preciso usar a anotação `@DateTimeFormat`. Abra a classe `Tarefa` e adicione a anotação acima do atributo `dataFinalizacao`:

```

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

```

- Falta criar um método que cuidará da alteração da tarefa. Na classe `TarefasController` adicione esse método:

```

@RequestMapping("alteraTarefa")
public String altera(Tarefa tarefa) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.altera(tarefa);
    return "redirect:listaTarefas";
}

```

- Acesse a lista de tarefas em <http://localhost:8080/fj21-tarefas/listaTarefas> e altere algumas tarefas.

11.17 DESAFIO - CALENDÁRIO

- Adicione o campo com calendário que fizemos no capítulo de criação de Tags em nosso projeto e utilize-o no formulário de alteração.

11.18 MELHORANDO A USABILIDADE DA NOSSA APLICAÇÃO

Sempre que precisamos finalizar uma tarefa, precisamos entrar na tela de alteração da tarefa que

queremos e escolher a data de finalização. Essa data de finalização na maioria das vezes é a própria data atual.

Para facilitar a usabilidade para o usuário, vamos adicionar um novo link na nossa tabela que se chamará "*Finalizar agora*". Ao clicar nesse link, uma ação será invocada para finalizarmos a tarefa no dia atual.

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*.

AJAX nada mais é do que uma técnica que nos permite enviar requisições assíncronas, ou seja, manter a página que estava aberta intacta, e recuperar a resposta dessa requisição para fazermos qualquer processamento com eles. Essas respostas costumam ser XML, HTML ou um formato de transferência de dados chamado JSON (*Javascript Object Notation*).

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). O jQuery nos fornece duas funções: `$.post` e `$.get`, cada uma para cada método.

Para as funções basta passarmos o endereço que queremos invocar, como por exemplo:

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

Nesse caso, estamos enviando uma requisição via GET para o endereço `minhaPagina.jsp`.

Sabendo isso, vamos criar um link que invocará uma função Javascript e fará requisição AJAX para uma ação que finalizará a tarefa:

```
<td><a href="#" onclick="finalizaAgora(${tarefa.id})">
    Finalizar agora
</a></td>
```

Vamos criar a função `finalizaAgora` que recebe o id da tarefa que será finalizada e a passará como parâmetro para a ação:

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

Por fim, basta criarmos a nossa ação que receberá o parâmetro e invocará um método do

`JdbcTarefaDao` para fazer a finalização da tarefa. 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, ou seja, se foi executado com sucesso, se a página não foi encontrada, se algum erro aconteceu e assim por diante.

O protocolo HTTP define que o código 200 indica que a execução ocorreu com sucesso, portanto, vamos apenas indicar na nossa resposta o código, sem devolver nada no corpo da nossa resposta. Para setar o código da resposta programaticamente precisamos do objeto `HttpServletResponse`. Podemos receber a resposta HTTP como parâmetro de qualquer método que é uma ação. Com a resposta na mão podemos chamar o método `setStatus(200)`.

Dessa forma, poderíamos ter um método na classe `TarefasController` para fazer a finalização da tarefa com o seguinte código:

```
@RequestMapping("finalizaTarefa")
public void finaliza(Long id, HttpServletResponse response) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.finaliza(id);
    response.setStatus(200);
}
```

O primeiro parâmetro é a id da tarefa que vem através da requisição, o segundo é a resposta para setar o código HTTP.

Quando estamos usando um Framework MVC como o Spring, queremos que toda a parte de manipulação de `request` e `response` fique por conta do framework. Da forma que mostramos antes é possível obter o resultado que esperamos, mas manipular o status da resposta é uma tarefa que não queremos fazer diretamente. O Spring nos ajuda nessa tarefa, fazendo com que o retorno do nosso método seja *HTTP 200*, exceto em caso de alguma exception. Para isso, precisamos usar uma annotation chamada `@ResponseBody`. Como o próprio nome da annotation fala, tudo que for retornado, será o corpo da nossa resposta. Ou se nada for retornado, então a resposta será vazia porém o status HTTP será 200. Nossa código pode ficar mais enxuto e sem a manipulação do response:

```
@ResponseBody
@RequestMapping("finalizaTarefa")
public void finaliza(Long id) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.finaliza(id);
}
```

11.19 UTILIZANDO AJAX PARA MARCAR TAREFAS COMO FINALIZADAS

Através do jQuery, podemos enviar uma requisição AJAX para o servidor, e na ação vamos manipular a resposta e setar apenas o status 200. Também seria útil notificar o usuário da aplicação que a

requisição finalizou com sucesso. O jQuery já fornece um jeito muito simples de implementar isso. É preciso adicionar no JavaScript uma função que é chamada quando a requisição termina com sucesso (status 200).

```
<script type="text/javascript">
    function finalizaAgora(id) {
        $.get("finalizaTarefa?id=" + id, function(dadosDeResposta) {
            alert("Tarefa Finalizada!");
        });
    }
</script>
```

Repare que `$.get` recebe mais uma função como parâmetro (também chamado *callback de sucesso*). Nela, definimos apenas um simples `alert` para mostrar uma mensagem ao usuário. Também é possível manipular o HTML da página dinamicamente. O jQuery oferece recursos poderosos para alterar qualquer elemento HTML dentro do navegador.

Por exemplo, podemos selecionar um elemento da página pela `id` e mudar o conteúdo desse elemento:

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

Para o nosso exemplo, então é interessante atualizar a coluna da tarefa para indicar que ela foi finalizada:

```
$("#tarefa_"+id).html("Tarefa finalizada");
```

Leia mais sobre o jQuery na documentação:

<http://api.jquery.com/jQuery.get/> <http://api.jquery.com/id-selector/>

11.20 CONFIGURAR O SPRING MVC PARA ACESSAR ARQUIVOS COMUNS

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/>
```

11.21 EXERCÍCIOS: AJAX

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 sistema.

2. Vamos adicionar AJAX na nossa aplicação. Para isso, utilizaremos o jQuery que precisamos importar para nosso projeto e em nossas páginas.

- Vá ao Desktop, e entre na pasta 21 ;
- Copie o diretório js e cole-o dentro de WebContent/resources no seu projeto f21-tarefas; Caso você esteja em casa, faça o download em <http://jquery.com/download>
- Precisamos importar o jQuery em nossa página de listagem. Para isso, adicione logo após a Tag <html> o seguinte código no arquivo lista.jsp :

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

- Pronto, importamos o jQuery para nossa aplicação.

3. Caso a tarefa não esteja finalizada, queremos que ela possua um novo link que se chamará "Finalizar agora". Ao clicar nele, será chamada via AJAX uma Action que marcará a tarefa como finalizada e a data de hoje será marcada como a data de finalização da mesma.

- Altere a coluna que mostra a tarefa como **não** finalizada no arquivo lista.jsp . Adicione um link que ao ser clicada, chamará uma função Javascript passando o id da tarefa para finalizar. Também adicione uma id para cada elemento <td> .

No arquivo procure o *c:if* para tarefas não finalizadas, altere o elemento td dentro *c:if*:

```
<c:if test="#{tarefa.finalizado eq false}">
    <td id="tarefa_${tarefa.id}">
        <a href="#" onClick="finalizaAgora(${tarefa.id})">
            Finaliza agora!
        </a>
    </td>
</c:if>
```

- Crie a função Javascript *finalizaAgora* para chamar a ação que criaremos a seguir via uma requisição POST:

```
<!-- Começo da página com o import do Javascript -->
<body>
    <script type="text/javascript">
        function finalizaAgora(id) {
            $.post("finalizaTarefa", {"id" : id}, function() {
                // selecionando o elemento html através da
                // ID e alterando o HTML dele
                $("#tarefa_"+id).html("Finalizado");
            });
        }
    </script>

    <a href="novaTarefa">Criar nova tarefa</a>
    <br /> <br />
```

```

<table>
<!-- Resto da página com a tabela -->

```

4. Vamos criar o método para finalizar a tarefa. Após o mesmo ser executado, ele não deverá nos redirecionar para lugar nenhum, apenas indicar que a execução ocorreu com sucesso.

- Abra a classe `TarefasController`. Adicione o método `finaliza` com o conteúdo:

```

@ResponseBody
@RequestMapping("finalizaTarefa")
public void finaliza(Long id) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.finaliza(id);
}

```

- Acesse a listagem <http://localhost:8080/fj21-tarefas/listaTarefas> e clique no novo link para finalizar tarefa. A tela muda sem precisar uma atualização inteira da página.

Id	Descrição	Finalizado?	Data de finalização	
16	Estudar Spring MVC	Finaliza agora!	30/11/2011	Alterar Remover
17	Ler Clean Code	Finalizado	29/11/2011	Alterar Remover

5. (Opcional, Avançado) No mesmo estilo do exercício anterior, use o jQuery para acionar o método `removeTarefa` quando clicado em um botão de "excluir". Para isso, crie uma nova coluna na tabela com um link que o onClick vai chamar o endereço associado a `removeTarefa`, e via AJAX devemos remover a linha da tabela. Pra isso você pode usar um recurso poderoso do jQuery e pedir que seja escondida a linha de onde veio o clique:

```
$(elementoHtml).closest("tr").hide();
```

Dessa maneira você nem precisaria usar `id`s nas `tr`s.

11.22 PARA SABER MAIS: ALTERANDO VALOR DA DATA COM AJAX

Agora ao finalizar nossa tarefa via AJAX o usuário tem um feedback na alteração do HTML de *Não Finalizado* para *Finalizado*. Porém todas as tarefas finalizadas possuem a coluna de data de finalização preenchidas, menos as que acabamos de finalizar.

Para resolver esse problema, de alguma forma o nosso Controller deveria passar uma data para nosso jQuery. Mas como? Uma solução possível seria escrevê-la direto no response. Algo parecido com isso:

```

@RequestMapping("finalizaTarefa")
public void finaliza(Long id, HttpServletResponse response) throws IOException {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.finaliza(id);
    Date dataDeFinalizacao = dao.buscaPorId(id).getDataFinalizacao().getTime();
    String data = new SimpleDateFormat("dd/MM/yyyy").format(dataDeFinalizacao);
    response.getWriter().write(data);
    response.setStatus(200);
}

```

Resolve nosso problema mas ainda assim teríamos que ficar trabalhando direto em um camada mais baixa que são as classes de `HTTPServletRequest` e `HTTPServletResponse`. Agora além de finalizar uma tarefa nossa action tem as responsabilidades de buscar pela data finalizada, formatar a data, escrever no response e mudar o status para `200`. Responsabilidade de mais, não é mesmo?

Retornar uma JSP já nos traz o benefício do status `200`, necessário para nossa função de callback no jQuery. Sabendo disso usaremos uma JSP para renderizar a data formatada. Mas antes é preciso passar essa data a nossa JSP, ou simplesmente passar uma `Tarefa` para ela e depois fazer com que a Action retorne a `String` referente a JSP.

```

@RequestMapping("finalizaTarefa")
public void finaliza(Long id, Model model) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.finaliza(id);
    model.addAttribute("tarefa", dao.buscaPorId(id));
    return "tarefa/dataFinalizada";
}

```

Agora só falta realmente criarmos o arquivo `dataFinalizada` na pasta `/WEB-INF/views/tarefa` e formatar a data usando a tag .

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<fmt:formatDate value="${tarefa.dataFinalizacao.time}" pattern="dd/MM/yyyy" />

```

Legal, já renderizamos uma data formatada e por consequência o status `200` é enviado para o jQuery. Só que até agora não fizemos nada com essa data que o JSP renderizou. Para que se pegue algo enviado pelo nosso servidor, a função de callback deve receber um parâmetro com esse conteúdo. Vamos executar um `alert` com esta variável.

```

<script type="text/javascript">
    function finalizaAgora(id) {
        $.post("finalizaTarefa", {"id": id}, function(resposta) {
            $("#tarefa_"+id).html("Finalizado");
            alert(resposta);
        });
    }
</script>

```

A execução desse `alert` nos mostra a data, falta apenas inseri-la em lugar apropriado. Vamos usar a mesma estratégia que usamos para mudar de *Não Finalizado* para *Finalizado*, atrelando um id a nossa `<td>` referente ao campo de de data.

```

<td id="tarefa_data_${tarefa.id}">
    <fmt:formatDate value="${tarefa.dataFinalizacao.time}" pattern="dd/MM/yyyy" />
</td>

```

Fazendo isso basta trocarmos o conteúdo do `<td>` na função.

```

<script type="text/javascript">
    function finalizaAgora(id) {
        $.post("finalizaTarefa", { 'id' : id}, function(resposta) {
            $("#tarefa_"+id).html("Finalizado");
            $("#tarefa_data_"+id).html(resposta);
        });
    }
</script>

```

O nosso desafio foi cumprido de forma elegante, mas ainda assim poderíamos melhorar o nosso código. A função de callback do AJAX tem que modificar dois `<td>`'s e isso tornaria repetitivo para modelos com mais alterações que a nossa `Tarefa`. Sabendo que a função recebe apenas um parâmetro de resposta, teríamos mais problemas ao ter que passar mais de um parâmetro a ela. Como resolver esta questão? Uma solução viável é passar a própria `<tr>`, completa, com as alterações necessárias. Para isso uma alteração na JSP se faz necessária.

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>

<td>${tarefa.id}</td>
<td>${tarefa.descricao}</td>
<td>Finalizada</td>
<td>
    <fmt:formatDate value="${tarefa.dataFinalizacao.time}" pattern="dd/MM/yyyy" />
</td>
<td><a href="removeTarefa?id=${tarefa.id}">Remover</a></td>
<td><a href="mostraTarefa?id=${tarefa.id}">Alterar</a></td>

```

Uma alteração no nome do arquivo `dataFinalizada.jsp` seria mais do que recomendada, já que agora não possui apenas uma data e sim todas as alterações de uma `<tr>`. Vamos renomeá-lo para `finalizada.jsp`. Sem esquecer de modificar o retorno da action de `tarefa/dataFinalizada` para `tarefa/finalizada`:

```

@RequestMapping("finalizaTarefa")
public String finaliza(Long id, Model model) {
    ...
    return "tarefa/finalizada";
}

```

Agora que retornamos uma `<tr>` com todas as alterações vamos modificar a função de callback, para que ela apenas modifique o conteúdo da `<tr>` correspondente a tarefa finalizada. Para isso é preciso diferenciar um `<tr>` do outro. Vamos utilizar agora um id na própria `<tr>` e removeremos os ids desnecessários dos `<td>`'s.

```

<table>
    ...
    <c:forEach items="${tarefas}" var="tarefa">
        <tr id="tarefa_${tarefa.id}">
            <td>${tarefa.id}</td>

```

```

<td>${tarefa.descricao}</td>

<c:if test="${tarefa.finalizado eq true}">
    <td>Finalizado</td>
</c:if>

<c:if test="${tarefa.finalizado eq false}">
    <td>
        <a href="#" onClick="finalizaAgora(${tarefa.id})">
            Finalizar
        </a>
    </td>
</c:if>

<td>
    <fmt:formatDate
        value="${tarefa.dataFinalizacao.time}"
        pattern="dd/MM/yyyy" />
</td>

<td><a href="removeTarefa?id=${tarefa.id}">Remover</a></td>
<td><a href="mostraTarefa?id=${tarefa.id}">Alterar</a></td>

</tr>

</c:forEach>
.....
</table>

```

E por último, falta mudar a função de callback para que esta modifique o conteúdo do `<tr>`.

```

...
$.post("finalizaTarefa", {'id' : id}, function(resposta) {
    $("#tarefa_"+id).html(resposta);
});
...

```

Agora sim, temos um código simples e fácil de manter. Tudo o que um bom programador gostaria de encontrar.

11.23 EXERCÍCIOS OPCIONAIS: MELHORANDO NOSSO AJAX

- Vamos buscar uma tarefa e passá-la para nossa JSP através do Model. Abra o `TarefasController.java` e modifique a action que finaliza uma tarefa para o que segue:

```

@RequestMapping("finalizaTarefa")
public String finaliza(Long id, Model model) {
    JdbcTarefaDao dao = new JdbcTarefaDao();
    dao.finaliza(id);
    model.addAttribute("tarefa", dao.buscaPorId(id));
    return "tarefa/finalizada";
}

```

- Agora falta criarmos o arquivo `finalizada.jsp` dentro do diretório: `/WEB-INF/views/tarefa/`. Que deverá ter o seguinte conteúdo da relacionada a tarefa finalizada.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
```

```

<td>${tarefa.id}</td>
<td>${tarefa.descricao}</td>
<td>Finalizada</td>
<td><fmt:formatDate value="${tarefa.dataFinalizacao.time}" pattern="dd/MM/yyyy" /></td>
<td><a href="removeTarefa?id=${tarefa.id}">Remover</a></td>
<td><a href="mostraTarefa?id=${tarefa.id}">Alterar</a></td>

```

3. Por último devemos modificar o arquivo `tarefa/lista.jsp` para que ele tenha um identificador de cada linha, ou seja, elemento da tabela. De maneira análoga ao que foi feito no exercício anterior vamos concatenar com o `id` da tarefa um valor de `tarefa_`. Lembre-se de remover os ids dos outros's já que eles não serão necessários e podem estar com o mesmo nome do identificador da .

```

.....
<c:forEach items="${tarefas}" var="tarefa">
<tr id="tarefa_${tarefa.id}">
    <td>${tarefa.id}</td>
    <td>${tarefa.descricao}</td>

    <c:if test="${tarefa.finalizado eq true}">
        <td>Finalizado</td>
    </c:if>

    <c:if test="${tarefa.finalizado eq false}">
        <td>
            <a href="#" onClick="finalizaAgora(${tarefa.id})">
                Finalizar
            </a>
        </td>
    </c:if>

    <td>
        <fmt:formatDate
            value="${tarefa.dataFinalizacao.time}"
            pattern="dd/MM/yyyy" />
    </td>
    ....
</tr>
.....

```

4. E agora para fazer uso do conteúdo renderizado pela JSP, é necessário que a função de callback do AJAX receba como parâmetro esse conteúdo. Vamos alterar a função do `finalizaAgora` no mesmo arquivo `lista.jsp`, para o que segue:

```

<script type="text/javascript">
function finalizaAgora(id) {
    $.post("finalizaTarefa", {'id' : id}, function(resposta) {
        $("#tarefa_"+id).html(resposta);
    });
}
</script>

```

5. Reinicie o servidor e verifique que agora ao clicar no link `Finalizar` o usuário tem a alteração tanto de *Não Finalizada* para *Finalizada* quando uma mudança na data de finalização.

SPRING MVC: AUTENTICAÇÃO E AUTORIZAÇÃO

"Experiência é apenas o nome que damos aos nossos erros." -- Oscar Wilde

Nesse capítulo, você aprenderá:

- O escopo de sessão e como ele funciona;
- O que são interceptadores;
- Implantar sua aplicação em qualquer container.

12.1 AUTENTICANDO USUÁRIOS: COMO FUNCIONA?

É comum hoje em dia acessarmos algum site que peça para fazermos nosso login para podermos ter acesso a funcionalidades da aplicação. Esse processo também é conhecido como autenticação. Mas, como o site sabe, nas requisições seguintes que fazemos, quem somos nós?

12.2 COOKIES

O protocolo HTTP utilizado para o acesso à páginas é limitado por não manter detalhes como quem é quem entre uma conexão e outra. Para resolver isso, foi inventado um sistema para facilitar a vida dos programadores.

Um **cookie** é normalmente um par de strings guardado no cliente, assim como um mapa de strings. Esse par de strings possui diversas limitações que variam de acordo com o cliente utilizado, o que torna a técnica de utilizá-los algo do qual não se deva confiar muito. Já que as informações do cookie são armazenadas no cliente, o mesmo pode alterá-la de alguma maneira... sendo inviável, por exemplo, guardar o nome do usuário logado...

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.

Um exemplo de bom uso de cookies é na tarefa de lembrar o nome de usuário na próxima vez que ele quiser se logar, para que não tenha que redigitar o mesmo.

Cada cookie só é armazenado para um website. Cada website possui seus próprios cookies e estes não são vistos em outra página.

12.3 SESSÃO

Usar Cookies parece facilitar muito a vida, mas através de um cookie não é possível marcar um cliente com um objeto, somente com `Strings`. Imagine gravar os dados do usuário logado através de cookies. Seria necessário um cookie para cada atributo: `usuario`, `senha`, `id`, `data de inscrição`, etc. Sem contar a falta de segurança.

O Cookie também pode estar desabilitado no cliente, sendo que não será possível lembrar nada que o usuário fez...

Uma sessão facilita a vida de todos por permitir atrelar objetos de qualquer tipo a um cliente, não sendo limitada somente à strings e é independente de cliente.

A abstração da API facilita o trabalho do programador pois ele não precisa saber como é que a sessão foi implementada no servlet container, ele simplesmente sabe que a funcionalidade existe e está lá para o uso. Se os cookies estiverem desabilitados, a sessão não funcionará e devemos recorrer para uma técnica (trabalhosa) chamada `url-rewriting`.

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.

12.4 CONFIGURANDO O TEMPO LIMITE

Para configurar 3 minutos como o padrão de tempo para o usuário perder a sessão basta incluir o seguinte código no arquivo `web.xml`:

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

12.5 REGISTRANDO O USUÁRIO LOGADO NA SESSÃO

Podemos criar uma ação que fará o login da aplicação. Caso o usuário informado na tela de login exista, guardaremos o mesmo na sessão e entraremos no sistema, e caso não exista vamos voltar para a página de login.

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

```
@Controller
public class LoginController {
```

```

public String efetuaLogin(HttpServletRequest session) {
    //....

```

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:

```

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

```

12.6 EXERCÍCIO: FAZENDO O LOGIN NA APLICAÇÃO

- Vamos criar o formulário de Login, uma ação para chamar este formulário e uma outra que realmente autentica o usuário.

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

```

<html>
    <body>
        <h2>Página de Login das Tarefas</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 tarefas" />
        </form>
    </body>
</html>

```

- Crie uma nova classe chamada `LoginController` no pacote `br.com.caelum.tarefas.controller`. Crie um método para exibir o `formulario-login.jsp`:

```

@Controller
public class LoginController{

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

```

- Na mesma 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";
}

```

- 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 Tarefas</h2>
        <p>Bem vindo, ${usuarioLogado.login}</p>
        <a href="listaTarefas">Clique aqui</a> para acessar a
        lista de tarefas
    </body>
</html>

```

- Acesse a página de login em <http://localhost:8080/fj21-tarefas/loginForm> e se logue na aplicação.
- Verifique o banco de dados para ter um login e senha válidos. Para isso, no terminal faça:

```

mysql -u root
use fj21;
select * from usuarios;

```

Se a tabela não existir, você pode criá-la executando o comando:

```

create table usuarios (
    login VARCHAR(255),
    senha VARCHAR(255)
);

```

- Caso não exista usuários cadastrados, cadastre algum utilizando o mesmo terminal aberto antes da seguinte maneira:

```

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

```

12.7 BLOQUEANDO ACESSOS DE USUÁRIOS NÃO LOGADOS COM INTERCEPTADORES

Não podemos permitir que nenhum usuário acesse as tarefas sem que ele esteja logado na aplicação, pois essa não é uma informação pública. Precisamos portanto garantir que antes de executarmos qualquer ação o usuário esteja autenticado, ou seja, armazenado na sessão.

Utilizando o 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`.

Os métodos `preHandle` e `postHandle` serão executados antes e depois, respectivamente, da ação. Enquanto o método `afterCompletion` é chamado no final da requisição, ou seja após ter renderizado o JSP.

Para nossa aplicação queremos verificar o acesso antes de executar uma ação, por isso vamos usar apenas o método `preHandle` da interface `HandlerInterceptor`. Porém, usando a interface `HandlerInterceptor` somos obrigados implementar todos os métodos definidos na interface. Queremos usar apenas o `preHandle`. Por isso 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:

```
public class AutorizadorInterceptor extends HandlerInterceptorAdapter {  
  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
        HttpServletResponse response,  
        Object controller) throws Exception {  
        //...  
        response.sendRedirect("loginForm");  
        return false;  
    }  
}
```

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();
```

Dessa maneira podemos verificar no Interceptador a existência do atributo `usuarioLogado`:

```
public class AutorizadorInterceptor extends HandlerInterceptorAdapter {  
  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
        HttpServletResponse response,  
        Object controller) throws Exception {  
  
        if(request.getSession().getAttribute("usuarioLogado") != null) {  
            return true;  
        }  
  
        response.sendRedirect("loginForm");  
        return false;  
    }  
}
```

Falta só mais uma verificação. 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. No entanto a classe `AutorizadorInterceptor` fica:

```
public class AutorizadorInterceptor extends HandlerInterceptorAdapter {  
  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
        HttpServletResponse response,  
        Object controller) throws Exception {  
  
        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;  
    }  
}
```

Ainda precisamos registrar o nosso novo interceptador. Mas o Spring MVC não permite que façamos isso via anotações, então usaremos a configuração via XML nesse caso. Já vimos o arquivo `spring-context.xml`, nele vamos usar o tag `mvc:interceptors` para cadastrar a classe `AutorizadorInterceptor`:

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

Caso seja necessário alguma ordem na execução de diversos interceptors, basta registrá-los na sequência desejada dentro da tag `mvc:interceptors`.

12.8 EXERCÍCIOS: INTERCEPTANDO AS REQUISIÇÕES

1. Vamos criar um `Interceptor` que não permitirá que o usuário acesse as ações sem antes ter logado na aplicação.
 - Crie a classe `AutorizadorInterceptor` no pacote `br.com.caelum.tarefas.interceptor`
 - Estenda a classe `HandlerInterceptorAdapter` do package `org.springframework.web.servlet.handler`
 - Sobrescreve o método `preHandle`. O usuário só pode acessar os métodos do `LoginController`

SEM ter feito o login. Caso outra lógica seja chamada é preciso verificar se o usuário existe na sessão. Existindo na sessão, seguiremos o fluxo normalmente, caso contrário indicaremos que o usuário não está logado e que deverá ser redirecionado para o formulário de login. O código completo do interceptador fica:

```
public class AutorizadorInterceptor extends HandlerInterceptorAdapter {  
  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
        HttpServletResponse response,  
        Object controller) throws Exception {  
  
        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;  
    }  
}
```

- Temos que registrar o nosso novo interceptador no XML do spring. Abra o arquivo `spring-context.xml`. Dentro da tag `<beans>` adicione:

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

2. Reinicie o servidor e tente acessar a lista de tarefas em <http://localhost:8080/fj21-tarefas/listaTarefas>. Você deverá ser redirecionado para o formulário de login.

12.9 EXERCÍCIOS OPCIONAIS: LOGOUT

1. Faça o logout da aplicação. Crie um link no `menu.jsp` que invocará um método que removerá o usuário da sessão e redirecione a navegação para a action do formulário de login (`loginForm`).

No `menu.jsp` acrescente:

```
<a href="logout">Sair do sistema</a>
```

Na classe `LoginController` adicione o método para o logout e invalide a sessão do usuário:

```
@RequestMapping("logout")  
public String logout(HttpServletRequest session) {  
    session.invalidate();
```

```
        return "redirect:loginForm";
    }
```

SPRING IOC E DEPLOY DA APLICAÇÃO

"Fazer troça da filosofia é, na verdade, filosofar" -- Blaise Pascal

Nesse capítulo, você aprenderá:

- O que é um container de inversão de controller
- Como gerenciar qualquer objeto e injetar dependências com o Spring
- Implantar sua aplicação em qualquer container.

13.1 MENOS ACOPLAMENTO COM INVERSÃO DE CONTROLE E INJEÇÃO DE DEPENDÊNCIAS

Na nossa classe `TarefasController` usamos o `JdbcTarefaDao` para executar os métodos mais comuns no banco de dados relacionado com a `Tarefa`. Em cada método do controller criamos um `JdbcTarefaDao` para acessar o banco. Repare no código abaixo quantas vezes instanciamos o DAO na mesma classe:

```
@Controller
public class TarefasController {

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

    @RequestMapping("listaTarefas")
    public String lista(Model model) {
        JdbcTarefaDao dao = new JdbcTarefaDao();
        model.addAttribute("tarefas", dao.lista());
        return "tarefa/lista";
    }

    @RequestMapping("adicionaTarefa")
    public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

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

        JdbcTarefaDao dao = new JdbcTarefaDao();
        dao.adiciona(tarefa);
        return "tarefa/adicionada";
    }
}
```

```
//outros métodos remove, altera e finaliza também criam a JdbcTarefaDao  
}
```

A classe `TarefasController` instancia um objeto do tipo `JdbcTarefaDao` manualmente. Estamos repetindo a criação do DAO em cada método. Podemos melhorar o código, criar o `JdbcTarefaDao` no construtor da classe `TarefasController` e usar um atributo. Assim podemos apagar todas as linhas de criação do DAO:

```
@Controller  
public class TarefasController {  
  
    private JdbcTarefaDao dao;  
  
    public TarefasController() {  
        this.dao = new JdbcTarefaDao();  
    }  
  
    @RequestMapping("mostraTarefa")  
    public String mostra(Long id, Model model) {  
        //dao já foi criado  
        model.addAttribute("tarefa", dao.buscaPorId(id));  
        return "tarefa/mostra";  
    }  
  
    @RequestMapping("listaTarefas")  
    public String lista(Model model) {  
        //dao já foi criado  
        model.addAttribute("tarefas", dao.lista());  
        return "tarefa/lista";  
    }  
  
    //outros métodos também aproveitam o atributo dao  
}
```

O nosso código melhorou, pois temos menos código para manter, mas há mais um problema. Repare que continuamos responsáveis pela criação do DAO. A classe que faz uso deste DAO está intimamente ligada com a maneira de instanciação do mesmo, fazendo com que ela mantenha um alto grau de **acoplamento**.

Dizemos que a classe `TarefasController` tem uma **dependência** com o `JdbcTarefaDao`. Vários métodos dela dependem do `JdbcTarefaDao`. No código acima continuamos a dar `new` diretamente na implementação `JdbcTarefaDao` para usá-la. **Nossa aplicação cria a dependência, chamando diretamente a classe específica.**

O problema em gerenciar a dependência diretamente na aplicação fica evidente se analisamos a classe `JdbcTarefaDao`. Se a classe precisar de um processo de construção mais complicado, precisaremos nos preocupar com isso em todos os lugares onde criamos o `JdbcTarefaDao`.

Para entender isso vamos verificar o construtor do DAO:

```
public class JdbcTarefaDao {
```

```

private final Connection connection;

public JdbcTarefaDao() {
    try {
        this.connection = new ConnectionFactory().getConnection();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

//métodos omitidos
}

```

Repare que aqui existe o mesmo problema. O DAO também resolve a sua dependência e cria através da `ConnectionFactory`, a conexão. Estamos acoplados à classe `ConnectionFactory` enquanto apenas queremos uma conexão. A classe está com a responsabilidade de procurar a dependência. Essa responsabilidade deve estar em outro lugar. Para melhorar, vamos então declarar a dependência em um lugar natural que é o construtor. O código fica muito mais simples e não precisa mais da `ConnectionFactory`:

```

public class JdbcTarefaDao {

    private final Connection connection;

    public JdbcTarefaDao(Connection connection) {
        this.connection = connection;
    }

    //métodos omitidos
}

```

Já com essa pequena mudança, não podemos criar o `JdbcTarefaDao` sem nos preocuparmos com a conexão antes. Veja como ficaria o código da classe `TarefasController`:

```

@Controller
public class TarefasController {

    private JdbcTarefaDao dao;

    public TarefasController() {
        try {
            Connection connection = new ConnectionFactory().getConnection();
            this.dao = new JdbcTarefaDao(connection);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    //métodos omitidos
}

```

A classe `TarefasController` não só criará o DAO, como também a conexão. Já discutimos que não queremos ser responsáveis, então vamos agir igual ao `JdbcTarefaDao` e declarar a dependência no construtor. Novamente vai simplificar demais o nosso código:

```

@Controller
public class TarefasController {

```

```

private JdbcTarefaDao dao;

public TarefasController(JdbcTarefaDao dao) {
    this.dao = dao;
}

//métodos omitidos
}

```

Assim cada classe delega a dependência para cima e não se responsabiliza pela criação. Quem for usar essa classe `TarefasController`, agora, vai precisar satisfazer as dependências. Mas quem no final se preocupa com a criação do DAO e com a conexão?

13.2 CONTAINER DE INJEÇÃO DE DEPENDÊNCIAS

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 declara que depende de alguma outra classe. E de alguma outra forma que não sabemos ainda, alguém resolverá essa dependência para nós. Alguém pega o controle dos objetos que liga (ou amarra) as dependências. Não estamos mais no controle, há algum container que gerencia as dependências e amarra tudo. Esse container já existia e já usamos ele sem saber.

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**.

Essas são as principais funcionalidades de qualquer container de inversão de controle/injeção de dependência. O Spring é um dos vários outros containers disponíveis no mundo Java. Segue uma pequena lista com os containers mais famosos:

- **Spring IoC** ou só **Spring** base de qualquer projeto Spring, popularizou DI e o desenvolvimento com POJOs
- **JBoss Seam 2** Container DI que se integra muito bem com JSF, principalmente na versão 1.2 do JSF
- **Guice** Container DI criado pelo Google que favoreceu configurações programáticas sem XML
- **CDI** Especificação que entrou com o Java EE 6, fortemente influenciado pelo JBoss Seam e Guice
- **Pico** Container muito leve e flexível, foi pioneiro nessa área
- **EJB 3** Container Java EE com um suporte limitado ao DI

13.3 CONTAINER SPRING IOC

O Spring Container sabe criar o objeto, mas também liga as dependências dele. Como o Spring está

no controle, ele administra todo o ciclo da vida. Para isso acontecer será preciso definir pelo menos duas configurações:

- declarar o objeto como componente
- ligar a dependência

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

```
@Controller
public class TarefasController {

    private JdbcTarefaDao dao;

    @Autowired
    public TarefasController(JdbcTarefaDao dao) {
        this.dao = dao;
    }

    //métodos omitidos
}
```

Para o Spring conseguir criar o `JdbcTarefaDao` vamos declarar a classe como componente. Aqui o Spring possui a anotação `@Repository` que deve ser utilizada nas classes como o DAO. Além disso, vamos também amarrar a conexão com `@Autowired`. Veja o código similar a classe `TarefasController`:

```
@Repository
public class JdbcTarefaDao {

    private final Connection connection;

    @Autowired
    public JdbcTarefaDao(Connection connection) {
        this.connection = connection;
    }

    //métodos omitidos
}
```

Ao usar `@Autowired` no construtor, o Spring tenta descobrir como abrir uma conexão, mas claro que o Container não faz ideia com qual banco queremos nos conectar. Para solucionar isso o Spring oferece uma configuração de XML que define um provedor de conexões. No mundo JavaEE, este provedor é chamado `DataSource` e abstrai as configurações de Driver, URL, etc da aplicação.

Sabendo disso, devemos declarar um `DataSource` no XML do Spring, dentro do `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/fj21"/>
    <property name="username" value="root"/>
    <property name="password" value="" />
</bean>
```

Definimos um **bean** no XML. Um *bean* é apenas um sinônimo para *componente*. Ao final, cada *bean* se torna um objeto administrado pelo Spring. Para o Spring Container, a `mysqlDataSource`, o `JdbcTarefaDao` e `TarefasController` são todos componentes(ou *beans*) que foram ligados/amarrados. Ou seja, um depende ao outro. O Spring vai criar todos e administrar o ciclo da vida deles.

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

```
@Repository
public class JdbcTarefaDao {

    private final Connection connection;

    @Autowired
    public JdbcTarefaDao(DataSource dataSource) {
        try {
            this.connection = dataSource.getConnection();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    //métodos omitidos
}
```

Pronto! Repare que no nosso projeto o *controlador* -- *depende do* --> *dao que* -- *depende do* --> *datasource*. O Spring vai primeiro criar a `mysqlDataSource`, depois o DAO e no final o controlador. Ele é o responsável pela criação de toda a cadeia de dependências.

SPRING IoC

Vimos apenas uma parte do poderoso framework Spring. A inversão de controle não só se limita a injeção de dependências, o Spring também sabe assumir outras preocupações comuns entre aplicações de negócios, como por exemplo, tratamento de transação ou segurança.

Além disso, é comum integrar outros frameworks e bibliotecas com Spring. Não é raro encontrar projetos que usam controladores MVC como Struts ou JSF com Spring. O Spring é muito flexível e sabe gerenciar qualquer componente, seja ele simples, como uma classe do nosso projeto, ou um framework sofisticado como o **Hibernate**. O treinamento FJ-27 mostra como aproveitar o melhor do Spring.

13.4 OUTRAS FORMAS DE INJEÇÃO

Vimos como injetar uma dependência com Spring usando o construtor da classe junto com a anotação `@Autowired`. O construtor é um **ponto de injeção** que deixa claro que essa dependência é

obrigatória já que não há como instanciar o objeto sem passar pelo seu construtor.

Há outros pontos de injeção como, por exemplo, o *atributo*. Podemos usar a anotação `@Autowired` diretamente no atributo. Assim o construtor não é mais necessário, por exemplo podemos injetar o `JdbcTarefaDao` na classe `TarefasController`:

```
@Controller
public class TarefasController {

    @Autowired
    JdbcTarefaDao dao;

    //sem construtor
}
```

Outra forma é criar um método dedicado para dependência, normalmente é usado um *setter* aplicando a anotação em cima do método:

```
@Controller
public class TarefasController {

    private JdbcTarefaDao dao;

    //sem construtor

    @Autowired
    public void setTarefaDao(JdbcTarefaDao dao) {
        this.dao = dao;
    }
}
```

Há vantagens e desvantagens de cada forma, mas em geral devemos favorecer o construtor já que isso é o lugar natural para a dependência. No entanto é importante conhecer os outros pontos de injeção pois alguns frameworks/bibliotecas exigem o construtor sem parâmetro, obrigando o desenvolvedor usar o atributo ou método. Como veremos no apêndice até mesmo o Spring precisa em alguns casos o construtor sem argumentos.

PARA SABER MAIS: @INJECT

A injeção de dependência é um tópico bastante difundido na plataforma Java. Existem vários container com esse poder, tanto que isso é o padrão para a maioria dos projetos Java hoje em dia.

CDI (Context e Dependency Injection, JSR-299) é a especificação que está se popularizando. O Spring só dá suporte limitado ao CDI mas aproveita as anotações padronizadas pela especificação JSR-330. Dessa maneira basta adicionar o JAR da especificação (`java.inject`) no classpath e assim podemos substituir a anotação `@Autowired` com `@Inject` :

```
@Controller
public class TarefasController {

    private JdbcTarefaDao dao;

    @Inject
    public TarefasController(JdbcTarefaDao dao) {
        this.dao = dao;
    }

    //métodos omitidos
}
```

13.5 EXERCÍCIOS: INVERSÃO DE CONTROLE COM O SPRING CONTAINER

1. Para configurar a `Datasource` é preciso copiar dois JARs.
 - Primeiro, vá ao Desktop, e entre no diretório `21/jars-datasource` .
 - Haverá dois JARs, `commons-dbcp-x.x.jar` e `commons-pool-x.x.jar` .
 - Copie-os (CTRL+C) e cole-os (CTRL+V) dentro de `workspace/fj21-tarefas/WebContent/WEB-INF/lib`
2. No arquivo `spring-context.xml` adicione a configuração da `Datasource` :

(Dica: um exemplo dessa configuração encontra-se na pasta `21/jars-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/fj21"/>
    <property name="username" value="root"/>
    <property name="password" value="" />
</bean>
```

Repare que definimos as propriedades da conexão, igual a antiga classe `ConnectionFactory` .

3. Vamos configurar a classe `JdbcTarefaDao` como componente (*Bean*) do Spring. Para isso, adicione

a anotação `@Repository` em cima da classe:

```
@Repository  
public class JdbcTarefaDao {  
    ...  
}
```

Use `Ctrl+Shift+O` para importar a anotação.

4. Além disso, altere o construtor da classe `JdbcTarefaDao`. Use a anotação `@Autowired` em cima do construtor e coloque a `DataSource` no construtor. Através dela obteremos uma nova conexão. A classe `DataSource` vem do package `javax.sql`.

Altere o construtor da classe `JdbcTarefaDao`:

```
@Autowired  
public JdbcTarefaDao(DataSource dataSource) {  
    try {  
        this.connection = dataSource.getConnection();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Ao alterar o construtor vão aparecer erros de compilação na classe `TarefasController`.

5. Abra a classe `TarefaController`. Nela vamos criar um atributo para a `JdbcTarefaDao` e gerar o construtor que recebe o DAO, novamente usando a anotação `@Autowired`:

```
@Controller  
public class TarefasController {  
  
    private final JdbcTarefaDao dao;  
  
    @Autowired  
    public TarefasController(JdbcTarefaDao dao) {  
        this.dao = dao;  
    }  
}
```

6. Altere todos os métodos que criam uma instância da classe `JdbcTarefaDao`. São justamente esse métodos que possuem erros de compilação.

Remova todas as linhas como:

```
JdbcTarefaDao dao = new JdbcTarefaDao();
```

Por exemplo, o método para listar as tarefas fica como:

```
@RequestMapping("listaTarefas")  
public String lista(Model model) {  
    model.addAttribute("tarefas", dao.lista());  
    return "tarefa/lista";  
}
```

Arrume os outros erros de compilação, apague todas as linhas que instanciam a classe

```
JdbcTarefaDao .
```

7. Reinicie o Tomcat e acesse a aplicação. Tudo deve continuar funcionando.
8. (opcional) A classe `ConnectionFactory` ainda é necessária?

13.6 APRIMORANDO O VISUAL ATRAVÉS DE CSS

Melhoramos gradativamente a nossa aplicação modificando classes e utilizando frameworks, tudo no lado do servidor, mas o que o usuário final enxerga é o HTML gerado que é exibido em seu navegador. Utilizamos algumas tags para estruturar as informações, mas nossa apresentação ainda deixa a desejar. Não há atrativo estético.

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 conteúdos da página (tabelas, parágrafos, div etc) e o CSS formata e define a aparência.

Sintaxe e inclusão de CSS

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;  
}
```

Declaração no arquivo externo

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:

```
<!DOCTYPE html>
<html>
  <head>
    <link type="text/css" rel="stylesheet" href="tarefas.css">
  </head>
  <body>
    <p>
      O conteúdo desta tag será exibido em azul com fundo amarelo!
    </p>

  </body>
</html>
```

E dentro do arquivo `tarefas.css` :

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

PROGRAMAÇÃO FRONT-END

A preocupação com a organização do código não é exclusiva do programador back-end, isto é, aquele que processa seu código no servidor.

Há também o programador front-end, aquele responsável pelo código que roda no navegador do cliente, que também deve se preocupar com a organização de seu código, inclusive com a apresentação através de CSS e de um HTML bem estruturado.

A Caelum oferece os treinamentos WD-43 e WD-47 para quem quer dominar as melhores técnicas de desenvolvimento Web com semântica perfeita, estilos CSS poderosos e JavaScripts corretos e funcionais.

13.7 EXERCÍCIOS OPCIONAIS: APLICANDO CSS NAS PÁGINAS

1.

* No seu projeto, crie uma nova pasta `css` dentro da pasta `WebContent/resources`.

* Vá ao Desktop e entre no diretório `21/css`. Copie o arquivo `tarefas.css` (`CTRL+C`) para a pasta `WebContent/resources/css` (`CTRL+V`).

1.

* Abra a página `formulario-login.jsp` que está dentro da pasta `WebContent/WEB-INF/views`.

* Na página JSP, adicione um cabeçalho(`<head></head>`) com o link para o arquivo CSS. Adicione o cabeçalho entre da tag `<html>` e `<body>`:

```

``` xml
<head>
 <link type="text/css" href="resources/css/tarefas.css" rel="stylesheet" />
</head>
```

```

Através do link definimos o caminho para o arquivo CSS.

Cuidado: _Na página deve existir apenas um cabeçalho_ (`<head></head>`).

1. Reinicie o Tomcat e chama a página de login:

<http://localhost:8080/fj21-tarefas/loginForm>

A página já deve aparecer com um visual novo.

2. Aplique o mesmo arquivo CSS em todas as outras páginas. Tenha cuidado para não repetir o cabeçalho, algumas páginas já possuem a tag `<head>`, outras não.

Verifique o resultado na aplicação.

[Criar nova tarefa](#)

| Id | Descrição | Finalizado? | Data de finalização | Remover | Alterar |
|-----------|------------------|---------------------------------|----------------------------|-------------------------|-------------------------|
| 2 | Tarefa I | Finalizado | 12/12/2013 | Remover | Alterar |
| 3 | Tarefa II | Finalizado | 26/02/2013 | Remover | Alterar |
| 4 | Tarefa III | Finaliza agora! | | Remover | Alterar |

13.8 DEPLOY DO PROJETO EM OUTROS AMBIENTES

Geralmente, ao desenvolvermos uma aplicação Web, possuímos um ambiente de desenvolvimento com um servidor que está em nossas máquinas locais. Depois que o desenvolvimento está finalizado e nossa aplicação está pronta para ir ao ar, precisamos enviá-la para um ambiente que costumamos chamar de **ambiente de produção**. Esse processo de disponibilizarmos nosso projeto em um determinado ambiente é o que chamamos de **deploy** (implantação).

O servidor de produção é o local no qual o projeto estará hospedado e se tornará acessível para os usuários finais.

Mas como fazemos para enviar nosso projeto que está em nossas máquinas locais para um servidor externo?

O processo padrão de deploy de uma aplicação web em Java é o de criar um arquivo com extensão `.war`, que nada mais é que um arquivo `zip` com o diretório base da aplicação sendo a raiz do zip.

No nosso exemplo, todo o conteúdo do diretório `WebContent` pode ser incluído em um arquivo `tarefas.war`. Após compactar o diretório, efetuaremos o *deploy*.

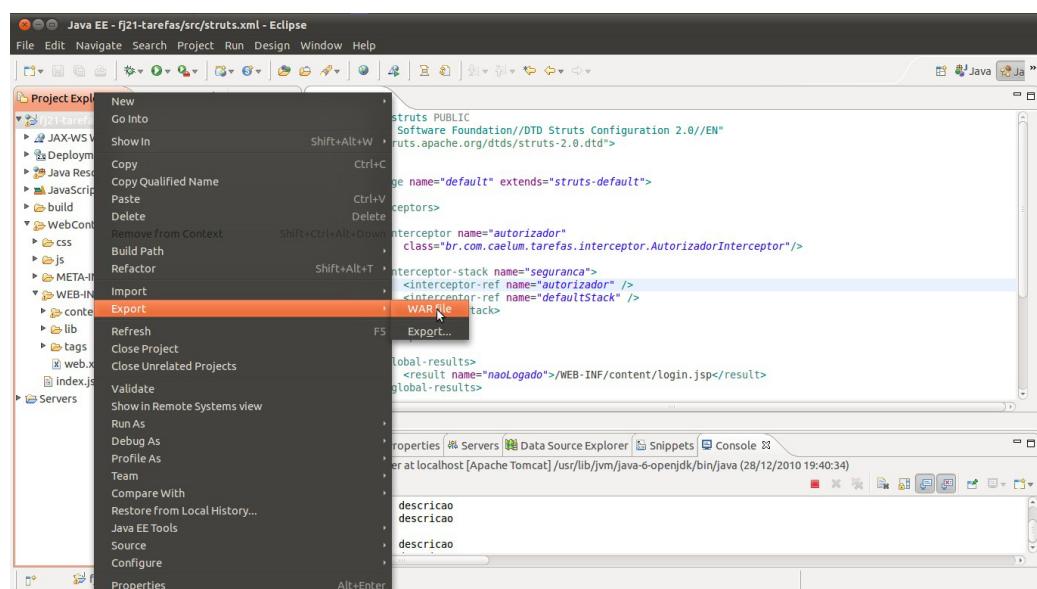
No Tomcat, basta copiar o arquivo .war no diretório **TOMCAT/webapps/**. Ele será descompactado pelo container e um novo contexto chamado **tarefas** estará disponível. Repare que o novo contexto gerado pelo Tomcat adota o mesmo nome que o seu arquivo .war , ou seja, nosso arquivo chamava-se **tarefas.war** e o contexto se chama **tarefas** .

Ao colocarmos o war no Tomcat, podemos acessar nossa aplicação pelo navegador através do endereço: <http://localhost:8080/tarefas/loginForm>

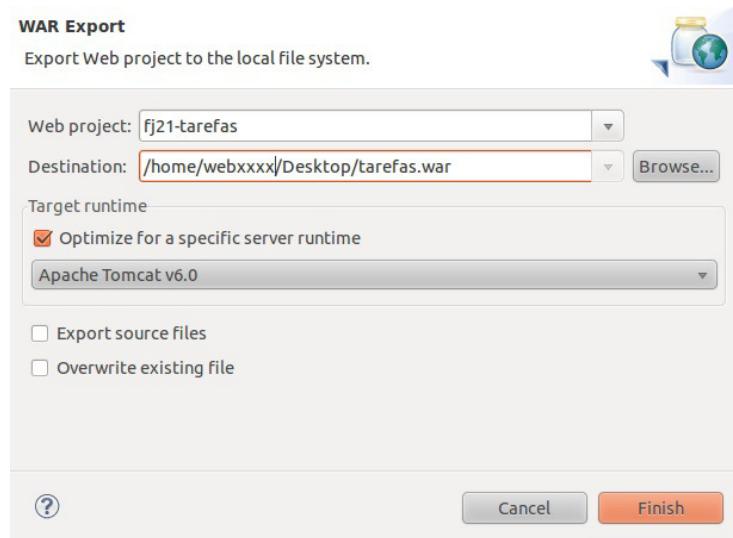
13.9 EXERCÍCIOS: DEPLOY COM WAR

1. Vamos praticar criando um war e utilizando-o, mas, antes de começarmos, certifique-se de que o Tomcat esteja no ar.

- Clique com o botão direito no projeto e vá em Export -> WAR file



- Clique no botão Browse e escolha a pasta do seu usuário e o nome **tarefas.war**.

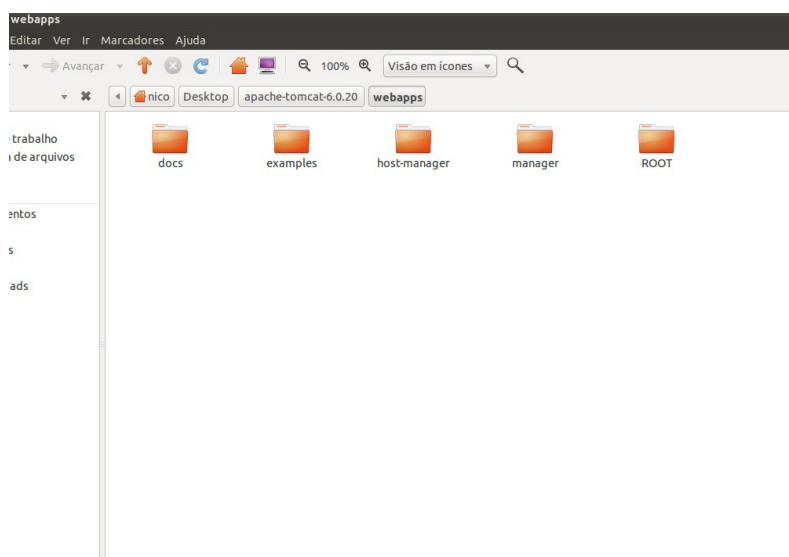


- Clique em Finish

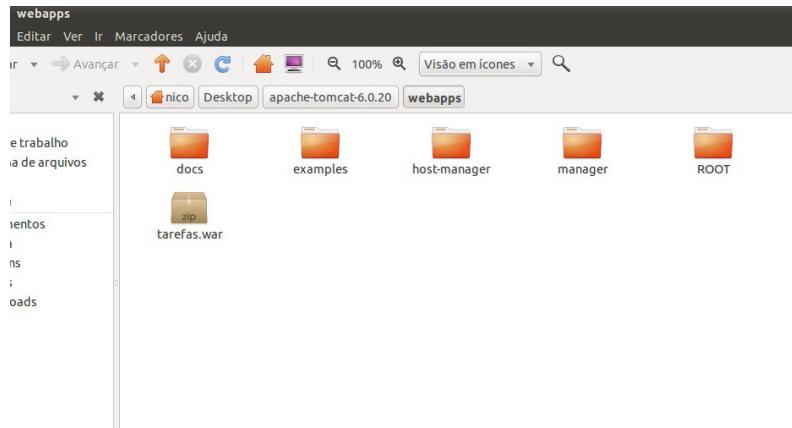
Pronto, nosso war está criado!

2. Vamos instalar nosso war!

- Abra o File Browser
- Clique da direita no arquivo `tarefas.war` e escolha **Cut**(Recortar).
- Vá para o diretório **apache-tomcat**, **webapps**. Certifique-se de que seu Tomcat esteja rodando.



- Cole o seu arquivo aqui: (**Edit** -> **Paste**). Repare que o diretório `tarefas` foi criado.



- Podemos acessar o projeto através da URL: <http://localhost:8080/tarefas/loginForm>

13.10 DISCUSSÃO EM AULA: LIDANDO COM DIFERENTES NOMES DE CONTEXTO

UMA INTRODUÇÃO PRÁTICA AO JPA COM HIBERNATE

"É uma experiência eterna de que todos os homens com poder são tentados a abusar." -- Baron de Montesquieu

Neste capítulo, você aprenderá a:

- Entender a diferença entre a JPA e o Hibernate;
- Usar a ferramenta de ORM JPA com Hibernate;
- Gerar as tabelas em um banco de dados qualquer a partir de suas classes de modelo;
- Inserir e carregar objetos pelo JPA no banco;
- Buscar vários objetos pelo JPA;

14.1 MAPEAMENTO OBJETO RELACIONAL

Com a popularização do Java em ambientes corporativos, logo se percebeu que grande parte do tempo do desenvolvedor era gasto na codificação de queries SQL e no respectivo código JDBC responsável por trabalhar com elas.

Além de um problema de produtividade, algumas outras preocupações aparecem: SQL que, apesar de ter um padrão ANSI, apresenta diferenças significativas dependendo do fabricante. Não é simples trocar um banco de dados pelo outro.

Há ainda a mudança do paradigma. A programação orientada a objetos difere muito do esquema entidade relacional e precisamos pensar das duas maneiras para fazer um único sistema. Para representarmos as informações no banco, utilizamos tabelas e colunas. As tabelas geralmente possuem chave primária (PK) e podem ser relacionadas por meio da criação de chaves estrangeiras (FK) em outras tabelas.

Quando trabalhamos com uma aplicação Java, seguimos o paradigma orientado a objetos, onde representamos nossas informações por meio de classes e atributos. Além disso, podemos utilizar também herança, composição para relacionar atributos, polimorfismo, enumerações, entre outros. Esse buraco entre esses dois paradigmas gera bastante trabalho: a todo momento devemos "transformar" objetos em registros e registros em objetos.

14.2 JAVA PERSISTENCE API E FRAMEWORKS ORM

Ferramentas para auxiliar nesta tarefa tornaram-se populares entre os desenvolvedores Java e são conhecidas como ferramentas de **mapeamento objeto-relacional** (ORM). O Hibernate é uma ferramenta ORM open source e é a líder de mercado, sendo a inspiração para a especificação **Java Persistence API** (JPA). O Hibernate nasceu sem JPA mas hoje em dia é comum acessar o Hibernate pela especificação JPA. Como toda especificação, ela deve possuir implementações. Entre as implementações mais comuns, podemos citar: Hibernate da JBoss, EclipseLink da Eclipse Foundation e o OpenJPA da Apache. Apesar do Hibernate ter originado a JPA, o EclipseLink é a implementação referencial.

O Hibernate abstrai o seu código SQL, toda a camada JDBC e o SQL será gerado em tempo de execução. Mais que isso, ele vai gerar o SQL que serve para um determinado banco de dados, já que cada banco fala um "dialeto" diferente dessa linguagem. Assim há também a possibilidade de trocar de banco de dados sem ter de alterar código Java, já que isso fica como responsabilidade da ferramenta.

Como usaremos JPA abstraímos mais ainda, podemos desenvolver sem conhecer detalhes sobre o Hibernate e até trocar o Hibernate com uma outra implementação como OpenJPA:

14.3 BIBLIOTECAS DO HIBERNATE E JPA

Vamos usar o JPA com Hibernate, ou seja, precisamos baixar os JARs no site do Hibernate. O site oficial do Hibernate é o www.hibernate.org, onde você baixa a última versão na seção ORM e Download.

Com o ZIP baixado em mãos, vamos descompactar o arquivo. Dessa pasta vamos usar todos os JARs obrigatórios (`required`). Não podemos esquecer o JAR da especificação JPA que se encontra na pasta `jpa` .

Para usar o Hibernate e JPA no seu projeto é necessário colocar todos esses JARs no *classpath*.

O Hibernate vai gerar o código SQL para qualquer banco de dados. Continuaremos utilizando o banco MySQL, portanto também precisamos o arquivo `.jar` correspondente ao driver JDBC.

14.4 MAPEANDO UMA CLASSE TAREFA PARA NOSSO BANCO DE DADOS

Para este capítulo, continuaremos utilizando a classe que representa uma tarefa:

```
package br.com.caelum.tarefas.modelo;

public class Tarefa {
    private Long id;
    private String descricao;
    private boolean finalizado;
    private Calendar dataFinalizacao;
```

}

Criamos os getters e setters para manipular o objeto, mas fique atento que só devemos usar esses métodos se realmente houver necessidade.

Essa é uma classe como qualquer outra que aprendemos a escrever em Java. Precisamos configurar o Hibernate para que ele saiba da existência dessa classe e, desta forma, saiba que deve inserir uma linha na tabela `Tarefa` toda vez que for requisitado que um objeto desse tipo seja salvo. Em vez de usarmos o termo "configurar", falamos em **mapear** uma classe a tabela.

Para mapear a classe `Tarefa`, basta adicionar algumas poucas **anotações** em nosso código. Anotação é um recurso do Java que permite inserir **metadados** em relação a nossa classe, atributos e métodos. Essas anotações depois poderão ser lidas por frameworks e bibliotecas, para que eles tomem decisões baseadas nessas pequenas configurações.

Para essa nossa classe em particular, precisamos de apenas quatro anotações:

```
@Entity
public class Tarefa {

    @Id
    @GeneratedValue
    private Long id;

    private String descricao;
    private boolean finalizado;

    @Temporal(TemporalType.DATE)
    private Calendar dataFinalizacao;

    // métodos...
}
```

`@Entity` indica que objetos dessa classe se tornem "persistível" no banco de dados. `@Id` indica que o atributo `id` é nossa chave primária (você precisa ter uma chave primária em toda entidade) e `@GeneratedValue` diz que queremos que esta chave seja populada pelo banco (isto é, que seja usado um `auto increment` ou `sequence`, dependendo do banco de dados). Com `@Temporal` configuramos como mapear um `Calendar` para o banco, aqui usamos apenas a data (sem hora), mas poderíamos ter usado apenas a hora (`TemporalType.TIME`) ou timestamp (`TemporalType.TIMESTAMP`). Essas anotações precisam dos devidos `imports`, e pertencem ao pacote `javax.persistence`.

Mas em que tabela essa classe será gravada? Em quais colunas? Que tipo de coluna? Na ausência de configurações mais específicas, o Hibernate vai usar convenções: a classe `Tarefa` será gravada na tabela de nome também `Tarefa`, e o atributo `descricao` em uma coluna de nome `descricao` também!

Se quisermos configurações diferentes das convenções, basta usarmos outras anotações, que são completamente opcionais. Por exemplo, para mapear o atributo `dataFinalizacao` numa coluna chamada `data_finalizado` farímos:

```
@Column(name = "data_finalizado", nullable = true)
private Calendar dataFinalizacao;
```

Para usar uma tabela com o nome `tarefas` :

```
@Entity
@Table(name="tarefas")
public class Tarefa {
```

Repare que nas entidades há todas as informações sobre as tabelas. Baseado nelas podemos até pedir gerar as tabelas no banco, mas para isso é preciso configurar o JPA.

14.5 CONFIGURANDO O JPA COM AS PROPRIEDADES DO BANCO

Em qual banco de dados vamos gravar nossas `Tarefa`s? Qual é o login? Qual é a senha? O JPA necessita dessas configurações, e para isso criaremos o arquivo `persistence.xml`.

Alguns dados que vão nesse arquivo são específicos do Hibernate e podem ser bem avançados, sobre controle de cache, transações, connection pool etc, tópicos que são abordados no curso FJ-25.

Para nosso sistema, precisamos de quatro linhas com configurações que já conhecemos do JDBC: string de conexão com o banco, o driver, o usuário e senha. Além dessas quatro configurações, precisamos dizer qual dialeto de SQL deverá ser usado no momento que as queries são geradas; no nosso caso, MySQL. Vamos também mostrar o SQL no console para acompanhar o trabalho do Hibernate.

Vamos também configurar a criação das tabelas baseado nas entidades.

Segue uma configuração completa que define uma unidade de persistência (*persistence-unit*) com o nome `tarefas`, seguidos pela definição do provedor, entidades e properties:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

<persistence-unit name="tarefas">

  <!-- provedor/implementacao do JPA -->
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

  <!-- entidade mapeada -->
  <class>br.com.caelum.tarefas.modelo.Tarefa</class>

  <properties>
    <!-- dados da conexao -->
    <property name="javax.persistence.jdbc.driver"
      value="com.mysql.jdbc.Driver" />
    <property name="javax.persistence.jdbc.url"
      value="jdbc:mysql://localhost/fj21" />
    <property name="javax.persistence.jdbc.user"
      value="root" />
    <property name="javax.persistence.jdbc.password"
      value="" />
  </properties>
</persistence-unit>
```

```

<!-- propriedades do hibernate -->
<property name="hibernate.dialect"
      value="org.hibernate.dialect.MySQL5InnoDBDialect" />
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />

<!-- atualiza o banco, gera as tabelas se for preciso -->
<property name="hibernate.hbm2ddl.auto" value="update" />

</properties>
</persistence-unit>
</persistence>

```

É importante saber que o arquivo `persistence.xml` deve ficar na pasta **META-INF** do seu *classpath*.

14.6 USANDO O JPA

Para usar o JPA no nosso código Java, precisamos utilizar sua API. Ela é bastante simples e direta e rapidamente você estará habituado com suas principais classes.

Nosso primeiro passo é fazer com que o JPA leia a nossa configuração: tanto o nosso arquivo `persistence.xml` quanto as anotações que colocamos na nossa entidade `Tarefa`. Para tal, usaremos a classe com o mesmo nome do arquivo XML: `Persistence`. Ela é responsável de carregar o XML e inicializar as configurações. Resultado dessa configuração é uma `EntityManagerFactory`:

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
```

Estamos prontos para usar o JPA. Antes de gravar uma `Tarefa`, precisamos que exista a tabela correspondente no nosso banco de dados. Em vez de criarmos o script que define o *schema* (ou DDL de um banco, *data definition language*) do nosso banco (os famosos `CREATE TABLE`) podemos deixar isto a cargo do próprio Hibernate. Ao inicializar a `EntityManagerFactory` também já será gerada uma tabela `Tarefas` pois configuramos que o banco deve ser atualizada pela propriedade do Hibernate: `hbm2ddl.auto`.

14.7 PARA SABER MAIS: CONFIGURANDO O JPA COM HIBERNATE EM CASA

Caso você esteja fazendo esse passo de casa. É preciso baixar o ZIP do Hibernate ORM 4.x (<http://hibernate.org>), extraí-lo, e copiar todos os JARs das pastas `lib/required` e `lib/jpa`.

Para essa aplicação usaremos as seguintes versões:

- `antlr-2.7.7.jar`
- `dom4j-1.6.1.jar`
- `hibernate-commons-annotations-4.0.4.Final.jar`

- hibernate-core-4.3.0.Final.jar
- hibernate-entitymanager-4.3.0.Final.jar
- hibernate-jpa-2.1-api-1.0.0.Final.jar
- jandex-1.1.0.Final.jar
- javassist-3.18.1-GA.jar
- jboss-logging-3.1.3.GA.jar
- jboss-logging-annotations-1.2.0.Beta1.jar
- jboss-transaction-api_1.2_spec-1.0.0.Final.jar

14.8 EXERCÍCIOS: CONFIGURANDO O JPA E GERANDO O SCHEMA DO BANCO

1. Vamos preparar nosso projeto `fj21-tarefas` com as dependências do Hibernate.

Copie os JARs do Hibernate para a pasta **WebContent/WEB-INF/lib** do seu projeto dessa forma:

- Vá ao diretório `21/jars-jpa/hibernate` ;
- Selecione todos os JARs, clique com o botão direito e escolha *Copy* (ou `CTRL+C`) ;
- Cole todos os JARs na pasta **WebContent/WEB-INF/lib** do projeto `fj21-tarefas` (`CTRL+V`)

2. Anote a classe `Tarefa` como uma entidade de banco de dados. Lembre-se que essa é uma anotação do pacote `javax.persistence` .

Obs: Não apague nenhuma anotação, apenas adicione as anotações do JPA.

```
@Entity
public class Tarefa {

}
```

Na mesma classe anote seu atributo `id` como chave primária e como campo de geração automática:

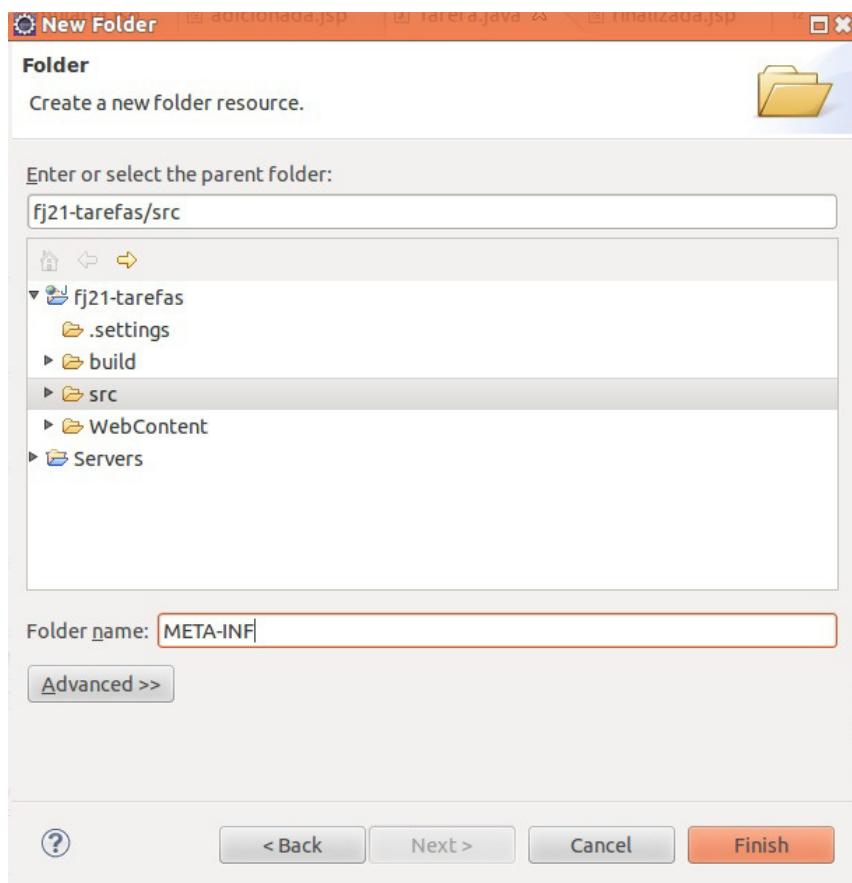
```
@Id
@GeneratedValue
private Long id;
```

Agora é preciso mapear o atributo `dataFinalizacao` para gravar apenas a data (sem hora) no banco:

```
@Temporal(TemporalType.DATE)
private Calendar dataFinalizacao;
```

- 3.

- Clique com o botão direito na pasta `src` do projeto do `fj21-tarefas` e escolha *New -> Folder*. Escolha **META-INF** como nome dessa pasta.



- Vá ao diretório `caelum/21/jars-jpa` e copie o arquivo `persistence.xml` para a pasta `META-INF`.
- O arquivo é apenas um esboço, falta configurar a unidade de persistência com o provedor, entidades e propriedades. Adicione dentro do elemento `persistence`:

```

<persistence-unit name="tarefas">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>br.com.caelum.tarefas.modelo.Tarefa</class>

    <properties>
        <property name="javax.persistence.jdbc.driver"
                  value="com.mysql.jdbc.Driver" />
        <property name="javax.persistence.jdbc.url"
                  value="jdbc:mysql://localhost/fj21" />
        <property name="javax.persistence.jdbc.user" value="root" />
        <property name="javax.persistence.jdbc.password" value="" />

        <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 duas propriedades `show_sql` e `format_sql` fazem com que todo SQL gerado pelo Hibernate apareça no console.

4. Crie a classe `GeraTabelas` no pacote `br.com.caelum.tarefas.jpa`.

```
package br.com.caelum.tarefas.jpa;

// imports omitidos

public class GeraTabelas {

    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence.
            createEntityManagerFactory("tarefas");

        factory.close();
    }
}
```

5. Crie sua tabela executando a classe `GeraTabelas`. Para isso, clique da direita no código e vá em *Run As -> Java Application*.

Não é preciso rodar o Tomcat para esse exercício.

6. Agora, abra uma nova aba no **Terminal** e digite `mysql -u root`. Após isto, digite `use fj21;` e em seguida, `show tables;`. Se seu banco de dados já existia, e não foi preciso criá-lo no passo anterior, você vai notar a presença de uma tabela chamada `tarefas`. Não é esta a tabela que queremos. Procuramos pela tabela `Tarefa`, com T, em maiúsculo (igual ao nome da classe `Tarefa`).

```
mysql> select * from Tarefa;
+-----+-----+-----+
| id | dataFinalizacao | descricao          | finalizado |
+-----+-----+-----+
| 1  | 2013-02-25      | Estudar JPA e Hibernate | 1           |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> ■
```

7. (opcional) Caso algum erro tenha ocorrido, é possível que o Hibernate tenha logado uma mensagem, mas você não a viu dado que o Log4J não está configurado. Mesmo que tudo tenha ocorrido de maneira correta, é muito importante ter o Log4J configurado.

Para isso, adicione no arquivo `log4j.properties` dentro da pasta `src` para que todo o log do nível `info` ou acima seja enviado para o console appender do `System.out` (default do console):

```
log4j.logger.org.hibernate=info
```

14.9 TRABALHANDO COM OS OBJETOS: O ENTITYMANAGER

Para se comunicar com o JPA, precisamos de uma instância de um objeto do tipo `EntityManager`.

Adquirimos uma `EntityManager` através da fábrica já conhecida: `EntityManagerFactory`.

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();
manager.close();
factory.close();
```

Persistindo novos objetos

Através de um objeto do tipo `EntityManager`, é possível gravar novos objetos no banco. Para isto, basta utilizar o método `persist` dentro de uma transação:

```
Tarefa tarefa = new Tarefa();
tarefa.setDescricao("Estudar JPA");
tarefa.setFinalizado(true);
tarefa.setDataFinalizacao(Calendar.getInstance());

EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();

manager.getTransaction().begin();
manager.persist(tarefa);
manager.getTransaction().commit();

System.out.println("ID da tarefa: " + tarefa.getId());

manager.close();
```

CUIDADOS AO USAR O JPA

Ao usar o JPA profissionalmente, fique atento com alguns cuidados que são simples, mas não dada a devida atenção podem gerar gargalos de performance. Abrir `EntityManager` indiscriminadamente é um desses problemas. Esse post da Caelum indica outros:

<http://bit.ly/bRc5g>

Esses detalhes do dia a dia assim como JPA com Hibernate avançado são vistos no curso FJ-25 da Caelum, **de Hibernate e JPA avançados**.

Carregar um objeto

Para buscar um objeto dada sua chave primária, no caso o seu `id`, utilizamos o método `find`, conforme o exemplo a seguir:

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();

Tarefa encontrada = manager.find(Tarefa.class, 1L);
```

```
System.out.println(encontrada.getDescricao());
```

14.10 EXERCÍCIOS: GRAVANDO E CARREGANDO OBJETOS

- Crie uma classe chamada `AdicionaTarefa` no pacote `br.com.caelum.tarefas.jpa`, ela vai criar um objeto e adicioná-lo ao banco:

```
package br.com.caelum.tarefas.jpa;

// imports omitidos

public class AdicionaTarefa {

    public static void main(String[] args) {

        Tarefa tarefa = new Tarefa();
        tarefa.setDescricao("Estudar JPA e Hibernate");
        tarefa.setFinalizado(true);
        tarefa.setDataFinalizacao(Calendar.getInstance());

        EntityManagerFactory factory = Persistence.
            createEntityManagerFactory("tarefas");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();
        manager.persist(tarefa);
        manager.getTransaction().commit();

        System.out.println("ID da tarefa: " + tarefa.getId());

        manager.close();
    }
}
```

- Rode a classe `AdicionaTarefa` e adicione algumas tarefas no banco. Saída possível:

```
<terminated> AdicionaTarefa [Java Application] /usr/lib/jvm/java-7-oracle/bin/java (21/02/2013 14:49:32)
Hibernate:
    insert
    into
        Tarefa
        (dataFinalizacao, descricao, finalizado)
    values
        (?, ?, ?)
ID da tarefa: 1
```

- Verifique os registros no banco, se conecte com o cliente do mysql:

```
mysql -u root
use fj21;
select * from Tarefa;
```

Rode novamente a classe `AdicionaTarefa` e verifique o banco.

- Vamos carregar tarefas pela chave primária.

Crie uma classe chamada `CarregaTarefa` no pacote `br.com.caelum.tarefas.jpa`:

```

package br.com.caelum.tarefas.jpa;

// imports omitidos

public class CarregaTarefa {

    public static void main(String[] args) {

        EntityManagerFactory factory = Persistence.
            createEntityManagerFactory("tarefas");
        EntityManager manager = factory.createEntityManager();

        Tarefa encontrada = manager.find(Tarefa.class, 1L);
        System.out.println(encontrada.getDescricao());

        manager.close();
    }
}

```

5. Rode a classe `CarregaTarefa` e verifique a saída.

Caso recebeu uma exception, verifique o id da tarefa. Ele deve existir no banco.

```

Markers Servers Data Source Explorer Snippets Console
<terminated> CarregaTarefa (1) [Java Application] /usr/lib/jvm/java-7-oracle/bin/java (25/02/2013 13:36:30)
Hibernate:
    select
        tarefa0_.id as id0_0,
        tarefa0_.dataFinalizacao as dataFina2_0_0,
        tarefa0_.descricao as descricao0_0,
        tarefa0_.finalizado as finalizado0_0
    from
        Tarefa tarefa0_
    where
        tarefa0_.id=?
Estudar JPA e Hibernate

```

14.11 REMOVENDO E ATUALIZANDO OBJETO

Remover e atualizar os objetos com JPA também é muito simples. O `EntityManager` possui métodos para cada operação. Para remover é preciso carregar a entidade antes e depois usar o método `remove`:

```

EntityManager manager = //abrir um EntityManager
Tarefa encontrada = manager.find(Tarefa.class, 1L);

manager.getTransaction().begin();
manager.remove(encontrada);
manager.getTransaction().commit();

```

Para atualizar um entidade que já possui um id existe o método `merge`, por exemplo:

```

Tarefa tarefa = new Tarefa();
tarefa.setId(2); //esse id já existe no banco
tarefa.setDescricao("Estudar JPA e Hibernate");
tarefa.setFinalizado(false);
tarefa.setDataFinalizacao(null);

EntityManager manager = //abrir um EntityManager

manager.getTransaction().begin();
manager.merge(tarefa);

```

```
manager.getTransaction().commit();
```

14.12 BUSCANDO COM UMA CLÁUSULA WHERE

O JPA possui uma linguagem própria de queries para facilitar a busca de objetos chamada de JPQL. O código a seguir mostra uma pesquisa que retorna todas as tarefas não finalizadas:

```
EntityManager manager = //abrir um EntityManager
List<Tarefa> lista = manager
    .createQuery("select t from Tarefa as t where t.finalizado = false")
    .getResultList();

for (Tarefa tarefa : lista) {
    System.out.println(tarefa.getDescricao());
}
```

Também podemos passar um parâmetro para a pesquisa. Para isso, é preciso trabalhar com um objeto que representa a pesquisa (`javax.persistence.Query`):

```
EntityManager manager = //abrir um EntityManager

Query query = manager
    .createQuery("select t from Tarefa as t "+
        "where t.finalizado = :paramFinalizado");
query.setParameter("paramFinalizado", false);

List<Tarefa> lista = query.getResultList();
```

Este é apenas o começo. O JPA é muito poderoso e, no curso FJ-25, veremos como fazer queries complexas, com joins, agrupamentos, projeções, de maneira muito mais simples do que se tivéssemos de escrever as queries. Além disso, o JPA com Hibernate é muito customizável: podemos configurá-lo para que gere as queries de acordo com dicas nossas, dessa forma otimizando casos particulares em que as queries que ele gera por padrão não são desejáveis.

Uma confusão que pode ser feita a primeira vista é pensar que o JPA com Hibernate é lento, pois, ele precisa gerar as nossas queries, ler objetos e suas anotações e assim por diante. Na verdade, o Hibernate faz uma série de otimizações internamente que fazem com que o impacto dessas tarefas seja próximo a nada. Portanto, o Hibernate é, sim, performático, e hoje em dia pode ser utilizado em qualquer projeto que se trabalha com banco de dados.

14.13 EXERCÍCIOS: BUSCANDO COM JPQL

1. Crie uma classe chamada `BuscaTarefas` no pacote `br.com.caelum.tarefas.jpa` :

Cuidado, a classe `Query` vem do pacote `javax.persistence` :

```
package br.com.caelum.tarefas.jpa;

import javax.persistence.Query;
// outros imports omitidos
```

```

public class BuscaTarefas {

    public static void main(String[] args) {

        EntityManagerFactory factory = Persistence.
            createEntityManagerFactory("tarefas");
        EntityManager manager = factory.createEntityManager();

        //cuidado, use o import javax.persistence.Query
        Query query = manager
            .createQuery("select t from Tarefa as t "+
                "where t.finalizado = :paramFinalizado");
        query.setParameter("paramFinalizado", true);

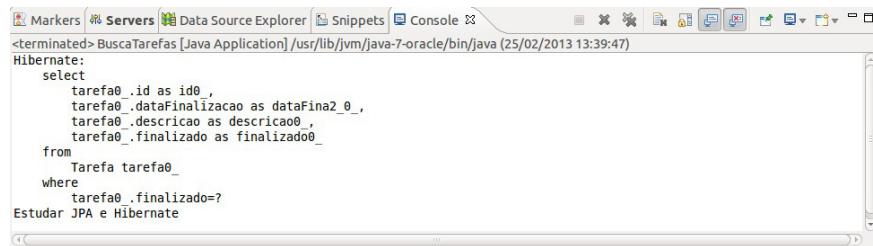
        List<Tarefa> lista = query.getResultList();

        for (Tarefa t : lista) {
            System.out.println(t.getDescricao());
        }

        manager.close();
    }
}

```

2. Rode a classe `BuscaTarefas` e verifique a saída.



```

Markers Servers Data Source Explorer Snippets Console
<terminated> BuscaTarefas [Java Application] /usr/lib/jvm/java-7-oracle/bin/java (25/02/2013 13:39:47)
Hibernate:
    select
        tarefa0_.id as id0_,
        tarefa0_.dataFinalizacao as dataFina2_0_,
        tarefa0_.descricao as descricao0_,
        tarefa0_.finalizado as finalizado0_
    from
        Tarefa tarefa0_
    where
        tarefa0_.finalizado=?
Estudar JPA e Hibernate

```

E AGORA?

"A nuca é um mistério para a vista." -- Paul Valéry

O curso **FJ-21, Java para desenvolvimento Web**, procura abordar as principais tecnologias do mercado Web em Java desde seus fundamentos até os frameworks mais usados.

Mas e agora, para onde direcionar seus estudos e sua carreira?

15.1 OS APÊNDICES DESSA APOSTILA

Os próximos capítulos da apostila são apêndices extras para expandir seu estudo em algumas áreas que podem ser de seu interesse:

- **JPA e Spring** - Mostra como integrar JPA com Spring para utilizar JPA na camada de persistência no projeto `fj21-tarefas`.
- **Servlets 3.0 e Java EE 6** - Mostra as últimas novidades do Java EE 6 na parte de Web lançadas em dezembro de 2009. É ainda uma versão muito recente que o mercado não adota e poucos servidores suportam. Mas é bom estar atualizado com as próximas novidades que se tornarão realidade no mercado em pouco tempo.
- **Tópicos da Servlet API** - Aborda vários tópicos sobre Servlets e JSPs que não abordamos antes. São detalhes a mais e alguns recursos mais avançados. É interessante para expandir seu conhecimento e ajudar entender melhor o padrão Servlets.

15.2 FRAMEWORKS WEB

O mercado de frameworks Java é imenso. Há muitas opções boas disponíveis no mercado e muitos pontos a considerar na adoção ou estudo de algum novo framework.

O **Struts** com certeza é o framework com maior unanimidade no mercado, em especial por causa de sua versão 1.x. A versão 2.0 não tem tanta força mas é um dos mais importantes.

O **VRaptor**, criado na USP em 2004 e mantido hoje pela Caelum e por uma grande comunidade, não tem o tamanho do mercado de outros grandes frameworks. Mas tem a vantagem da extrema simplicidade e grande produtividade, além de ser bem focado no mercado brasileiro, com documentação

em português e muito material disponível. A Caelum inclusive disponibiliza um curso de VRaptor, o **FJ-28** que possui sua apostila disponível para download gratuito na Internet em:

<http://www.caelum.com.br/apostilas>

Um dos frameworks mais usados hoje é o **JavaServer Faces - JSF**. Seu grande apelo é ser o framework oficial do Java EE para Web, enquanto que todos os outros são de terceiros. O JSF tem ainda muitas características diferentes do Struts ou do VRaptor que vimos nesse curso.

Em especial, o JSF é dito um framework *component-based*, enquanto que Struts (1 e 2) e VRaptor são ditos *request-based* ou *action-based*. A ideia principal de um framework de componentes é abstrair muitos dos conceitos da Web e do protocolo HTTP provendo uma forma de programação mais parecida com programação para Desktop. O JSF tem componentes visuais ricos já prontos, funciona através de tratamento de eventos e é *stateful* (ao contrário da Web "normal" onde tudo é *stateless*). Na Caelum, o curso **FJ-26** trata de JavaServer Faces:

<http://www.caelum.com.br/curso/fj26>

Mas o JSF não é único framework baseado em componentes. Há outras opções (menos famosas) como o **Google Web Toolkit - GWT** ou o **Apache Wicket**. Da mesma forma, há outros frameworks *request-based* além de Struts e VRaptor, como o **Stripes** ou o **Spring MVC**. Na Caelum, o Spring e Spring MVC é tratado em detalhes no curso **FJ-27**, junto com outras funcionalidades do Spring:

<http://www.caelum.com.br/curso/fj27>

Toda essa pluralidade de frameworks é boa para fomentar competição e inovação no mercado, mas pode confundir muito o programador que está iniciando nesse meio. Um bom caminho a seguir após esse curso FJ-21 é continuar aprofundando seus conhecimentos no Spring MVC e no VRaptor (com ajuda da apostila aberta) e partir depois para o estudo do JSF. Os outros frameworks você vai acabar eventualmente encontrando algum dia em sua carreira, mas não se preocupe em aprender todos no começo.

15.3 FRAMEWORKS DE PERSISTÊNCIA

Quando falamos de frameworks voltados para persistência e bancos de dados, felizmente, não há tanta dúvida quanto no mundo dos frameworks Web. O **Hibernate** é praticamente unanimidade no mercado Java, principalmente se usado como implementação da **JPA**. Há outras possibilidades, como o Toplink, o EclipseLink, o OpenJPA, o DataNucleus, o JDO, o iBatis etc. Mas o Hibernate é o mais importante.

Na Caelum, abordamos JPA e Hibernate avançado no curso **FJ-25**:

<http://www.caelum.com.br/curso/fj25>

Há ainda livros e documentações disponíveis sobre Hibernate. Recomendamos fortemente que você aprofunde seus estudos no Hibernate que é muito usado e pedido no mercado Java hoje.

15.4 ONDE SEGUIR SEUS ESTUDOS

A Caelum disponibiliza para download gratuito algumas das apostilas de seus treinamentos. Você pode baixar direto no site:

<http://www.caelum.com.br/apostilas>

O Blog da Caelum é ainda outra forma de acompanhar novidades e expandir seus conhecimentos:

<http://blog.caelum.com.br>

Diversas revistas, no Brasil e no exterior, estudam o mundo Java como ninguém e podem ajudar o iniciante a conhecer muito do que está acontecendo lá fora nas aplicações comerciais.

Lembrando que a editora Casa do Código possui livros de JSF e JPA que podem te auxiliar também na continuação e reforço dos seus estudos:

<http://www.CasaDoCodigo.com.br>

APÊNDICE - INTEGRAÇÃO DO SPRING COM JPA

"O caminho do inferno está pavimentado de boas intenções." -- Marx

Neste capítulo, você aprenderá a:

- Integrar JPA com Spring;
- Injetar o EntityManager dentro do DAO;
- Gerenciar as transações com Spring;

16.1 GERENCIANDO O ENTITYMANAGER

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**.

Vimos que o Spring é um container que controla objetos (ou componentes), administrando o ciclo da vida deles, inclusive ligando uns aos outros (amarrando-os). No caso do JPA, queremos que o Spring cuide da abertura e do fechamento da `EntityManagerFactory` e do `EntityManager`. Com isso, **todos** os componentes do Spring podem receber como dependência um `EntityManager`, mas agora controlado pelo Spring. Novamente aplicando a inversão de controle.

Mas a inversão de controle não se limita a inicialização de objetos. O Spring também tira do desenvolvedor a responsabilidade de controlar a transação. Ao delegar o controle do `EntityManager` para o Spring, ele consegue abrir e fechar transações automaticamente.

Veja como é a inicialização do JPA sem a ajuda do Spring:

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();

manager.getTransaction().begin();

//aqui usa o EntityManager

manager.getTransaction().commit();
manager.close();
```

Nesse pequeno trecho de código podemos ver como é trabalhoso inicializar o JPA manualmente. É preciso abrir e fechar todos os recursos para realmente começar a usar o `EntityManager`. O Spring

deve assumir essas responsabilidades e facilitar assim o uso do JPA.

16.2 CONFIGURANDO O JPA NO SPRING

Para integrar-se com o JPA, o Spring disponibiliza um *Bean* que devemos cadastrar no arquivo XML. Ele representa a `EntityManagerFactory`, mas agora gerenciada pelo Spring. Ou seja, toda inicialização da fábrica fica ao encargo do 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>
```

Repare que o *Bean* define *Hibernate* como implementação do JPA e recebe a `mysqlDataSource` que já definimos anteriormente dentro do XML do Spring. Como a nossa `Datasource` já sabe os dados do driver, login e senha, o arquivo `persistence.xml` do JPA também fica mais simples:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

    <persistence-unit name="tarefas">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>br.com.caelum.tarefas.modelo.Tarefa</class>
        <properties>
            <!-- SEM as propriedades URL, login, senha e driver -->
            <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>
</persistence>
```

16.3 INJETANDO O ENTITYMANAGER

Com JPA configurado podemos aproveitar a inversão de controle e injetar o `EntityManager` dentro de qualquer componente administrado pelo Spring.

Vamos criar uma classe `JpaTarefaDao` para usar o `EntityManager`. A classe `JpaTarefaDao`

precisa do `EntityManager`. Como JPA é uma especificação, o Spring também aproveita uma anotação de especificação para receber a dependência. Nesse caso não podemos usar a anotação `@Autowired` do Spring e sim `@PersistenceContext`. Infelizmente a anotação `@PersistenceContext` não funciona com construtores, exigindo um outro *ponto de injecção*. Usaremos o atributo para declarar a dependência:

```
@Repository
public class JpaTarefaDao{

    @PersistenceContext
    private EntityManager manager;

    //sem construtor

    //aqui vem os métodos
}
```

Nos métodos do `JpaTarefaDao` faremos o uso do `EntityManager` usando os métodos já conhecidos como `persist(..)` para persistir uma tarefa ou `remove(..)` para remover:

```
@Repository
public class JpaTarefaDao{

    @PersistenceContext
    private EntityManager manager;

    //sem construtor

    public void adiciona(Tarefa tarefa) {
        manager.persist(tarefa);
    }

    public void altera(Tarefa tarefa) {
        manager.merge(tarefa);
    }

    public List<Tarefa> lista() {
        return manager.createQuery("select t from Tarefa t").getResultList();
    }

    public Tarefa buscaPorId(Long id) {
        return manager.find(Tarefa.class, id);
    }

    public void remove(Tarefa tarefa) {
        Tarefa tarefaARemover = buscaPorId(tarefa.getId());
        manager.remove(tarefaARemover);
    }

    public void finaliza(Long id) {
        Tarefa tarefa = buscaPorId(id);
        tarefa.setFinalizado(true);
        tarefa.setDataFinalizacao(Calendar.getInstance());
        manager.merge(tarefa);
    }
}
```

A implementação do `JpaTarefaDao` ficou muito mais simples se comparada com o

`JdbcTarefaDao` , em alguns métodos é apenas uma linha de código.

16.4 BAIXO ACOPLAGEMTO PELO USO DE INTERFACE

Voltando para nossa classe `TarefasController` vamos lembrar que injetamos o `JdbcTarefaDao` para trabalhar com o banco de dados:

```
@Controller
public class TarefasController {

    private final JdbcTarefaDao dao;

    @Autowired
    public TarefasController(JdbcTarefaDao dao) {
        this.dao = dao;
    }

    @RequestMapping("mostraTarefa")
    public String mostra(Long id, Model model) {
        model.addAttribute("tarefa", dao.buscaPorId(id));
        return "tarefa/mostra";
    }

    //outros métodos omitidos
}
```

Quando olhamos para os métodos propriamente ditos, por exemplo, `mostra` , percebemos que o que é realmente necessário para executá-lo é *algum DAO*. A lógica em si é independente da instância específica que estamos instanciando, ou seja para a classe `TarefasController` não importa se estamos usando JDBC ou JPA. Queremos um desacoplamento da implementação do DAO específico, e algo deve decidir qual implementação usaremos por baixo dos panos.

O problema desse tipo de chamada é que, no dia em que precisarmos mudar a implementação do DAO, precisaremos mudar nossa classe. Agora imagine que tenhamos 10 controladores no sistema que usem nosso `JdbcTarefaDao` . Se todas usam a implementação específica, quando formos mudar a implementação para usar JPA para persistência por exemplo, digamos, `JpaTarefaDao` , precisaremos mudar em vários lugares.

O que precisamos então é apenas uma `TarefaDao` dentro da classe `TarefasController` , então vamos definir (*extrair*) uma nova interface `TarefaDao` :

```
public interface TarefaDao {

    Tarefa buscaPorId(Long id);
    List<Tarefa> lista();
    void adiciona(Tarefa t);
    void altera(Tarefa t);
    void remove(Tarefa t);
    void finaliza(Long id);
}
```

E a classe `JdbcTarefaDao` implementará essa interface:

```

@Repository
public class JdbcTarefaDao implements TarefaDao {

    //implementação do nosso dao usando jdbc
}

```

Dessa maneira vamos injetar uma implementação compatível com a interface dentro da classe TarefasController

```

@Controller
public class TarefasController {

    private TarefaDao dao; //usando a interface apenas!

    @Autowired
    public TarefasController(TarefaDao dao) {
        this.dao = dao;
    }

    //métodos omitidos
}

```

Agora a vantagem dessa abordagem é que podemos usar uma outra implementação da interface TarefaDao sem alterar a classe TarefasController , no nosso caso vamos usar JpaTarefaDao :

```

@Repository
public class JpaTarefaDao implements TarefaDao{

    @PersistenceContext
    EntityManager manager;

    //sem construtor

    //métodos omitidos
}

```

Repare que o DAO também vai implementar a interface TarefaDao . Assim temos duas implementações da mesma interface:



Como o Spring não sabe qual das duas implementações deve ser utilizada é preciso qualificar a dependência:

```

@Controller
public class TarefasController {

    private TarefaDao dao; //usa apenas a interface!

    @Autowired
    @Qualifier("jpaTarefaDao")
    public TarefasController(TarefaDao dao) {
        this.dao = dao;
    }
}

```

```
//métodos omitidos  
}
```

Dessa maneira o Spring injetará o `JpaTarefaDao`.

16.5 GERENCIANDO A TRANSAÇÃO

Para o suporte à transação ser ativado precisamos fazer duas configurações no XML do Spring. A primeira é habilitar o **gerenciador de transação** (`TransactionManager`). Porém, como o Spring pode controlar JPA, Hibernate e outros, devemos configurar o gerenciador exatamente para uma dessas tecnologias. No caso do JPA, a única dependência que o `JpaTransactionManager` precisa é uma `entityManagerFactory`:

```
<bean id="transactionManager"  
      class="org.springframework.orm.jpa.JpaTransactionManager">  
    <property name="entityManagerFactory" ref="entityManagerFactory"/>  
</bean>
```

`entityManagerFactory` é o nome do *Bean* configurado anteriormente:

A segunda parte é avisar que o controle de transações será feito via anotação, parecido com a forma de habilitar o uso de anotações para o Spring MVC.

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

Por fim, podemos usar o gerenciamento da transação dentro das nossas classes. Aqui fica muito simples, é só usar a anotação `@Transactional` no método que precisa de uma transação, por exemplo no método `adiciona` da classe `TarefasController`:

```
@Transactional  
@RequestMapping("adicionaTarefa")  
public String adiciona(@Valid Tarefa tarefa, BindingResult result) {  
  
    if(result.hasFieldErrors("descricao")) {  
        return "tarefa/formulario";  
    }  
  
    dao.adiciona(tarefa);  
    return "redirect:listaTarefas";  
}
```

A mesma anotação também pode ser utilizada na classe. Isso significa que todos os métodos da classe serão executados dentro de uma transação:

```
@Transactional  
@Controller  
public class TarefasController {
```

Repare aqui a beleza da inversão de controle. Não há necessidade de chamar `begin()`, `commit()` ou `rollback()`, basta usar `@Transactional` e o Spring assume a responsabilidade em gerenciar a transação.

Há um problema ainda, usando o gerenciamento de transação pelo Spring exige a presença do construtor padrão sem parâmetros. Vamos trocar aqui também o ponto de injeção do construtor para o atributo. A classe completa fica como:

```
package br.com.caelum.tarefas.controller;

//imports

@Controller
@Transactional
public class TarefasController {

    @Autowired
    TarefaDao dao;

    @RequestMapping("novaTarefa")
    public String form() {
        return "tarefa/formulario";
    }

    @RequestMapping("adicionaTarefa")
    public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

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

        dao.adiciona(tarefa);
        return "tarefa/adicionada";
    }

    @RequestMapping("listaTarefas")
    public String lista(Model model) {
        model.addAttribute("tarefas", dao.lista());
        return "tarefa/lista";
    }

    @RequestMapping("removeTarefa")
    public String remove(Tarefa tarefa) {
        dao.remove(tarefa);
        return "redirect:listaTarefas";
    }

    @RequestMapping("mostraTarefa")
    public String mostra(Long id, Model model) {
        model.addAttribute("tarefa", dao.buscaPorId(id));
        return "tarefa/mostra";
    }

    @RequestMapping("alteraTarefa")
    public String altera(Tarefa tarefa) {
        dao.altera(tarefa);
        return "redirect:listaTarefas";
    }

    @RequestMapping("finalizaTarefa")
    public String finaliza(Long id, Model model) {
        dao.finaliza(id);
        model.addAttribute("tarefa", dao.buscaPorId(id));
        return "tarefa/finalizada";
    }
}
```

}

16.6 EXERCÍCIOS: INTEGRANDO JPA COM SPRING

1. Vamos usar JPA através do Spring. Para isso é preciso copiar os JAR seguintes:

- aopalliance.jar
- spring-orm-4.x.x.RELEASE.jar
- spring-tx-4.x.x.RELEASE.jar

Para isso:

- Vá ao Desktop, e entre no diretório Caelum/21/jars-jpa/spring4 .
 - Copie os JARs (CTRL+C) e cole-o (CTRL+V) dentro de workspace/fj21-tarefas/WebContent/WEB-INF/lib
2. No projeto fj21-tarefas , vá a pasta WebContent/WEB-INF e abra o arquivo spring-context.xml .

Nele é preciso declarar o entityManagerFactory e o gerenciador de transações.

Você pode copiar essa parte do XML do arquivo Caelum/21/jars-jpa/spring4/spring-jpa.xml.txt . Copie o conteúdo do arquivo e cole dentro do 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>

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

<tx:annotation-driven/>
```

Com essas declarações, Spring gerencia a EntityManagerFactory e habilita o gerenciamento de transações.

3. O próximo passo é criar a interface TarefaDao . Crie uma nova interface dentro do package br.com.caelum.tarefas.dao :

```
package br.com.caelum.tarefas.dao;

// imports omitidos
```

```

public interface TarefaDao {

    Tarefa buscaPorId(Long id);
    List<Tarefa> lista();
    void adiciona(Tarefa t);
    void altera(Tarefa t);
    void remove(Tarefa t);
    void finaliza(Long id);
}

```

4. Crie uma classe `JpaTarefaDao` que recebe o `EntityManager`. Implemente a interface `TarefaDao` com todos os métodos.

```

package br.com.caelum.tarefas.dao;

// imports omitidos

@Repository
public class JpaTarefaDao implements TarefaDao{

    @PersistenceContext
    EntityManager manager;

    //sem construtor

    public void adiciona(Tarefa tarefa) {
        manager.persist(tarefa);
    }

    public void altera(Tarefa tarefa) {
        manager.merge(tarefa);
    }

    public List<Tarefa> lista() {
        return manager.createQuery("select t from Tarefa t")
            .getResultList();
    }

    public Tarefa buscaPorId(Long id) {
        return manager.find(Tarefa.class, id);
    }

    public void remove(Tarefa tarefa) {
        Tarefa tarefaARemover = buscaPorId(tarefa.getId());
        manager.remove(tarefaARemover);
    }

    public void finaliza(Long id) {
        Tarefa tarefa = buscaPorId(id);
        tarefa.setFinalizado(true);
        tarefa.setDataFinalizacao(Calendar.getInstance());
    }
}

```

5. Altere a classe `TarefasController`, use a interface `TarefaDao` apenas. Assim a classe `TarefasController` fica desacoplado da implementação. Não esquece de apagar o construtor:

```

@Controller
public class TarefasController {

    @Autowired

```

```
TarefaDao dao; //usa apenas a interface!  
  
//sem construtor  
  
//métodos omitidos, sem mudança  
}
```

6. Por fim, vamos habilitar o gerenciamento de transação para qualquer método da classe `TarefasController`.

Abra a classe e use a anotação `@Transactional` em cima da classe:

```
@Transactional  
@Controller  
public class TarefasController {
```

7. Reinicie o Tomcat e acesse a aplicação de tarefas em <http://localhost:8080/fj21-tarefas/listaTarefas>.

APÊNDICE - VRAPTOR3 E PRODUTIVIDADE NA WEB

"Aquele que castiga quando está irritado, não corrige, vinga-se" -- Michel de Montaigne

Neste capítulo, você aprenderá:

- O que é Inversão de Controle, Injeção de Dependências e *Convention over Configuration*;
- Como utilizar um framework MVC baseado em tais ideias;
- Como abstrair a camada HTTP da sua lógica de negócios;
- Como *não* utilizar arquivos XML para configuração da sua aplicação;
- A usar o framework MVC *VRaptor 3*.

17.1 MOTIVAÇÃO: EVITANDO APIs COMPLICADAS

Vamos lembrar como fica um código utilizando um controlador MVC simples para acessar os parâmetros do request, enviados pelo cliente.

É fácil notar como as classes, interfaces e apetrechos daquele controlador infectam o nosso código e surge a necessidade de conhecer a API de servlets a fundo. O código a seguir mostra uma Action do Struts 1 que utiliza um DAO para incluir um contato no banco de dados.

```
public class AdicionaContato implements Action {
    public String executa(HttpServletRequest req,
                          HttpServletResponse res) throws Exception{
        Contato contato = new Contato();
        contato.setNome(req.getParameter("nome"));
        contato.setEndereco(req.getParameter("endereco"));
        contato.setEmail(req.getParameter("email"));

        ContatoDao dao = new ContatoDao();
        dao.adiciona(contato);

        return "/ok.jsp";
    }
}
```

Baseado no código acima, percebemos que estamos fortemente atrelados a `HttpServletRequest` e seu método `getParameter`. Fora isso, usamos diversas classes estranhas ao nosso projeto: `Action`, `HttpServletRequest` e `HttpServletResponse`. Se estivéssemos controlando melhor a conexão, seria

necessário importar `Connection` também! Nenhuma dessas classes e interfaces citadas faz parte do nosso projeto! Não é o código que modela minha lógica!

É muito chato, e nada prático, repetir isso em toda a sua aplicação. Sendo assim, visando facilitar esse tipo de trabalho, vimos que o *Struts Action*, por exemplo, utiliza alguns recursos que facilitam o nosso trabalho:

```
public class AdicionaContato extends Action {  
  
    public ActionForward execute(ActionMapping map, ActionForm form,  
        HttpServletRequest req, HttpServletResponse res)  
        throws Exception {  
  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDao dao = new ContatoDao();  
        dao.adiciona(contato);  
  
        return map.findForward("ok");  
    }  
}
```

Mas, mesmo assim, imagine os limites de tal código:

- Você **não** pode receber mais de um `action form`;
- Sua classe **deve** estender `ActionForm`. Se você queria estender outra, azar;
- Você **deve** receber todos esses argumentos que não foi você quem criou;
- Você **deve** retornar esse tipo que não foi você quem criou;
- Você **deve** trabalhar com Strings ou tipos muito pobres. O sistema de conversão é complexo para iniciantes;
- O código fica muito alienado: ele é escrito de tal forma que o programador precisa agradar o framework e não o framework agradar o programador;
- Você acaba criando classes repetidas: deve criar dois beans repetidos ou parecidos, ou ainda escrever muito código xml para substituir um deles.

Dado esses problemas, surgiram diversos outros frameworks, inclusive diversos patterns novos, entre eles, Injeção de Dependências (*Dependency Injection*), Inversão de Controle (*Inversion of Control - IoC*) e o hábito de **preferir** ter convenções em vez de configuração (*Convention over Configuration - CoC*).

Imagine que possamos deixar de estender `Action`:

```
public class AdicionaContato {  
    public ActionForward execute(ActionMapping map, ActionForm form,  
        HttpServletRequest req, HttpServletResponse res)  
        throws Exception {  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDao dao = new ContatoDao();  
        dao.adiciona(contato);  
  
        return map.findForward("ok");  
    }  
}
```

```
}
```

Nesse momento estamos livres para mudar nosso método `execute`. Desejamos que ele se chame `adiciona`, e não um nome genérico como `execute`, sem contar que não precisamos de todos aqueles quatro argumentos. Afinal, não utilizamos o `request` e `response`:

```
public class AdicionaContato {  
    public ActionForward adiciona(ActionMapping map,  
        ActionForm form) throws Exception {  
  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDao dao = new ContatoDao();  
        dao.adiciona(contato);  
  
        return map.findForward("ok");  
    }  
}
```

Aos poucos, o código vai ficando mais simples. Repare que na maioria das vezes que retornamos o `ActionForward`, o retorno é "ok". Será que sempre temos que fazer isso? Não seria mais fácil não retornar valor algum, já que sempre retornamos o mesmo valor? Portanto, vamos remover esse valor de retorno ao invés de usar esse estranho `ActionMapping`:

```
public class AdicionaContato {  
    public void adiciona(ActionForm form) throws Exception {  
  
        Contato contato = ((ContatoForm) form).getContato();  
  
        ContatoDao dao = new ContatoDao();  
        dao.adiciona(contato);  
  
    }  
}
```

Por fim, em vez de criar uma classe que estende `ActionForm`, ou configurar toneladas de XML, desejamos receber um `Contato` como parâmetro do método.

Portanto nada mais natural que o parâmetro seja `Contato`, e não `ContatoForm`.

```
public class AdicionaContato {  
  
    public void adiciona(Contato contato) throws Exception {  
  
        ContatoDao dao = new ContatoDao();  
        dao.adiciona(contato);  
  
    }  
}
```

O resultado é um código bem mais legível, e um controlador menos intrusivo no seu código: nesse caso nem usamos a API de servlets!

17.2 VANTAGENS DE UM CÓDIGO INDEPENDENTE DE REQUEST E

RESPONSE

Você desconecta o seu programa da camada web, criando ações ou comandos que não trabalham com `request` e `response`.

Note que, enquanto utilizávamos a lógica daquela maneira, o mesmo servia de adaptador para a web. Não precisamos mais desse adaptador, nossa própria lógica que é uma classe Java comum, pode servir para a web ou a qualquer outro propósito.

Além disso, você recebe todos os objetos que precisa para trabalhar, não se preocupando em buscá-los. Isso é chamado de injeção de dependências. Por exemplo, se você precisa do usuário logado no sistema e, supondo que ele seja do tipo `Funcionario`, você pode criar um construtor que requer tal objeto:

```
public class AdicionaContato {  
  
    public AdicionaContato(Funcionario funcionario) {  
        // o parametro é o funcionario logado no sistema  
    }  
  
    public void adiciona(Contato contato) throws Exception {  
  
        ContatoDao dao = new ContatoDao();  
        dao.adiciona(contato);  
  
    }  
}
```

17.3 VRAPTOR 3

Tudo o que faremos neste capítulo está baseado no framework opensource **Vraptor 3**. Sua documentação pode ser encontrada em:

<http://www.vraptor.com.br/>

Iniciativa brasileira, o VRaptor foi criado inicialmente para o desenvolvimento de projetos internos do Instituto de Matemática e Estatística da USP, pelos então alunos do curso de ciência da computação.

O site do GUJ (www.guj.com.br), fundado em 2002 pelos mesmos criadores do VRaptor, teve sua versão nova escrita em VRaptor 2, e hoje em dia roda na versão 3. Este framework é utilizado em projetos open source e por várias empresas no Brasil e também pelo mundo.

Consulte o site para tutoriais, depoimentos, screencasts e até palestras filmadas a respeito do framework.

Para aprendermos mais do VRaptor, vamos utilizar seus recursos básicos em um pequeno projeto, e vale notar a simplicidade com qual o código vai ficar.

17.4 A CLASSE DE MODELO

O projeto já vem com uma classe de modelo pronta, chamada `Produto`, que utilizaremos em nossos exemplos, e é uma entidade do Hibernate:

```
@Entity
public class Produto {

    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    private Double preco;

    @Temporal(TemporalType.DATE)
    private Calendar dataInicioVenda;
    // getters e setters

}
```

A classe `ProdutoDAO` também já existe e utiliza o hibernate para acessar um banco de dados (mysql).

```
public class ProdutoDAO {

    private Session session;

    public ProdutoDao() {
        this.session = new HibernateUtil().getSession();
    }

    public void adiciona(Produto p) {
        Transaction tx = session.beginTransaction();
        session.save(p);
        tx.commit();
    }

    public void atualiza(Produto p) {
        Transaction tx = session.beginTransaction();
        session.update(p);
        tx.commit();
    }

    public void remove(Produto p) {
        Transaction tx = session.beginTransaction();
        session.delete(p);
        tx.commit();
    }

    @SuppressWarnings("unchecked")
    public List<Produto> lista() {
        return session.createCriteria(Produto.class).list();
    }
}
```

O foco desse capítulo não está em como configurar o Hibernate ou em boas práticas da camada de persistência portanto o código do DAO pode não ser o ideal por utilizar uma transação para cada

chamada de método, mas é o ideal para nosso exemplo.

17.5 MINHA PRIMEIRA LÓGICA DE NEGÓCIOS

Vamos escrever uma classe que adiciona um `Produto` a um banco de dados através do DAO e do uso do VRaptor 3:

```
@Resource  
public class ProdutoController {  
  
    public void adiciona(Produto produto) {  
        new ProdutoDao().adiciona(produto);  
    }  
}
```

Pronto! É assim que fica uma ação de adicionar produto utilizando esse controlador (e, futuramente, vamos melhorar mais ainda).

E se surgir a necessidade de criar um método `atualiza`? Poderíamos reutilizar a mesma classe para os dois métodos:

```
@Resource  
public class ProdutoController {  
  
    // a ação adiciona  
    public void adiciona(Produto produto) {  
        new ProdutoDao().adiciona(produto);  
    }  
  
    // a ação atualiza  
    public void atualiza(Produto produto) {  
        new ProdutoDao().atualiza(produto);  
    }  
}
```

O próprio controlador se encarrega de preencher o produto para chamar nosso método. O que estamos fazendo através da anotação `@Resource` é dizer para o Vraptor disponibilizar essa classe para ser instanciada e exposta para a web. Dessa forma será disponibilizado uma URI para que seja possível invocar a lógica desejada, seja de adição de contato, seja de atualização.

Não existe nenhum XML do VRaptor que seja de preenchimento obrigatório. Só essa classe já é suficiente.

E o JSP com o formulário? Para fazer com que a lógica seja invocada, basta configurarmos corretamente os campos do formulário no nosso código html. Não há segredo algum.

Note que nosso controller se chama `ProdutoController` e o nome do método que desejamos invocar se chama `adiciona`. Com isso, o VRaptor invocará este método através da URI `/produto/adiciona`. Repare que em nenhum momento você configurou esse endereço. Esse é um dos pontos no qual o VRaptor usa diversas convenções dele, ao invés de esperar que você faça alguma

configuração obrigatória.

17.6 REDIRECIONANDO APÓS A INCLUSÃO

Precisamos criar uma página que mostre uma mensagem de sucesso, aproveitamos e confirmamos a inclusão mostrando os dados que foram incluídos:

```
<html>
    Seu produto foi adicionado com sucesso!<br/>
</html>
```

Mas qual o nome desse arquivo? Uma vez que o nome do controller é `produto`, o nome da lógica é `adiciona`, o nome de seu arquivo de saída deve ser: `WebContent/WEB-INF/jsp/produto/adiciona.jsp`.

Você pode alterar o redirecionamento padrão da sua lógica, enviando o usuário para um outro jsp, para isso basta receber no construtor do seu controller um objeto do tipo `Result`. Esse objeto será passado para o seu controller através de Injeção de Dependências.

Dessa forma, um exemplo de redirecionar para outro jsp após a execução seria:

```
@Resource
public class ProdutoController {
    private Result result;

    public ProdutoController(Result result) {
        this.result = result;
    }

    // a ação adiciona
    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
        result.forwardTo("/WEB-INF/jsp/outroLugar.jsp");
    }
}
```

E a criação do arquivo `WEB-INF/jsp/outroLugar.jsp`. Dessa forma, você pode condicionar o retorno, de acordo com o que acontecer dentro do método que realizará toda a sua lógica, controlando para onde o usuário será redirecionado através de if's.

Além de termos a possibilidade de redirecionar para outro JSP, também podemos redirecionar para uma outra lógica. Novamente podemos fazer isso através do objeto `Result`. Basta dizermos que vamos enviar o usuário para outra lógica e indicarmos de qual controller será essa lógica e também qual método deverá ser chamado.

```
public void remove(Produto produto) {
    dao.remove(produto);
    result.redirectTo(ProdutoController.class).lista();
}
```

Isso fará com que logo após a lógica de remoção seja executada, o usuário execute a lógica de

listagem que desenvolveremos a seguir.

17.7 CRIANDO O FORMULÁRIO

Poderíamos criar nossos formulários como JSPs diretos dentro de `WebContent`, e acessarmos os mesmos diretamente no navegador, mas isso não é uma prática aconselhada, pois, futuramente podemos querer executar alguma lógica antes desse formulário e para isso ele teria que ser uma lógica do `VRaptor` e provavelmente teríamos que mudar sua URL, quebrando links dos usuários que já os possuíam gravados, por exemplo.

Vamos tomar uma abordagem diferente, pensando desde o começo que futuramente precisaremos de uma lógica executando antes do formulário, vamos criar uma lógica vazia, que nada mais fará do que repassar a execução ao formulário através da convenção do `VRaptor`.

```
@Resource  
public class ProdutoController {  
    public void formulario() {  
    }  
}
```

Podemos acessar o nosso formulário pelo navegador através da URL: `/produto/formulario`.

Já os parâmetros devem ser enviados com o nome do parâmetro que desejamos preencher, no nosso caso, `produto`, pois, é o nome do parâmetro do método `adiciona`:

```
<html>  
    <form action="produto/adiciona">  
        Nome: <input name="produto.nome"/><br/>  
        Descricao: <input name="produto.descricao"/><br/>  
        Preço: <input name="produto.preco"/><br/>  
        Data de início de venda: <input name="produto.dataInicioVenda"/><br/>  
        <input type="submit"/>  
    </form>  
</html>
```

The screenshot shows a web browser window with the URL `http://localhost:...oduto/formulario` in the address bar. Below the address bar is a toolbar with a refresh icon and a plus sign. The main content area contains a form with four labeled fields: "Nome:", "Descricao:", "Preço:", and "Data de início da venda:". Each label is followed by a text input field. At the bottom of the form is a button labeled "Submit Query".

17.8 A LISTA DE PRODUTOS

O próximo passo será criar a listagem de todos os produtos do sistema. Nossa lógica de negócios, mais uma vez, possui somente aquilo que precisa:

```
@Resource  
public class ProdutoController {  
  
    public void lista() {  
        new ProdutoDao().lista();  
    }  
  
}
```

Mas dessa vez precisamos disponibilizar esta lista de produtos para a nossa camada de visualização. Pensando em código java, qual é a forma mais simples que temos de retornar um valor (objeto) para alguém? Com um simples `return`, logo, nosso método `lista` poderia retornar a própria lista que foi devolvida pelo DAO, da seguinte forma:

```
public List<Produto> lista() {  
    return new ProdutoDao().lista();  
}
```

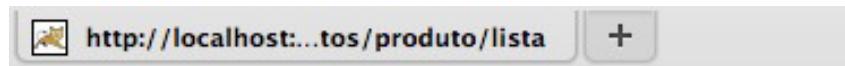
Dessa forma, o VRaptor disponibilizará na view um objeto chamado `produtoList`, que você possui acesso via *Expression Language*.

Repare que nossa classe pode ser testada facilmente, basta instanciar o bean `ProdutoController`, chamar o método `lista` e verificar se ele retorna ou não a lista que esperávamos. Todas essas convenções evitando o uso de configurações ajudam bastante no momento de criar testes de unidade para o seu sistema.

Por fim, vamos criar o arquivo `lista.jsp` no diretório `WebContent/WEB-INF/jsp/produto/` utilizando a taglib core da JSTL:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>  
  
<h1>Produtos</h1>  
<table>  
    <c:forEach var="produto" items="${produtoList}">  
        <tr>  
            <td>${produto.nome}</td>  
            <td>${produto.preco}</td>  
            <td>${produto.descricao}</td>  
            <td><fmt:formatDate pattern="dd/MM/yyyy"  
                value="${produto.dataInicioVenda.time}" />  
            </td>  
        </tr>  
    </c:forEach>  
</table>
```

E chame o endereço: `http://localhost:8080/controle-produtos/produto/lista`, o resultado deve ser algo parecido com a imagem abaixo:



Produtos

Abacaxi 13.0 Abacaxi verde 04/10/2009 [Remover](#)

Pera 7.0 Pera verde 24/10/2009 [Remover](#)

17.9 EXERCÍCIOS

Vamos criar um novo projeto do Eclipse, usando um projeto já começado com o VRaptor.

1. Crie um novo **Dynamic Web Project** chamado **controle-produtos**.
2. Clique da direita no nome do projeto **controle-produtos** e vá em **Import > Archive File**. Importe o arquivo **Desktop/caelum/21/controle-produtos.zip**.
3. Gere a tabela Produto. Procure a classe `GeraTabela` e execute o método `main`.
4. Associe o projeto com o Tomcat e accesse no seu navegador a URL: <http://localhost:8080/controle-produtos>
5. Vamos criar a parte de listagem dos produtos.
 - Crie a classe `ProdutoController` no pacote `br.com.caelum.produtos.controller`, com o método para fazer a listagem (**Não se esqueça de anotar a classe com `@Resource`**):

```
@Resource  
public class ProdutoController {  
  
    public List<Produto> lista() {  
        return new ProdutoDao().lista();  
    }  
}
```

- Crie o arquivo `lista.jsp` no diretório `WebContent/WEB-INF/jsp/produto` (crie pasta e jsp)

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>  
  
<h1>Produtos</h1>  
<table>  
    <c:forEach var="produto" items="${produtoList}">  
        <tr>  
            <td>${produto.nome}</td>  
            <td>${produto.preco}</td>  
            <td>${produto.descricao}</td>  
            <td>  
                <fmt:formatDate pattern="dd/MM/yyyy"  
                value ="${produto.dataInicioVenda.time}" />  
            </td>  
        </tr>  
    </c:forEach>  
</table>
```

```

        </td>
    </tr>
</c:forEach>
</table>
```

- Teste a sua lógica: <http://localhost:8080/controle-produtos/produto/lista> .

6. Vamos criar a funcionalidade de adicionar produtos.

- Na classe `ProdutoController` crie o método `adiciona` :

```

@Resource
public class ProdutoController {

    // a ação adiciona
    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
    }

}
```

- Após o produto ser gravado, devemos retornar para a listagem dos produtos. Para isso vamos adicionar o redirecionamento no método `adiciona` . Para fazermos esse redirecionamento vamos precisar receber um objeto do tipo `Result` no construtor do nosso `ProdutoController` :

```

@Resource
public class ProdutoController {
    private Result result;

    public ProdutoController(Result result) {
        this.result = result;
    }

    //método para fazer a listagem

    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
        result.redirectTo(ProdutoController.class).lista();
    }

}
```

- Vamos criar a lógica para o formulário

```

@Resource
public class ProdutoController {

    public void formulario() {
    }

}
```

- Crie o arquivo `formulario.jsp` no diretório `WebContent/WEB-INF/jsp/produto`

```

<%@ taglib tagdir="/WEB-INF/tags" prefix="caelum" %>
<%@ taglib
    uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
<html>
```

```

<head>
    <script type="text/javascript"
        src="

```

- Se achar conveniente, crie um arquivo para o cabecalho faça-o importar as bibliotecas em *Javascript* e o *CSS* e inclua-o no `formulario.jsp`
- Você importou alguma classe do pacote `javax.servlet`? É importante percebermos que não estamos atrelados com nenhuma API estranha. Apenas escrevemos código Java normal.
- Teste o seu formulário: `http://localhost:8080/controle-produtos/produto/formulario`. Adicione alguns produtos diferentes.

7. Vamos fazer a exclusão de produtos.

- Vamos primeiramente criar uma nova coluna na nossa tabela no `lista.jsp`, adicionando para cada produto um link para fazermos a exclusão:

```

<c:forEach var="produto" items="${produtoList}">
    <tr>
        <td>${produto.nome}</td>
        <td>${produto.preco}</td>
        <td>${produto.descricao}</td>
        <td>
            <fmt:formatDate pattern="dd/MM/yyyy"
                value="${produto.dataInicioVenda.time}"/>
        </td>
        <td>
            <a href="

```

- Vamos criar a nossa nova lógica para exclusão no `ProdutoController` que redirecionará para a lógica de lista após a remoção do produto:

```

public void remove(Produto produto) {
    new ProdutoDao().remove(produto);
    result.redirectTo(ProdutoController.class).lista();
}

```

- Acesse a lista de produtos em <http://localhost:8080/controle-produtos/produto/lista> e remova alguns produtos.

8. (Opcional) Faça a funcionalidade de alteração do produto da forma que achar conveniente

17.10 APROFUNDANDO EM INJEÇÃO DE DEPENDÊNCIAS E INVERSÃO DE CONTROLE

O VRaptor utiliza bastante Injeção de Dependências e também permite que nós utilizemos em nossos `Controller`s. O grande ganho que temos com isso é que nosso código fica muito menos acoplado com outras bibliotecas, aumentando a testabilidade da nossa aplicação através de testes de unidade, assunto coberto no curso **FJ-22**.

Além disso, outra vantagem que sentimos ao utilizar Injeção de Dependências é a legibilidade de nosso código. Nós sabemos o que ele recebe, mas lendo o nosso código, não precisamos saber de qual lugar essa dependência está vindo, apenas como ela será utilizada. O fato de não sabermos mais como a nossa dependência será criada é o que chamamos de Inversão de Controle.

O código que desenvolvemos no `ProdutoController` possui um exemplo muito bom de código acoplado. Repare que todas as nossas lógicas (`adiciona` , `remove` , `lista` etc) sabem como o `ProdutoDao` tem que ser criado, ou seja, que precisamos instanciá-lo sem passar nenhum argumento pro seu construtor. Se um dia mudássemos o construtor para receber algum parâmetro, teríamos que mudar todos os métodos de nossa lógica e isso não é algo que queremos nos preocupar no dia a dia.

17.11 INJEÇÃO DE DEPENDÊNCIAS COM O VRAPTOR

Podemos desacoplar os nossos métodos de `ProdutoController` fazendo com que o `Controller` apenas receba em seu construtor uma instância de `ProdutoDao` . E quando o VRaptor precisar criar o `ProdutoController` a cada requisição, ele também criará uma instância do `ProdutoDao` para passar ao `Controller` .

```

public class ProdutoController {
    private Result result;
    private ProdutoDao produtoDao;

    public ProdutoController(Result result, ProdutoDao produtoDao) {
        this.result = result;
        this.produtoDao = produtoDao;
    }

    //métodos para adicionar, excluir e listar produtos
}

```

No entanto, se executarmos qualquer lógica nossa aplicação vai parar de funcionar. Isso porque precisamos dizer para o VRaptor que ProdutoDao é uma classe que ele vai gerenciar.

Podemos fazer isso através da anotação `@Component` no nosso `ProdutoDao`.

Aqui, em um caso mais usual, `ProdutoDao` seria uma interface, e nosso componente seria uma classe concreta como, por exemplo, `HibernateProdutoDao`. O VRaptor vai saber descobrir que deve injetar um `HibernateProdutoDao` para quem precisa um `ProdutoDao`. Dessa forma desacoplamos ainda mais nosso código do controller, podendo passar mocks para ele quando formos testá-lo, como veremos no curso Laboratório Java.

```
@Component  
public class ProdutoDao {  
  
    //construtor e métodos do Dao  
}
```

17.12 ESCOPOS DOS COMPONENTES

Por padrão, o VRaptor criará um `ProdutoDao` por requisição. Mas nem sempre é isso que queremos. Portanto, podemos mudar esse comportamento através de anotações que determinam em qual escopo o nosso componente ficará:

- `@RequestScoped` - o componente será o mesmo durante a requisição
- `@SessionScoped` - o componente será o mesmo durante a sessão do usuário
- `@ApplicationScoped` - o componente será o mesmo para toda a aplicação
- `@PrototypeScoped` - o componente será instanciado sempre que requisitado

Para definirmos um escopo, o nosso `ContatoDao` poderia ficar da seguinte forma:

```
@Component  
@RequestScoped  
public class ProdutoDao {  
  
    //construtor e métodos do Dao  
}
```

COMO OS COMPONENTES SÃO GERENCIADOS PELO VRAPTOR?

Uma das boas ideias do VRaptor é o de não reinventar a roda e reutilizar funcionalidades e frameworks já existentes quando possível.

Uma das partes na qual o VRaptor se utiliza de frameworks externos é na parte de Injeção de Dependências, na qual ele utiliza o framework Spring que possui um container de Injeção de Dependências.

Aprendemos Spring a fundo no curso FJ-27 que cobre em detalhes o framework.

17.13 EXERCÍCIOS: USANDO INJEÇÃO DE DEPENDÊNCIAS PARA O DAO

1. Vamos diminuir o acoplamento do `ProdutoController` com o `ProdutoDao` recebendo-o no construtor.

- Precisamos fazer com que o `ProdutoDao` seja um componente gerenciado pelo VRaptor. Para isso, vamos anotá-lo com `@Component` e `@RequestScoped`.

```
@Component  
@RequestScoped  
public class ProdutoDao {  
  
    //construtor e métodos do Dao  
}
```

- Basta indicarmos que queremos receber `ProdutoDao` no construtor do `ProdutoController` e utilizarmos o `DAO` recebido para fazermos nossas operações com o banco de dados. Não se esqueça de alterar seus métodos.

```
@Resource  
public class ProdutoController {  
    private Result result;  
    private ProdutoDao produtoDao;  
  
    public ProdutoController(Result result, ProdutoDao produtoDao) {  
        this.result = result;  
        this.produtoDao = produtoDao;  
    }  
  
    public List<Produto> lista() {  
        return produtoDao.lista();  
    }  
  
    public void adiciona(Produto produto) {  
        produtoDao.adiciona(produto);  
        //redirecionamento  
    }  
}
```

```

        public void remove(Produto produto) {
            produtoDao.remove(produto);
            //redirecionamento
        }
    }
}

```

- Reinicie sua aplicação e verifique que tudo continua funcionando normalmente.

17.14 ADICIONANDO SEGURANÇA EM NOSSA APLICAÇÃO

Nossa aplicação de controle de produtos permite que qualquer pessoa modifique os dados de produtos. Isso não é algo bom, pois, pessoas sem as devidas permissões poderão acessar essa funcionalidade e guardar dados inconsistentes no banco.

Precisamos fazer com que os usuários façam login em nossa aplicação e caso ele esteja logado, permitiremos acesso às funcionalidades.

Para construir a funcionalidade de autenticação, precisamos antes ter o modelo de `Usuario` e o seu respectivo `DAO` com o método para buscar o `Usuario` através do login e senha.

```

@Entity
public class Usuario {
    @Id @GeneratedValue
    private Long id;

    private String nome;

    private String login;

    private String senha;

    //getters e setters
}

@Component
@RequestScoped
public class UsuarioDao {
    private Session session;

    public UsuarioDao() {
        this.session = new HibernateUtil().getSession();
    }

    public Usuario buscaUsuarioPorLoginESenha(Usuario usuario) {
        Query query = this.session.
            createQuery("from Usuario where " +
            "login = :pLogin and senha = :pSenha");
        query.setParameter("pLogin", usuario.getLogin());
        query.setParameter("pSenha", usuario.getSenha());
        return (Usuario) query.uniqueResult();
    }
}

```

Como já possuímos o modelo de usuário, precisamos guardar o usuário na sessão.

Já aprendemos que podemos criar um componente (`@Component`) que fica guardado na sessão do usuário. Portanto, vamos utilizar essa facilidade do VRaptor .

```
@Component
@SessionScoped
public class UsuarioLogado {
    private Usuario usuarioLogado;

    public void efetuaLogin(Usuario usuario) {
        this.usuarioLogado = usuario;
    }

    //getter pro usuarioLogado
}
```

Basta construirmos o Controller que fará o login do nosso usuário. Vamos criar uma classe chamada `LoginController` que receberá via construtor um `UsuarioLogado` e o `UsuarioDao` . Após a verificação de que o usuário informado está cadastrado o guardaremos dentro do nosso componente `UsuarioLogado` . Se o usuário existir, a requisição será redirecionada para a listagem dos produtos.

```
@Controller
public class LoginController {
    private UsuarioDao usuarioDao;
    private UsuarioLogado usuarioLogado;
    private Result result;

    public LoginController(UsuarioDao usuarioDao,
                          UsuarioLogado usuarioLogado, Result result){
        this.usuarioDao = usuarioDao;
        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void autentica(Usuario usuario) {
        Usuario autenticado = usuarioDao
            .buscaUsuarioPorLoginESenha(usuario);
        if(autenticado != null) {
            usuarioLogado.efetuaLogin(autenticado);
            result.redirectTo(ProdutoController.class).lista();
        }
    }
}
```

Basta criarmos o formulário para fazermos o login. Vamos utilizar a mesma estratégia do formulário para cadastro de produtos, criando um método no nosso Controller que redirecionará para o formulário.

```
@Controller
public class LoginController {
    //atributos, construtor e métodos para efetuar o login

    public void formulario() {
    }
}
```

E o JSP contendo o formulário, que estará em `WEB-INF/jsp/login` e se chamará `formulario.jsp` :

```

<html>
  <body>
    <h2>Login no Controle de Produtos</h2>
    <form action="login/autentica">
      Login: <input type="text" name="usuario.login" /><br />
      Senha: <input type="password" name="usuario.senha" />
      <input type="submit" value="Autenticar" />
    </form>
  </body>
</html>

```

Ainda precisamos fazer com que se o login seja inválido, o usuário volte para a tela de login:

```

public void autentica(Usuario usuario) {
  Usuario autenticado = dao.buscaUsuarioPorLoginESenha(usuario);
  if(autenticado != null) {
    usuarioLogado.efetuaLogin(autenticado);
    result.redirectTo(ProdutoController.class).lista();
    return;
  }
  result.redirectTo(LoginController.class).formulario();
}

```

Pronto, nossa funcionalidade de Login está pronta!

17.15 INTERCEPTANDO REQUISIÇÕES

Mas como garantir que o usuário está mesmo logado na nossa aplicação no momento em que ele tenta, por exemplo, acessar o formulário de gravação de produtos?

O que precisamos fazer é verificar, antes de qualquer lógica ser executada, se o usuário está guardado na sessão. Podemos fazer isso através de `Interceptor`s, que funcionam de forma parecida com os `Filter`s que aprendemos anteriormente. A vantagem é que os `Interceptor`s nos fornecem facilidades a mais que estão ligadas ao VRaptor, algo que a API de `Filter` não nos provê.

Para criarmos um `Interceptor` basta criarmos uma classe que implementa a interface `br.com.caelum.vraptor.Interceptor` e anotá-la com `@Intercepts`.

Ao implementarmos a interface, devemos escrever dois métodos: `intercept` e `accepts`.

- `intercept` : Possui o código que fará toda a lógica que desejamos executar antes e depois da requisição ser processada.
- `accepts` : Método que indica através de um retorno `boolean` quem deverá ser interceptado e quem não deverá ser interceptado.

O `Interceptor` como qualquer componente, pode receber em seu construtor outros componentes e no nosso caso ele precisará do `UsuarioLogado` para saber se existe alguém logado ou não.

Dessa forma, o nosso `Interceptor` terá o seguinte código:

```

@Intercepts
public class LoginInterceptor implements Interceptor {

```

```

private UsuarioLogado usuarioLogado;
private Result result;

public LoginInterceptor(UsuarioLogado usuarioLogado, Result result) {
    this.usuarioLogado = usuarioLogado;
    this.result = result;
}

public void intercept(InterceptorStack stack,
                      ResourceMethod method, Object instance)
                      throws InterceptionException {
    if(usuarioLogado.getUsuario() != null) {
        stack.next(method, instance);
    } else {
        result.redirectTo(LoginController.class).formulario();
    }
}

public boolean accepts(ResourceMethod method) {
    ResourceClass resource = method.getResource();

    return !resource.getType().isAssignableFrom(LoginController.class);
}
}

```

Pronto, temos nossa funcionalidade de autenticação e também a parte de autorização finalizadas.

17.16 EXERCÍCIOS: CONSTRUINDO A AUTENTICAÇÃO E A AUTORIZAÇÃO

- Vamos permitir que nossos usuários possam efetuar Login na aplicação.

- Crie o componente `UsuarioLogado` no pacote `br.com.caelum.produtos.component` com o seguinte código:

```

@Component
@SessionScoped
public class UsuarioLogado {
    private Usuario usuarioLogado;

    public void efetuaLogin(Usuario usuario) {
        this.usuarioLogado = usuario;
    }

    public Usuario getUsuario() {
        return this.usuarioLogado;
    }
}

```

- Faça com que `UsuarioDao` seja um componente anotando-o com `@Component` e indique que ele deverá estar no escopo de requisição (`@RequestScoped`).

```

@Component
@RequestScoped
public class UsuarioDao {

    //metodos e construtor

```

```
}
```

- Crie a classe `LoginController` dentro do pacote `br.com.caelum.produtos.controller`:

```
@Resource
public class LoginController {
    private UsuarioDao usuarioDao;
    private UsuarioLogado usuarioLogado;
    private Result result;

    public LoginController(UsuarioDao usuarioDao,
                          UsuarioLogado usuarioLogado, Result result) {
        this.usuarioDao = usuarioDao;
        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void autentica(Usuario usuario) {
        Usuario autenticado = usuarioDao
            .buscaUsuarioPorLoginESenha(usuario);
        if(autenticado != null) {
            usuarioLogado.efetuaLogin(autenticado);
            result.redirectTo(ProdutoController.class)
                .lista();
            return;
        }
        result.redirectTo(LoginController.class).formulario();
    }

    public void formulario() {
    }
}
```

- Vamos criar a tela para permitir com que os usuários se loguem na aplicação, crie um arquivo chamado `formulario.jsp` dentro de `WEB-INF/jsp/login` com o conteúdo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <body>
        <h2>Login no Controle de Produtos</h2>
        <form action="">
            Login: <input type="text" name="usuario.login" /><br />
            Senha: <input type="password" name="usuario.senha" />
            <input type="submit" value="Autenticar" />
        </form>
    </body>
</html>
```

- Vamos criar o interceptador para não deixar o usuário acessar as funcionalidades do nosso sistema sem ter logado antes. Crie a classe `LoginInterceptor` no pacote `br.com.caelum.produtos.interceptor`:

```
@Intercepts
public class LoginInterceptor implements Interceptor {

    private UsuarioLogado usuarioLogado;
    private Result result;

    public LoginInterceptor(UsuarioLogado usuarioLogado,
                           Result result) {
```

```

        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void intercept(InterceptorStack stack,
        ResourceMethod method, Object instance)
        throws InterceptionException {

        if(usuarioLogado.getUsuario() != null) {
            stack.next(method, instance);
        } else {
            result.redirectTo(LoginController.class).formulario();
        }
    }

    public boolean accepts(ResourceMethod method) {
        ResourceClass resource = method.getResource();

        return !resource.getType().isAssignableFrom(LoginController.class);
    }
}

```

- Tente acessar <http://localhost:8080/controle-produtos/produto/lista> e como você não efetuou o login, você é redirecionado para a devida tela de login.



- Efetue o seu login. Verifique o seu banco de dados para conseguir um login válido. Caso não exista ninguém cadastrado, insira um usuário no banco com o comando abaixo e tente efetuar o login:

```

insert into Usuario (nome, login, senha)
values ('Administrador', 'admin', 'admin123');

```

17.17 MELHORANDO A USABILIDADE DA NOSSA APLICAÇÃO

Sempre que clicamos no link *Remover* na listagem dos produtos uma requisição é enviada para o endereço `/produto/remove`, a exclusão é feita no banco de dados e em seguida **toda** a listagem é recriada novamente.

A requisição ser enviada e a exclusão ser feita no banco de dados são passos obrigatórios nesse processo, mas será que precisamos recriar toda a listagem outra vez?

Podemos destacar alguns pontos negativos nessa abordagem, por exemplo:

- Tráfego na rede: Recebemos como resposta todo o HTML para regerar a tela inteira. Não seria mais leve apenas removermos a linha da tabela que acabamos de excluir ao invés de recriarmos toda a tabela?
- Demora na resposta: Se a requisição demorar para gerar uma resposta, toda a navegação ficará comprometida. Dependendo do ponto aonde estiver a lentidão poderá até ficar uma tela em branco que não dirá nada ao usuário. Será que não é mais interessante para o usuário permanecer na mesma tela em que ele estava e colocar na tela uma mensagem indicando que está realizando a operação? Como a tela continuará aberta sempre, ele pode continuar sua navegação e uso normalmente.

17.18 PARA SABER MAIS: REQUISIÇÕES: SÍNCRONO X ASSÍNCRONO

Nós podemos enviar requisições em nossas aplicações web que não "travem" a navegação do usuário e mantenha a aplicação disponível para continuar sendo utilizada. As requisições tradicionais, com as quais estamos acostumados é o que chamamos de *Requisições Síncronas*, na qual o navegador ao enviar a requisição interrompe a navegação e reexibe toda a resposta devolvida pelo servidor.

Mas podemos utilizar um outro estilo, que são as *Requisições Assíncronas*. Nelas, quando a requisição é enviada, o navegador continua no estado em que estava antes, permitindo a navegação. Quando a resposta é dada pelo servidor é possível pegá-la e apenas editar algum pedaço da página que já estava exibida antes para mostrar um conteúdo novo (sem recarregar toda a página).

17.19 PARA SABER MAIS: AJAX

Podemos fazer requisições assíncronas através de uma técnica conhecida como **AJAX** (*Assynchronous Javascript and XML*). Essa técnica nada mais é do que utilizar a linguagem *Javascript* para que em determinados eventos da sua página, por exemplo, o clique de um botão, seja possível enviar as requisições de forma assíncrona para um determinado lugar, recuperar essa resposta e editar nossa página para alterarmos dinamicamente o seu conteúdo.

Um dos grandes problemas de **AJAX** quando a técnica surgiu é que era muito complicado utilizá-la, principalmente porque haviam incompatibilidades entre os diversos navegadores existentes. Muitos tratamentos tinham que ser feitos, para que as funcionalidades fossem compatíveis com os navegadores. E dessa forma, tínhamos códigos muito grandes e de difícil legibilidade.

Hoje em dia temos muito mais facilidade para trabalhar com **AJAX**, através de bibliotecas de *Javascript* que encapsulam toda sua complexidade nos fornecem uma API agradável para trabalhar, como por exemplo, **jQuery**, **YUI** (*Yahoo User Interface*), **ExtJS** e assim por diante.

17.20 ADICIONANDO AJAX NA NOSSA APLICAÇÃO

Aqui no curso utilizaremos o jQuery, mas tudo o que fizermos também poderá ser feito com as outras bibliotecas. Para detalhes de como fazer, consulte suas documentações.

Vamos colocar AJAX na remoção dos produtos. Ao clicarmos no link *Remover*, vamos disparar um evento que estará associado com uma função em Javascript, que utilizará o jQuery para enviar a requisição, pegar a resposta que será uma mensagem indicando que o produto foi removido e colocar essa mensagem logo acima da listagem dos produtos.

Primeiramente, precisamos importar o jQuery na listagem de produtos, o arquivo WEB-INF/jsp/produto/lista.jsp . Podemos fazer isso através adicionando a tag <head> importando um arquivo Javascript, como a seguir:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <head>
        <script type="text/javascript"
            src="/controle-produtos/js/jquery.js">
        </script>
    </head>
    <body>
        <!-- continuação da pagina -->
```

Podemos alterar o link para disparar um evento ao ser clicado, e não mais chamar uma URL. O nosso link não nos enviará para lugar nenhum, será mais um artifício visual para exibi-lo como tal, portanto, o seu atributo href ficará com o valor # . Adicionaremos também ao link um evento do tipo onclick que fará uma chamada à função removeProduto que precisará do id do produto, para saber quem será removido:

```
<td>
    <a href="#" onclick="return removeProduto(${produto.id})">
        Remover
    </a>
</td>
```

Precisamos implementar a nossa função removeProduto() . Para isso, dentro do body da nossa página vamos colocar a função que receberá o id para enviar a requisição para a lógica de exclusão. A resposta gerada por essa lógica, nós vamos colocar em uma div cujo id se chamará mensagem :

```
<script type="text/javascript">
    function removeProduto(id) {
        $('#mensagem')
            .load('/controle-produtos/produto/remove?produto.id=' + id);
    }
</script>
```

No jQuery a # serve para especificar qual elemento você deseja trabalhar através do id desse elemento. Vamos criar a div antes da tabela da listagem dos produtos:

```
<html>
    <head>
        <script type="text/javascript"
            src="/controle-produtos/js/jquery.js">
        </script>
```

```

</head>
<body>
    <h1>Produtos</h1>
    <div id="mensagem"></div>
    <!-- tabela para mostrar a lista dos produtos -->

```

Nossa remoção ainda não funcionará do jeito que queremos, pois, a resposta gerada está sendo a própria página da listagem e não uma mensagem de confirmação. Isso tudo acontece devido ao redirecionamento que colocamos na lógica para a remoção. Vamos retirar o redirecionamento e criarmos um .jsp para mostrar a mensagem de confirmação da exclusão.

O método `remove` do `ProdutoController` deverá possuir apenas o código da remoção:

```

public void remove(Produto produto) {
    produtoDao.remove(produto);
}

```

E vamos criar um novo .jsp que será chamado após a execução da remoção. Criaremos ele no diretório `WEB-INF/jsp/produto` com o nome `remove.jsp` e o seguinte conteúdo:

`Produto removido com sucesso`

Por fim, o último passo que precisamos fazer é remover da tabela o produto que foi excluído. Mais uma vez utilizaremos o jQuery para nos auxiliar. Vamos precisar identificar qual linha vamos remover da tabela. Para isso, todas as linhas (`tr`) terão uma identificação única que será composta pelo `id` de cada produto precedida pela palavra "produto", por exemplo, `produto1`, `produto2` e assim por diante.

```

<c:forEach var="produto" items="${produtoList}">
    <tr id="produto${produto.id}">
        <td>${produto.nome}</td>
        <td>${produto.preco}</td>
        <td>${produto.descricao}</td>
        <td>
            <fmt:formatDate pattern="dd/MM/yyyy"
                value="${produto.dataInicioVenda.time}" />
        </td>
        <td><a href="#" onclick="return removeProduto(${produto.id})">
            Remover
        </a></td>
    </tr>
</c:forEach>

```

Em nossa função `Javascript` para remover o produto, podemos também remover a linha da tabela:

```
$('#produto' + id).remove();
```

Vale lembrar que a forma que apresentamos é apenas uma das formas de fazer, existem muitas outras formas diferentes de se atingir o mesmo comportamento.

17.21 EXERCÍCIOS OPCIONAIS: ADICIONANDO AJAX NA NOSSA APLICAÇÃO

1.

- Na lista.jsp no diretório WEB-INF/jsp/produto , faça a importação do jQuery:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <head>
        <script type="text/javascript"
            src="">
        </script>
    </head>
    <body>
        <!-- continuação da pagina -->
```

- Altere os links para chamar uma função chamada removeProduto :

```
<td>
    <a href="#" onclick="return removeProduto(${produto.id})">
        Remover
    </a>
</td>
```

- Vamos criar a nossa função que executará o AJAX e removerá o item da lista. Adicione as seguintes linhas dentro do body da sua página lista.jsp :

```
<!-- inicio da pagina e import do javascript -->
<body>
    <script type="text/javascript">
        function removeProduto(id) {
            $('#mensagem')
                .load('<c:url value="/produto/remove"/>' +
                    '?produto.id=' + id);
            $('#produto' + id).remove();
        }
    </script>

    <!-- continuação da pagina -->
</body>
```

- Adicione uma div na sua página com o id mensagem, aonde será colocada a resposta devolvida pelo servidor:

```
<h1>Produtos</h1>
<div id="mensagem"></div>
<!-- tabela para mostrar a lista dos produtos -->
```

- Dê um id para os <tr> , dessa forma você poderá apagá-los:

```
<tr id="produto${produto.id}">
```

- Remova o redirecionamento do método remove do ProdutoController , de forma que ele fique como a seguir:

```
public void remove(Produto produto) {
    produtoDao.remove(produto);
}
```

- Por fim, crie no diretório WEB-INF/jsp/produto o arquivo remove.jsp com o seguinte conteúdo:

Produto removido com sucesso

- Acesse novamente a listagem dos produtos e faça a remoção deles. Perceba que a página não é recarregada quando você clica no *link*.

APÊNDICE - JAVA EE 6

"Nove pessoas não fazem um bebê em 1 mês" -- Fred Brooks

18.1 JAVA EE 6 E AS NOVIDADES

O Java EE, desde seu lançamento, é considerado uma maneira de desenvolvermos aplicativos Java com suporte a escalabilidade, flexibilidade e segurança. Em sua última versão, a 6, um dos principais focos foi simplificar e facilitar a codificação de aplicativos por parte dos desenvolvedores.

Lançada oficialmente no dia 10 de dezembro de 2009, a nova especificação Java EE 6 foi liberada para download juntamente com o Glassfish v3, que é a sua implementação de referência. O Java EE 6 vem com mudanças significativas que foram em parte baseadas na grande comunidade de desenvolvedores Java, e que visam facilitar de verdade o uso das principais tecnologias do Java EE.

Um dos principais problemas da antiga especificação Java EE era que, na maioria das vezes que desenvolvíamos algum aplicativo, não era necessário fazer uso de todas as tecnologias disponíveis nela. Mesmo usando apenas parte dessas tecnologias, tínhamos que lidar com todo o restante de tecnologias da especificação. Por exemplo, quando desenvolvemos Web Services, precisamos utilizar apenas as especificações JAX-WS e JAXB, mas precisamos lidar com todo o restante das especificações mesmo sem precisarmos delas.

Para resolver esse problema, no Java EE 6 foi introduzido o conceito de profiles. Um profile é uma configuração onde podemos criar um subconjunto de tecnologias presentes nas especificações Java EE 6 ou até mesmo adicionar novas tecnologias definidas pela JCP (Java Community Process) que não fazem parte da especificação. A própria especificação Java EE já incluiu um profile chamado **Web Profile**.

O Web Profile reúne apenas as tecnologias usadas pela maioria dos desenvolvedores Web. Ela inclui as principais tecnologias vista durante este curso (Servlets, JSP, JSTL) e outras tecnologias como, por exemplo, JPA e JSF que podem ser vistas no curso FJ-26 da Caelum.

No Java EE 5, foram feitas mudanças que já facilitaram muito a vida do desenvolvedor, como, por exemplo, o uso de POJOs (Plain Old Java Object) e anotações e preterindo o uso de XML's como forma de configuração. Além da criação da JPA (Java Persistence API) para facilitar o mapeamento objeto relacional em nossas aplicações cuja principal implementação é o famoso Hibernate.

Seguindo a onda de melhorias, o Java EE 6 introduziu melhorias em praticamente todas as

tecnologias envolvidas no desenvolvimento de aplicativos enterprise ou Web. Por exemplo:

- **DI (Dependency Injection)** - Usar anotações para criarmos classes que podem ser injetadas como dependência em outras classes;
- **JPA 2.0** - Melhorias na Java Persistence Query Language e a nova API de Criteria;
- **EJB 3.1** - Facilidades no desenvolvimento de Enterprise Java Beans usando a nova API de EJB 3.1;
- **Bean Validation** - Validar nossos POJOs de maneira fácil utilizando anotações;
- **JAX-RS** - Especificação sobre como criar Web Services de maneira RESTful.

18.2 PROCESSAMENTO ASSÍNCRONO

Muitas vezes em nossas aplicações temos alguns servlets que, para finalizarem a escrita da resposta ao cliente, dependem de recursos externos como, por exemplo, uma conexão JDBC, um Web Service ou uma mensagem JMS. Esses são exemplos de recurso que não temos controle quanto ao tempo de resposta.

Não tínhamos muita saída a não ser esperar pelo recurso para que nosso `Servlet` terminasse sua execução. Porém, essa solução nunca foi a mais apropriada, pois bloqueávamos uma thread que estava à espera de um recurso intermitente.

Na nova API de Servlets 3.0, esse problema foi resolvido. Podemos processar nossos servlets de maneira **assíncrona**, de modo que a thread não fica mais à espera de recursos. A thread é liberada para executar outras tarefas em nosso servidor de aplicação.

Quando o recurso que estávamos esperando se torna disponível, outra thread pode escrever a resposta e enviá-la ao cliente ou podemos, como foi visto durante o curso e vimos que é uma ótima prática, delegar a escrita da resposta para outras tecnologias como JSP.

Tanto servlets quanto filtros podem ser escritos de maneira assíncrona. Para isso, basta colocarmos o atributo `asyncSupported` como `true` nas anotações `@WebServlet` ou `@WebFilter`. Aqui vemos um exemplo de como fazer nosso servlet ter suporte à requisições assíncronas:

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        //Aguardando conexão JDBC
        ContatoDao dao = new ContatoDao();
    }
}
```

No exemplo acima, habilitamos suporte assíncrono em nossa servlet, porém ainda não tornamos o código realmente assíncrono. Na interface `javax.servlet.ServletRequest`, temos um método

chamado `startAsync` que recebe como parâmetros o `ServletRequest` e o `ServletResponse` que recebemos em nosso servlet. Uma chamada a este método torna a requisição assíncrona. Agora sim tornamos nosso código assíncrono:

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        AsyncContext context = req.startAsync(req, res);

        //Aguardando conexão JDBC
        ContatoDao dao = new ContatoDao();
    }
}
```

Note que, ao fazer uma chamada ao método `startAsync`, é retornado um objeto do tipo `AsyncContext`. Esse objeto guarda os objetos `request` e `response` passados ao método `startAsync`. A partir dessa chamada de método, a thread que tratava nossa requisição foi liberada para executar outras tarefas. Na classe `AsyncContext`, temos o método `complete` para confirmar o envio da resposta ao cliente.

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        final AsyncContext context = req.startAsync(req, res);
        final PrintWriter out = res.getWriter();

        context.addListener(new AsyncListener() {
            @Override
            public void onComplete(AsyncEvent event) throws IOException {
                out.println("<html>");
                out.println("Olá mundo!");
                out.println("</html>");
            }
        });
    }

    //Aguardando conexão JDBC
    ContatoDao dao = new ContatoDao();
    //Nossa logica

    //Lógica completa, chama listener onComplete
    context.complete();
}
}
```

Também podemos delegar a escrita da resposta para um JSP por exemplo. Neste caso, faremos uso do método `dispatch`:

```
@WebServlet(asyncSupported=true, urlPatterns={"/adiciona/contato"})
public class AdicionaContatoServlet extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        final AsyncContext context = req.startAsync(req, res);

        context.addListener(new AsyncListener() {
            @Override
            public void onComplete(AsyncEvent event) throws IOException {
                context.dispatch("/adicionado.jsp");
            }
        });
    }
}
```

```

    });

    //Aguardando conexão JDBC
    ContatoDao dao = new ContatoDao();
    //Nossa logica

    //Lógica completa, chama listener onComplete
    context.complete();
}
}

```

ASYNCLISTENER E SEUS OUTROS MÉTODOS

A classe `AsyncListener` suporta outros eventos. Quando a chamada assíncrona excede o tempo máximo que ela tinha para ser executada sobrescrevemos o método `onTimeout`. Para tratarmos um erro podemos sobreescrivê-lo no método `onError`.

Podemos prosseguir configurando nossas Servlets e Filtros no arquivo `web.xml`. Para configurarmos um servlet ou filtro como possivelmente assíncrono devemos especificar a tag `<async-supported>true</async-supported>` no mapeamento da servlet ou do filtro.

18.3 PLUGABILIDADE E WEB FRAGMENTS

Com a criação das novas anotações disponíveis na API Servlets 3.0, o uso do `web.xml` não é mais obrigatório. O `web.xml` é usado quando desejamos sobreescrivê-lo algumas configurações definidas em nossas anotações.

Percebemos que a grande ideia da nova API de Servlets é criar a menor quantidade de XML possível e colocar nossas configurações em anotações. Se pensarmos nos frameworks que utilizamos para nos auxiliar no desenvolvimento Web, por exemplo, o `VRaptor` ou o `Struts`, sabemos que ele exige que seja feita a configuração de um filtro no arquivo `web.xml` de nossa aplicação. Temos um fato que vai contra as premissas da nova API de Servlets, que é criar um arquivo `web.xml` quando o mesmo deveria ser opcional.

Com o intuito de resolver esse problema, foi introduzido o conceito de **web fragment**, que são módulos de um `web.xml`. Podemos ter vários `web fragment` que, em conjunto, podem ser visto como se fossem um `web.xml` completo. O framework que utilizamos pode criar seu próprio `web fragment` e colocar nele todas as configurações que teríamos que fazer manualmente em nosso `web.xml`. Por exemplo, no caso do `VRaptor` teríamos a configuração do filtro obrigatório do `VRaptor` dentro do próprio framework. Com isso, seríamos poupadinhos de termos que colocar qualquer configuração em nossa aplicação. O próprio container conseguirá buscar por essas configurações e aplicá-las em nossa aplicação.

No web fragment podemos adicionar quase todos os elementos que estão disponíveis para o arquivo web.xml. As únicas restrições que devemos seguir são:

- Arquivo **deve** ser chamado de web-fragment.xml
- Root tag do arquivo **deve** ser <web-fragment>

No caso do VRaptor o arquivo web-fragment.xml ficaria assim:

```
<web-fragment>
    <filter>
        <filter-name>vraptor</filter-name>
        <filter-class>br.com.caelum.vraptor.VRaptor</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>vraptor</filter-name>
        <url-pattern>/*</url-pattern>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>
</web-fragment>
```

De preferência o framework que utilizamos devem colocar seus web fragments na pasta META-INF do arquivo .jar , que está normalmente localizado na pasta WEB-INF/lib da nossa aplicação.

O VRaptor 3.1 já possui essa configuração de Servlets 3.0 pronta.

TAG METADATA - COMPLETE

Neste caso se a tag metadata-complete for setada como true o container não analisará qualquer web fragment presente em nossa aplicação, até mesmo os web fragments de algum framework que utilizamos.

Essa funcionalidade permite modularizarmos nosso web.xml . Podemos ter o tradicional arquivo descriptor web.xml , ou podemos modularizá-lo em um ou mais web fragment .

Devido a essa possível modularização, pode ser importante a ordem em que esses fragmentos são processados. Por exemplo, a ordem em que estes fragmentos são processados pode alterar a ordem de execução dos filtros em nossa aplicação.

Servlets 3.0 nos permite configurar a ordem que estes fragmentos são processados de maneira absoluta ou relativa. Se quisermos configurar nossos fragmentos para serem processados em ordem absoluta, devemos colocar a tag <absolute-ordering> em nosso web.xml . Também podemos configurar uma ordem relativa colocando a tag <ordering> no arquivo web-fragment.xml .

Nossos fragmentos podem ter identificadores, nomes que os diferenciam. Sendo assim, se quisermos configurar uma ordem absoluta para nossos fragmentos, teríamos a seguinte configuração em nosso

web.xml :

```
<web-app>
    <name>Minha Aplicação</name>
    <absolute-ordering>
        <name>Fragmento1</name>
        <name>Fragmento2</name>
    </absolute-ordering>
</web-app>
```

No código acima, eles seriam processados na seguinte ordem:

- web.xml - Sempre será processado primeiro
- Fragmento1
- Fragmento2

A API de Servlets 3.0 tem uma nova interface chamada `ServletContainerInitializer` que nos permite plugar frameworks em nossa aplicação. Para isso, basta que o framework crie uma classe que implemente esta interface.

Por exemplo, na implementação da API JAX-WS (Web Services RESTful, que vemos no curso FJ-31) seria criado uma classe que implemente esta interface:

```
@HandlesTypes(WebService.class)
public class JAXWSServletContainerInitializer
    implements ServletContainerInitializer {
    public void onStartup(Set<Class<?>> c, ServletContext ctx)
        throws ServletException {
        ServletRegistration reg =
            ctx.addServlet("JAXWSServlet",
                "com.sun.webservice.JAXWSServlet");
        reg.addServletMapping("/jaxws");
    }
}
```

Os frameworks que implementam esta interface devem colocá-las na pasta META-INF/services em um arquivo chamado `javax.servlet.ServletContainerInitializer` que faça referência para a classe de implementação. Quando nosso container for iniciado, ele vai procurar por essas implementações quando ele for iniciado.

A anotação `@HandlesTypes` serve para especificar quais classes podem ser manipuladas pela implementação de `ServletContainerInitializer`

18.4 REGISTRO DINÂMICO DE SERVLETS

Na nova API de Servlets 3.0 temos a possibilidade de adicionar um servlet em nossa aplicação de 3 maneiras. Duas delas já foram vistas durante este capítulo (XML e anotações). A terceira delas pode ser feita de uma maneira programática ou dinâmica.

Dentro do Java EE podemos classificar os objetos principais que habitam nossa aplicação em 3

escopos. O escopo de requisição que tem uma duração curta, o escopo de sessão que é muito utilizado quando queremos guardar informações referentes a algum usuário específico e o terceiro deles que não abordamos durante o curso que é o escopo de aplicação.

São objetos que permanecem em memória desde o momento que nossa aplicação é iniciada até o momento que a mesma é finalizada. Os servidores de aplicação nos fornecem um objeto que é uma implementação da interface `ServletContext` que é um objeto de escopo de aplicação.

Com esse objeto podemos fazer coisas relativas a aplicação, como por exemplo, fazer logs de acesso, criar atributos que podem ser compartilhados por toda a aplicação, etc.

A interface `ServletContext` nos fornece alguns métodos adicionais que nos permitem fazer o registro dinâmico de servlets e filtros. A única restrição para chamarmos esses métodos é que devemos fazer isso no momento em que nossa aplicação está sendo iniciada em nosso servidor de aplicação.

Podemos fazer isso de duas maneiras distintas. A primeira delas e a mais conhecida, seria criarmos um listener do tipo `ServletContextListener` e adicionarmos o servlet no método `contextInitialized` como no código abaixo:

```
public class ServletListener implements ServletContextListener {  
    public void contextInitialized(ServletContextEvent event) {  
        ServletContext context = event.getServletContext();  
        context.addServlet("MeuServlet", MeuServlet.class);  
    }  
  
    public void contextDestroyed(ServletContextEvent event) {  
    }  
}
```

Para adicionarmos um filtro, faríamos algo bem similar: só mudaríamos a chamada de `addServlet` para `addFilter` e passaríamos como parâmetro o filtro que desejamos adicionar.

A outra maneira de fazermos isso, seria criarmos uma implementação de `ServletContainerInitializer` e no método `onStartup`. Por exemplo:

```
public class MeuServletContainerInitializer  
    implements ServletContainerInitializer {  
    public void onStartup(Set<Class<?>> c,  
        ServletContext context) throws ServletException {  
        ServletRegistration reg = context  
            .addServlet("MeuServlet", MeuServlet.class);  
        reg.addServletMapping("/jaxws");  
    }  
}
```

Este segundo exemplo seria mais usado se fossemos distribuir nossa aplicação como um framework e nos beneficiarmos da plugabilidade como foi visto na seção anterior.

APÊNDICE - TÓPICOS DA SERVLET API

*"Measuring programming progress by lines of code is like measuring aircraft building progress by weight." -
- Bill Gates*

Este capítulo aborda vários outros pequenos assuntos da Servlet API ainda não tratados, muitos deles importantes para a certificação SCWCD.

19.1 INIT-PARAMS E CONTEXT-PARAMS

Podemos configurar no web.xml alguns parâmetros que depois vamos ler em nossa aplicação. Uma forma é passarmos parâmetros especificamente para uma Servlet ou um filtro usando a tag `<init-param>` como nos exemplos abaixo:

```
<!-- em servlet -->
<servlet>
    <servlet-name>MinhaServlet</servlet-name>
    <servlet-class>pacote.MinhaServlet</servlet-class>
    <init-param>
        <param-name>nome</param-name>
        <param-value>valor</param-value>
    </init-param>
</servlet>

<!-- em filter -->
<filter>
    <filter-name>MeuFiltro</filter-name>
    <filter-class>pacote.Meufiltro</filter-class>
    <init-param>
        <param-name>nome</param-name>
        <param-value>valor</param-value>
    </init-param>
</filter>
```

Podemos, inclusive, ter vários parâmetros na mesma servlet ou filtro. Depois, no código Java da Servlet ou do filtro específico, podemos recuperar esses parâmetros usando:

```
// em servlet
String valor = getServletConfig().getInitParameter("nome");

// em filtro, no init
String valor = filterConfig.getInitParameter("nome")
```

Outra possibilidade é configurar parâmetros para o contexto inteiro e não apenas uma servlet específica. Podemos fazer isso com a tag `<context-param>`, como abaixo:

```
<context-param>
  <param-name>nome</param-name>
  <param-value>param</param-value>
</context-param>
```

E, no código Java de uma Servlet, por exemplo:

```
String valor = getServletContext().getInitParameter("nome");
```

Muitos frameworks usam parâmetros no web.xml para configurar. O VRaptor e o Spring são exemplos de uso desse recurso. Mas podemos usar isso em nossas aplicações também para retirar do código Java certas configurações parametrizáveis.

19.2 WELCOME-FILE-LIST

É possível configurar no **web.xml** qual arquivo deve ser chamado quando alguém acessar uma URL raiz no servidor, como por exemplo:

```
http://localhost:8080/fj-21-agenda/
http://localhost:8080/fj-21-agenda/uma-pasta/
```

São os arquivos index que normalmente usamos em outras plataformas. Mas no Java EE podemos listar os nomes de arquivos que desejamos que sejam os *welcome files*. Basta defini-los no XML e o servidor vai tentá-los na ordem de declaração:

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>default.html</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

19.3 PROPRIEDADES DE PÁGINAS JSP

Como dizer qual o encoding de nossos arquivos jsp de uma maneira global? Como nos proteger de programadores iniciantes em nossa equipe e desabilitar o código *scriptlet*? Como adicionar um arquivo antes e/ou depois de todos os arquivos JSPs? Ou de todos os JSPs dentro de determinado diretório?

Para responder essas e outras perguntas, a API de jsp resolveu possibilitar definir algumas tags no nosso arquivo **web.xml**.

Por exemplo, para desativar *scripting* (os scriptlets):

```
<scripting-invalid>true</scripting-invalid>
```

Ativar *expression language* (que já vem ativado):

```
<el-ignored>false</el-ignored>
```

Determinar o encoding dos arquivos de uma maneira genérica:

```
<page-encoding>UTF-8</page-encoding>
```

Incluir arquivos estaticamente antes e depois de seus JSPs:

```
<include-prelude>/antes.jspf</include-prelude>
<include-coda>/depois.jspf</include-coda>
```

O código a seguir mostra como aplicar tais características para todos os JSPs, repare que a tag `url-pattern` determina o grupo de arquivos cujos atributos serão alterados:

```
<jsp-config>
  <jsp-property-group>
    <display-name>todos os jsps</display-name>
    <description>configuracoes de todos os jsps</description>
    <url-pattern>*.jsp</url-pattern>
    <page-encoding>UTF-8</page-encoding>
    <scripting-invalid>true</scripting-invalid>
    <el-ignored>false</el-ignored>
    <include-prelude>/antes.jspf</include-prelude>
    <include-coda>/depois.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

19.4 INCLUSÃO ESTÁTICA DE ARQUIVOS

Existe uma maneira em um arquivo JSP de incluir um outro arquivo estaticamente. Isto faz com que o arquivo a ser incluído seja literalmente copiado e colado dentro do seu arquivo antes da primeira interpretação (compilação) do seu jsp.

A vantagem é que como a inclusão é feita uma única vez antes do arquivo ser compilado, essa inclusão é extremamente rápida, porém vale lembrar que o arquivo incluído pode ou não funcionar separadamente.

```
<%@ include file="outra_pagina.jsp" %>
```

19.5 TRATAMENTO DE ERRO EM JSP

Como tratar possíveis exceptions em nossa página JSP? Nossos exercícios de listagem de contatos tanto com scriptlets quanto com JSTL usam o `ContatoDao` que pode lançar uma exceção se o banco de dados estiver fora do ar, por exemplo. Como tratar?

Se nosso JSP é um imenso scriptlet de código Java, o tratamento é o mesmo de códigos Java normais:

```
try catch :
```

```
<html>
  <%
  try {
    ContatoDao dao = new ContatoDao();
    // ... etc ...
  } catch(Exception ex) {
  %>
    Ocorreu algum erro ao acessar o banco de dados.
```

```
<%
}
%>
</html>
```

Não parece muito elegante. Mas e quando usamos tags, há uma forma melhor? Poderíamos usar a tag `c:catch`, com o mesmo tipo de problema da solução anterior:

```
<c:catch var="error">
  <jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDao"/>
  <c:forEach var="contato" items="${dao.lista}">
    ...
  </c:forEach>
</c:catch>
<c:if test="${not empty error}">
  Ocorreu algum erro ao acessar o banco de dados.
</c:if>
```

Repare que a própria JSTL nos apresenta uma solução que não se mostra boa para esse tipo de erro que queremos tratar. É importante deixar claro que desejamos tratar o tipo de erro que não tem volta, devemos mostrar uma mensagem de erro para o cliente e pronto, por exemplo quando a conexão com o banco cai ou quando ocorre algum erro no servidor.

Quando estávamos trabalhando com Servlets, havia uma solução simples e elegante: não tratar as exceções de forma espalhada mas sim criar uma página centralizada de tratamento de erros. Naquele caso, conseguimos isso com o `<error-page>`.

Com JSPs, conseguimos o mesmo resultado mas sem XML. Usamos uma diretiva no topo do JSP que indica qual é a página central de tratamento de erro. E nesse caso não precisamos nem de `try/catch` nem de `<c:catch>`:

```
<%@ page errorPage="/erro.html" %>
...
<jsp:useBean id="dao" class="br.com.caelum.jdbc.dao.ContatoDao"/>
...
```

19.6 DESCOBRINDO TODOS OS PARÂMETROS DO REQUEST

Para ler todos os parâmetros do `request` basta acessar o método `getParameterMap` do `request`.

```
Map<String, Object> parametros = request.getParameterMap();
for(String parametro:parametros.keySet()) {
  // faça algo com o parametro
}
```

19.7 TRABALHANDO COM LINKS COM A C:URL

Às vezes não é simples trabalhar com links pois temos que pensar na URL que o cliente acessa ao visualizar a nossa página.

A JSTL resolve esse problema: supondo que a sua aplicação se chame `jspteste`, o código abaixo

gera a string /jspteste/imagem/banner.jpg .

```
<c:url value="/imagem/banner.jpg"/>
```

É bastante útil ao montar menus únicos incluídos em várias páginas e que precisam lidar com links absolutos.

19.8 CONTEXT LISTENER

Sabemos que podemos executar código no momento que uma Servlet ou um filtro são inicializados através dos métodos `init` de cada um deles. Mas e se quisermos executar algo no início da aplicação Web (do contexto Web), independente de termos ou não Servlet e filtros e do número deles?

Para isso existem os **context listeners**. Você pode escrever uma classe Java com métodos que serão chamados automaticamente no momento que seu contexto for iniciado e depois desligado. Basta implementar a interface `ServletContextListener` e usar a tag `<listener>` no web.xml para configurá-la.

Por exemplo:

```
public class MeuListener implements ServletContextAttributeListener {  
    public void contextInitialized(ServletContextEvent event) {  
        System.out.println("Contexto iniciado...");  
    }  
  
    public void contextDestroyed(ServletContextEvent event) {  
        System.out.println("Contexto desligado...");  
    }  
}
```

E depois no XML:

```
<listener>  
    <listener-class>pacote.Meulistener</listener-class>  
</listener>
```

19.9 O SERVLETCONTEXT E O ESCOPO DE APLICAÇÃO

As aplicações Web em Java têm 3 escopos possíveis. Já vimos e usamos dois deles: o de request e o de sessão. Podemos colocar um atributo no request com `request.setAttribute(...)` e ele estará disponível para todo o request (desde a Action até o JSP, os filtros etc).

Da mesma forma, podemos pegar a `HttpSession` e colocar um atributo com `session.setAttribute(...)` e ela estará disponível na sessão daquele usuário através de vários requests.

O terceiro escopo é um escopo global, onde os objetos são compartilhados na aplicação inteira, por todos os usuários em todos os requests. É o chamado **escopo de aplicação**, acessível pelo

`ServletContext .`

Podemos, em uma Servlet, setar algum atributo usando:

```
getServletContext().setAttribute("nomeGlobal", "valor");
```

Depois, podemos recuperar esse valor com:

```
Object valor = getServletContext().getAttribute("nomeGlobal");
```

Um bom uso é compartilhar configurações globais da aplicação, como por exemplo usuário e senha de um banco de dados, ou algum objeto de cache compartilhado etc. Você pode, por exemplo, inicializar algum objeto global usando um `ServletContextListener` e depois disponibilizá-lo no `ServletContext` para o resto da aplicação acessar.

E como fazemos para acessar o escopo de aplicação no nosso JSP? Simples, uma das variáveis que já existe em um JSP se chama `application`, algo como:

```
ServletContext application = getServletContext();
```

Portanto podemos utilizá-la através de scriptlet:

```
<%= application.getAttribute("nomeGlobal") %><br/>
```

Como já vimos anteriormente, o código do tipo scriptlet pode ser maléfico para nossa aplicação, sendo assim vamos utilizar Expression Language para acessar um atributo do escopo aplicação:

```
Acessando com EL: ${nomeGlobal}<br/>
```

Repare que a Expression Language procurará tal atributo não só no escopo do `application`, como veremos mais a frente. Para deixar claro que você procura uma variável do escopo de aplicação, usamos a variável implícita chamada `applicationScope`:

```
Acessando escopo application: ${applicationScope['nomeGlobal']}<br/>
```

MÉTODOS NO SERVLETCONTEXT

Além da característica de escopo global com os métodos `getAttribute` e `setAttribute`, outros métodos úteis existem na `ServletContext`. Consulte o Javadoc para mais informações.

19.10 OUTROS LISTENERS

Há ainda outros listeners disponíveis na API de Servlets para capturar outros tipos de eventos:

- `HttpSessionListener` - provê métodos que executam quando uma sessão é criada (`sessionCreated`), destruída (`sessionDestroyed`);

- `ServletContextAttributeListener` - permite descobrir quando atributos são manipulados no `ServletContext` com os métodos `attributeAdded` , `attributeRemoved` e `attributeReplaced` ;
- `ServletRequestAttributeListener` - tem os mesmos métodos que o `ServletContextAttributeListener` mas executa quando os atributos do *request* são manipulados;
- `HttpSessionAttributeListener` - tem os mesmos métodos que o `ServletContextAttributeListener` mas executa quando os atributos da `HttpSession` são manipulados;
- `ServletRequestListener` - permite executar códigos nos momentos que um request chega e quando ele acaba de ser processado (métodos `requestDestroyed` e `requestInitialized`);
- Outros menos usados: `HttpSessionActivationListener` e `HttpSessionBindingListener` .

A configuração de qualquer um desses listeners é feita com a tag `<listener>` como vimos acima. É possível inclusive que uma mesma classe implemente várias das interfaces de listeners mas seja configurada apenas uma vez o web.xml.