FJ-21

Java para Desenvolvimento Web

Caelum
Ensino e Inovação
www.caelum.com.br

A Caelum atua no mercado com consultoria, desenvolvimento e ensino em computação. Sua equipe participou do desenvolvimento de projetos em vários clientes e, após apresentar os cursos de verão de Java na Universidade de São Paulo, passou a oferecer treinamentos para o mercado. Toda a equipe tem uma forte presença na comunidade através de eventos, artigos em diversas revistas, participação em muitos projetos *open source* como o VRaptor e o Stella e atuação nos fóruns e listas de discussão como o GUJ.

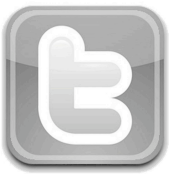
Com uma equipe de mais de 60 profissionais altamente qualificados e de destaque do mercado, oferece treinamentos em Java, Ruby on Rails e Scrum em suas três unidades - São Paulo, Rio de Janeiro e Brasília. Mais de 8 mil alunos já buscaram qualificação nos treinamentos da Caelum tanto em nas unidades como nas próprias empresas com os cursos *incompany*.

O compromisso da Caelum é oferecer um treinamento de qualidade, com material constantemente atualizado, uma metodologia de ensino cuidadosamente desenvolvida e instrutores capacitados tecnicamente e didaticamente. E oferecer ainda serviços de consultoria ágil, mentoring e desenvolvimento de projetos sob medida para empresas.

Comunidade

Nossa equipe escreve constantemente artigos no **Blog da Caelum** que já conta com 150 artigos sobre vários assuntos de Java, Rails e computação em geral. Visite-nos e assine nosso RSS:

➡ blog.caelum.com.br



Acompanhe também a equipe Caelum no **Twitter**:

➡ twitter.com/caelumdev/equipe

O **GUJ** é maior fórum de Java em língua portuguesa, com 700 mil posts e 70 mil usuários. As pessoas da Caelum participam ativamente, participe também:

➡ www.guj.com.br

Assine também nossa **Newsletter** para receber as novidades e destaques dos eventos, artigos e promoções da Caelum:

➡ www.caelum.com.br/newsletter

No site da Caelum há algumas de nossas **Apostilas** disponíveis gratuitamente para download e alguns dos **artigos** de destaque que escrevemos:

➡ www.caelum.com.br/apostilas

➡ www.caelum.com.br/artigos

Conheça alguns de nossos cursos

FJ-11:
Java e Orientação a objetos

FJ-25:
Persistência com JPA2 e Hibernate

FJ-16:
Laboratório Java com Testes, XML e Design Patterns

FJ-26:
Laboratório Web com JSF2 e CDI

FJ-19:
Preparatório para Certificação de Programador Java

FJ-31:
Java EE avançado e Web Services

FJ-21:
Java para Desenvolvimento Web

FJ-91:
Arquitetura e Design de Projetos Java

RR-71:
Desenvolvimento Ágil para Web 2.0 com Ruby on Rails

RR-75:
Ruby e Rails avançados: lidando com problemas do dia a dia

Para mais informações e outros cursos, visite: caelum.com.br/cursos

- ✓ Mais de 8000 alunos treinados;
- ✓ Reconhecida nacionalmente;
- ✓ Conteúdos atualizados para o mercado e para sua carreira;
- ✓ Aulas com metodologia e didática cuidadosamente preparadas;
- ✓ Ativa participação nas comunidades Java, Rails e Scrum;
- ✓ Salas de aula bem equipadas;
- ✓ Instrutores qualificados e experientes;
- ✓ Apostilas disponíveis no site.

Caelum
Ensino e Inovação

Sobre esta apostila

Esta apostila da Caelum visa ensinar de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material. Todos os comentários, críticas e sugestões serão muito bem-vindos.

Essa apostila é constantemente atualizada e disponibilizada no site da Caelum. Sempre consulte o site para novas versões e, ao invés de anexar o PDF para enviar a um amigo, indique o site para que ele possa sempre baixar as últimas versões. Você pode conferir o código de versão da apostila logo no final do índice.

Baixe sempre a versão mais nova em: www.caelum.com.br/apostilas

www.caelum.com.br

Índice

Versão: 13.1.27

CAPÍTULO 1

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 Dostoievski, 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 Struts 2 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 impeditivo 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 - Bibliografia

É possível aprender muitos dos detalhes e pontos não cobertos no treinamento em tutoriais na Internet em portais como o GUJ, em blogs (como o da Caelum: http://blog.caelum.com.br) e em muitos sites especializados.

Sobre Java para Web

- Use a cabeça! JSP e Servlets

Sobre Java e melhores práticas

- Refactoring, Martin Fowler
- Effective Java - 2nd edition, Joshua Bloch
- Design Patterns, Erich Gamma et al

Para iniciantes na plataforma Java:

- Apostila do FJ-11 da Caelum (disponível online)
- Java - Como programar, de Harvey M. Deitel
- Use a cabeça! - Java, de Bert Bates e Kathy Sierra

CAPÍTULO 2

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, seja 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, e também existem diversas ferramentas do tipo **ORM** (*Object Relational Mapping*) que facilitam bastante o uso do JDBC. Neste momento, iremos focar nos conceitos e no uso do JDBC. Veremos um pouco da ferramenta de ORM Hibernate ao final deste mesmo curso e, no curso FJ-26, com muitos detalhes, recursos e tópicos avançados.

2.2 - Persistindo através de Sockets?

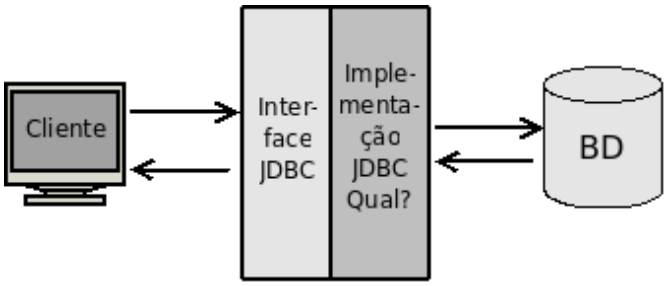
Para conectar-se a um banco de dados poderíamos abrir *sockets* diretamente com o servidor que o hospeda, por exemplo um Oracle ou MySQL e nos comunicarmos com ele através de seu protocolo proprietário.

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.

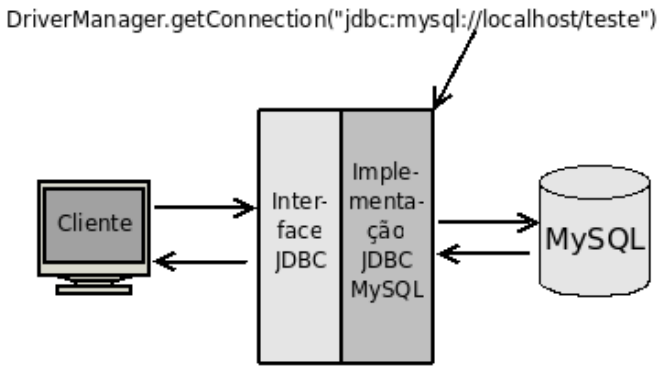
2.3 - A conexão em Java

Conectar-se a um banco de dados com Java é feito de maneira elegante. Para evitar que cada banco tenha a sua própria API e 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 ela 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 e 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 iremos utilizar 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 que?

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*, 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, fazemos isso através de um clique da direita em nosso projeto, *Properties/Java Build Path* e em *Libraries* adicionamos o jar do driver JDBC do MySQL. 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`.

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-26.

Lista de drivers

Os drivers podem ser baixados normalmente no site do fabricante do banco de dados. A Sun possui um sistema de busca de drivers em seu site: http://developers.sun.com/product/jdbc/drivers
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);
        }
```

```
    }
}
```

Poderíamos colocar um aviso na nossa aplicação, notificando todos os programadores a adquirir uma conexão:

```
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 elas vierame 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 que indicaria 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 simplesmente resolver 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. Essa solução padrão tem o nome de **Design Pattern** (**Padrão 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

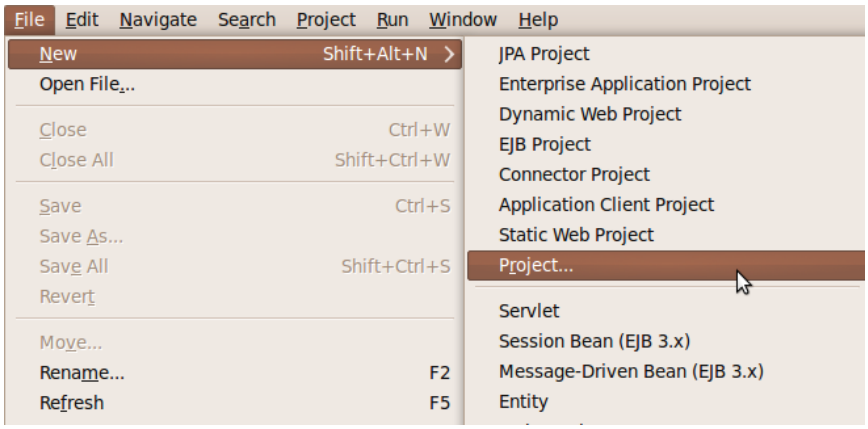
1) a) Na Caelum, clique no ícone do Eclipse no Desktop;

> **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

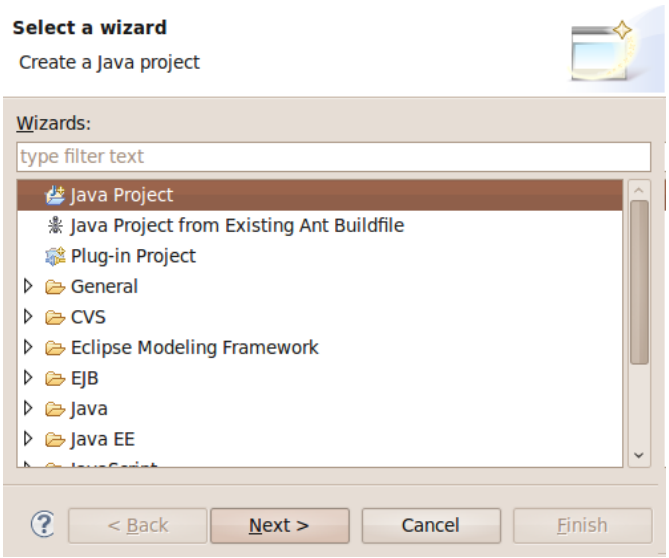
b) Feche a tela de *Welcome* caso ela apareça

c) Vamos criar um projeto no Eclipse chamado `jdbc`.

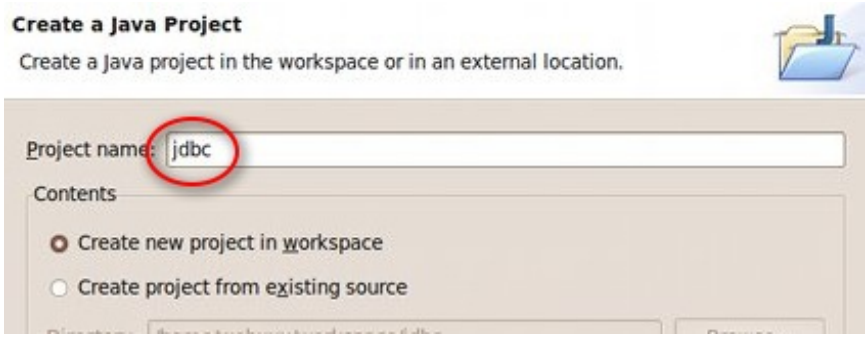
d) Vá em **File -> New -> Project**:



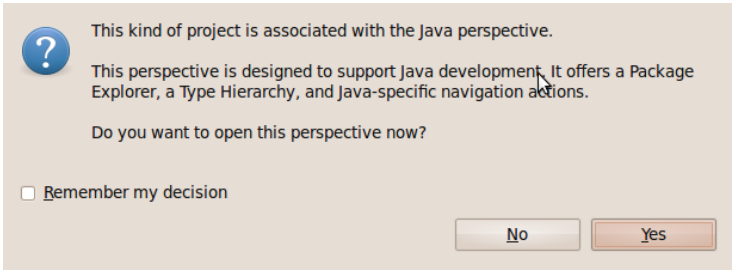
e) Selecione **Java Project** e clique em **Next**:



f) Coloque o nome do projeto como **jdbc** e clique em **Finish**:



g) Aceite a mudança de perspectiva:

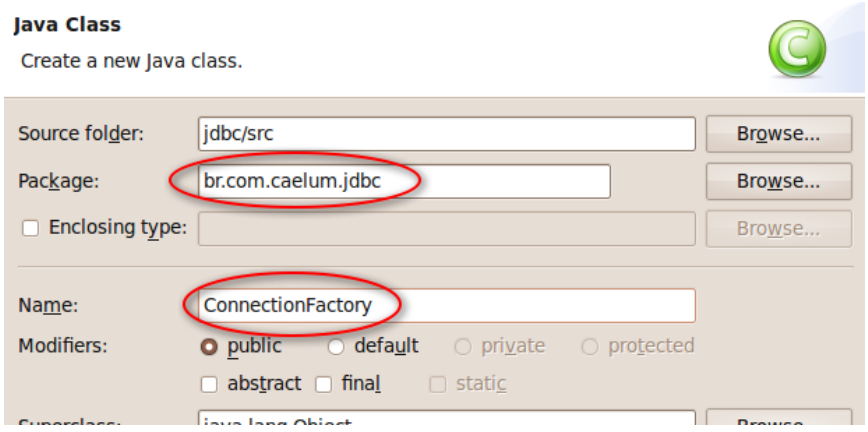


2) Copie o driver do `MySQL` para o seu projeto.

a) no seu Desktop, clique na pasta **Caelum/21**;
b) clique da direita no driver do MySQL mais novo, escolha Copy;
c) vá para sua pasta principal (webXXX) na coluna da direita do *File Browser*;
d) entre no diretório workspace, `jdbc`;
e) clique da direita e escolha Paste: você acaba de colocar o arquivo ".jar" no seu projeto.

3) Vamos criar a classe que fábrica conexões:

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



c) Clique em **Finish**

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

```
1 public Connection getConnection() {

2     System.out.println("Conectando ao banco");
3     try {
4         return DriverManager.getConnection("jdbc:mysql://localhost/fj21", "root", "");
5     } catch(SQLException e) {
6         throw new RuntimeException(e);
7     }
8 }
```

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

a) Crie um método `main` dentro da classe. Use o atalho do Eclipse para ajudar.

b) 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`:

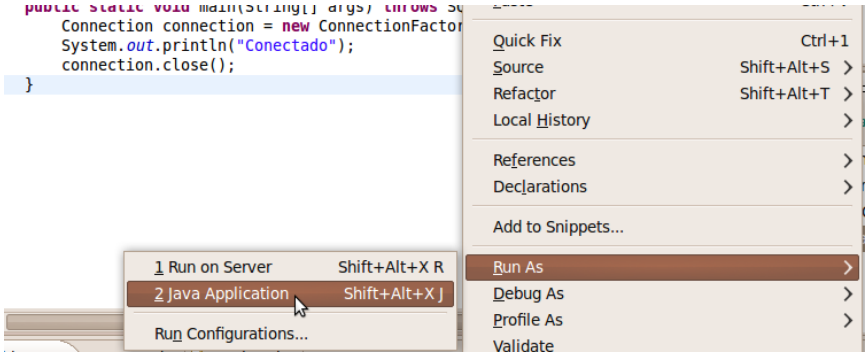
```
Connection connection = new ConnectionFactory().getConnection();

System.out.println("Conexão aberta!");
connection.close();
```

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

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

a) Clique da direita na sua classe `TestaConexao`

b) Escolha **Run as**, **Java Application** (caso prefira, aprenda a tecla de atalho para agilizar nas próximas execuções)



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

a) Clique no seu projeto, pressione F5 para executar um *Refresh*.

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

c) 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
```

Agora 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 a 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 por diversos motivos, normalmente as classes de modelo da nossa aplicação costumam ser javabeans.

Agora utilizaremos:

- uma classe com métodos do tipo `get` e `set` para cada um de seus parâmetros, que representa algum objeto;
- uma classe com construtor sem argumentos que representa uma coleção de objetos.

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

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

    public String getEndereco() {
        return this.endereco;
    }
    public void setEndereco(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://java.sun.com/products/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 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-16 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 no formato que o banco de dados entenda e como uma `String`, 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 strings). 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 iremos utilizar 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 agora 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 milisegundos 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 milisegundos:

```
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();
```

Agora imagine todo esse processo sendo escrito toda vez que desejar inserir algo no banco? Ainda não consegue visualizar o quão 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();
    }
}
```

Fechando a conexão

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

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.

JodaTime

A API de datas do Java, mesmo considerando algumas melhorias da `Calendar` em relação a `Date`, ainda é muito pobre. Existe uma chance de que na versão 7 entre novas classes para facilitar o trabalho com datas e horários, baseado na excelente biblioteca **JodaTime**.
Para mais informações: http://blog.caelum.com.br/2007/03/15/jsr-310-date-and-time-api/ http://jcp.org/en/jsr/detail?id=310

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 idéia 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 adiciona 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, passariamos 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());

      // grave 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 idéia, 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.

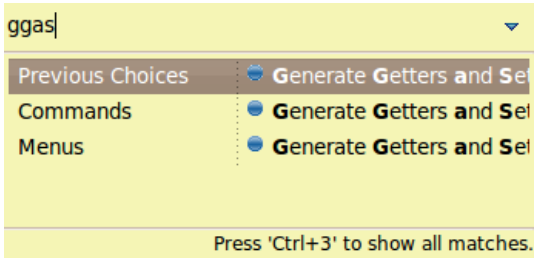
2.11 - Exercícios: Javabeans e ContatoDAO

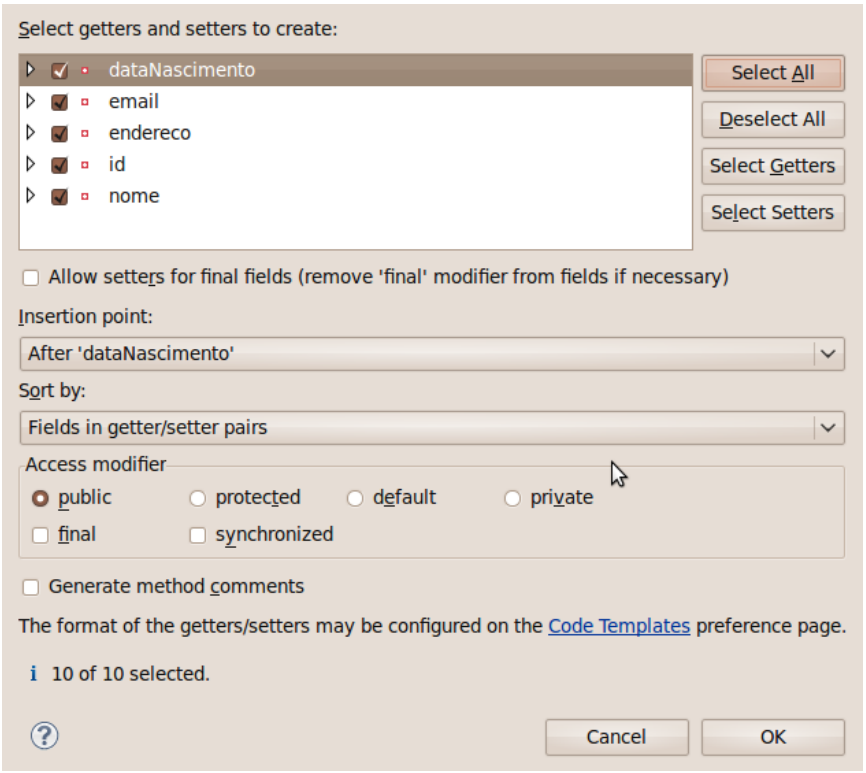
1) Crie a classe de `Contato` no pacote `br.com.caelum.jdbc.modelo`. Ela será nosso JavaBean para representar a entidade do banco. Deverá ter id, nome, email, endereço e uma data de nascimento.

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

```
1 public class Contato {

2     private Long id;
3     private String nome;
4     private String email;
5     private String endereco;
6     private Calendar dataNascimento;
7 }
```

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) Agora 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:

```
1 public class ContatoDAO {
```

```
2
3     // a conexão com o banco de dados
4     private Connection connection;
5
6     public ContatoDAO() {
7         this.connection = new ConnectionFactory().getConnection();
8     }
9
10 }
```

Use o Eclipse para ajudar com os imports! (atalho *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:

```
1 public void adiciona(Contato contato) {

2     String sql = "insert into contatos (nome,email,endereco,dataNascimento) values (?,?,?,?)";
3
4     try {
5           // prepared statement para inserção
6           PreparedStatement stmt = connection.prepareStatement(sql);
7
8         // seta os valores
9         stmt.setString(1,contato.getNome());
10        stmt.setString(2,contato.getEmail());
11        stmt.setString(3,contato.getEndereco());
12        stmt.setDate(4, new Date( contato.getDataNascimento().getTimeInMillis() ));
13
14        // executa
15        stmt.execute();
16        stmt.close();
17    } catch (SQLException e) {
18        throw new RuntimeException(e);
19    }
20 }
```

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:

```
1 // pronto para gravar

2 Contato contato = new Contato();
3 contato.setNome("Caelum");
4 contato.setEmail("contato@caelum.com.br");
5 contato.setEndereco("R. Vergueiro 3185 cj57");
6 contato.setDataNascimento(Calendar.getInstance());
7
8 // grave nessa conexão!!!
9 ContatoDAO dao = new ContatoDAO();
10
```

```
11 // método elegante
12 dao.adiciona(contato);
13
14 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 irá retornar `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 idéias 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

1) Crie o método `getLista` na classe `ContatoDAO`. Importe `List` de `java.util`:

```
1  public List<Contato> getLista() {
2      try {
3          List<Contato> contatos = new ArrayList<Contato>();
4          PreparedStatement stmt = this.connection.prepareStatement("select * from contatos");
5          ResultSet rs = stmt.executeQuery();
6
7          while (rs.next()) {
8              // criando o objeto Contato
9              Contato contato = new Contato();
10             contato.setNome(rs.getString("nome"));
11             contato.setEmail(rs.getString("email"));
12             contato.setEndereco(rs.getString("endereco"));
13
14             // montando a data através do Calendar
15             Calendar data = Calendar.getInstance();
16             data.setTime(rs.getDate("dataNascimento"));
17             contato.setDataNascimento(data);
18
19             // adicionando o objeto à lista
20             contatos.add(contato);
21         }
22         rs.close();
23         stmt.close();
24         return contatos;
25     } catch (SQLException e) {
26         throw new RuntimeException(e);
27     }
28 }
```

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

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

   a) Crie um ContatoDAO:

   ```
   ContatoDAO dao = new ContatoDAO();
   ```

   b) Liste os contatos com o DAO:

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

   c) 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 da direita no mesmo, *Run*, *Run as Java Application* (aproveite para aprender a tecla de atalho para executar a aplicação).

2.14 - Um pouco mais...

1) 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!

2) O Hibernate tomou conta do mercado e virou febre mundial pois não se faz necessário escrever uma linha de código SQL!

3) Se um projeto não usa nenhuma das tecnologias de ORM (**O**bject **R**elational **M**apping) 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:

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

```
15     }
16 }
```

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

Agora 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:

```
1  public void remove(Contato contato) {
2      try {
3          PreparedStatement stmt = connection.prepareStatement("delete from contatos where id=?");
4          stmt.setLong(1, contato.getId());
5          stmt.execute();
6          stmt.close();
7      } catch (SQLException e) {
8          throw new RuntimeException(e);
9      }
10 }
```

2.17 - Exercícios opcionais - Alterar e remover

1) Adicione o método para alterar contato no seu `ContatoDAO`.

```
1  public void altera(Contato contato) {
2      String sql = "update contatos set nome=?, email=?, endereco=?, dataNascimento=? where id=?";
3
4      try {
5          PreparedStatement stmt = connection.prepareStatement(sql);
6          stmt.setString(1, contato.getNome());
7          stmt.setString(2, contato.getEmail());
8          stmt.setString(3, contato.getEndereco());
9          stmt.setDate(4, new Date(contato.getDataNascimento().getTimeInMillis()));
10         stmt.setLong(5, contato.getId());
11         stmt.execute();
12         stmt.close();
13     } catch (SQLException e) {
14         throw new RuntimeException(e);
15     }
16 }
```

2) Adicione o método para remover contato no seu `ContatoDAO`

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

```
 9     }
10 }
```

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.

CAPÍTULO 3

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 infraestrutra.

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 Autenthication 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 6, lançada em 10 de dezembro de 2009. É uma versão ainda muito recente, com poucas ferramentas e servidores disponíveis. A versão mais usada no mercado é a versão 5, de 2006. Este curso é focado na versão 5 que você encontrará no mercado e já apresentando as novidades do novo Java EE 6 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 e o Hibernate (muito próximo ao JPA). 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).

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 auxilia-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 referencia 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:

- RedHat, JBoss Application Server, gratuito, Java EE 5;
- Sun, GlassFish, gratuito, Java EE 6.
- Apache, Apache Geronimo, gratuito, Java EE 5;
- Oracle/BEA, WebLogic Application Server, Java EE 5;
- IBM, IBM Websphere Application Server, Java EE 5;
- Sun, Sun Java System Application Server (baseado no GlassFish), Java EE 5;
- SAP, SAP Application Server, Java EE 5;

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:

- JSP
- Servlets
- JSTL
- JSF

Um **Servlet Container** é um servidor que suporta essas funcionalidades, mas não necessariamente 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 se encaixa nessa categoria).

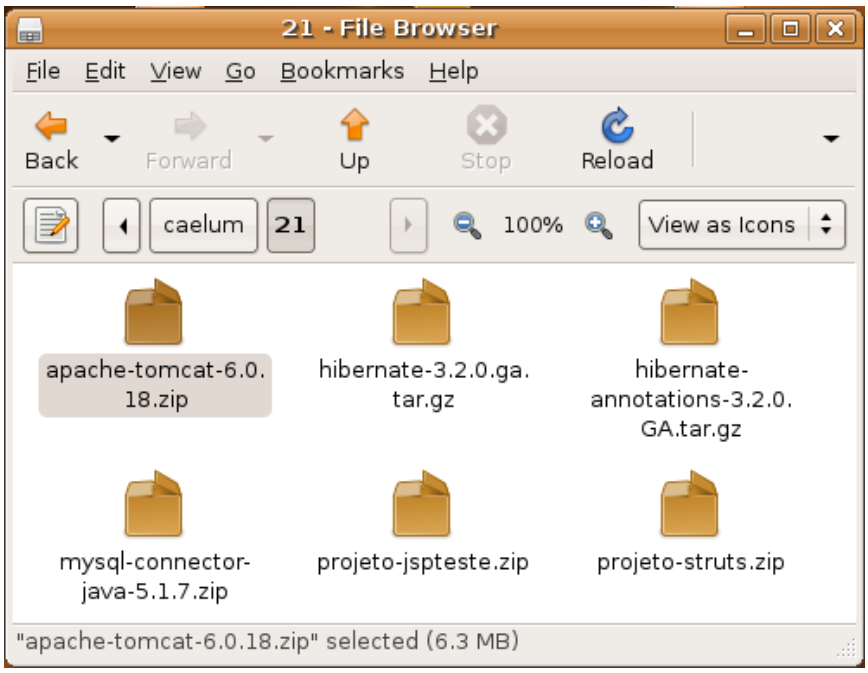
Há alguns servlet containers famosos no mercado. O mais famoso é o Apache Tomcat, mas há outros como o Jetty, que nós da Caelum usamos muito em projetos e o Google usa em seu cloud Google App Engine.

3.5 - Preparando o Tomcat

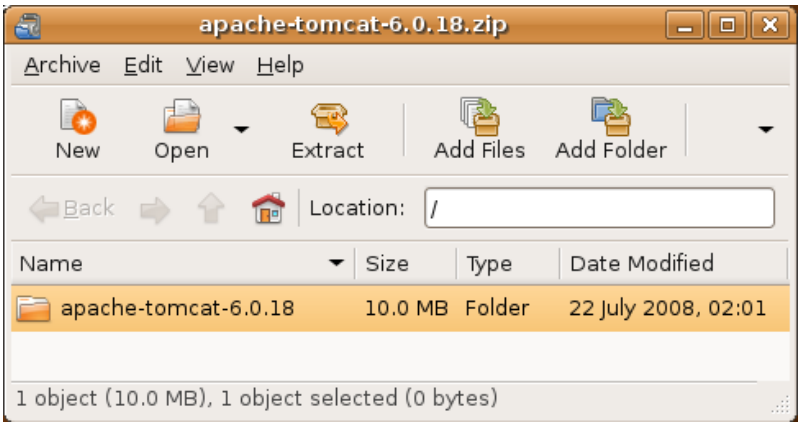
Para preparar o Tomcat na Caelum, siga os seguintes passos:

1) Entre no atalho **caelum** no Desktop;

2) Entre na pasta **21** e selecione o arquivo do **apache-tomcat**;

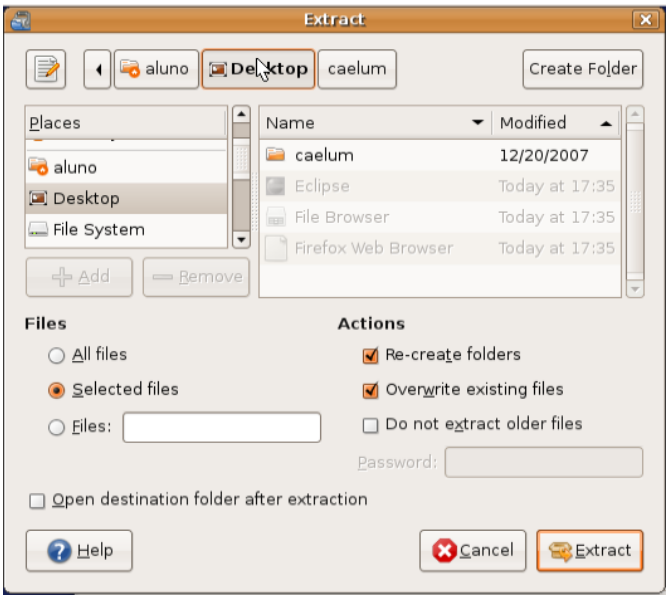


3) Dê dois cliques para abrir o Archive Manager do Linux;

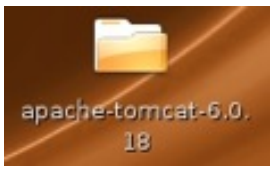


4) Clique em **Extract**;

5) Escolha o seu **Desktop** e clique em extract;



6) O resultado é uma pasta chamada **apache-tomcat**: o tomcat já está instalado.



3.6 - Preparando o Tomcat em casa

Baixe o Tomcat 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 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, idéias 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.

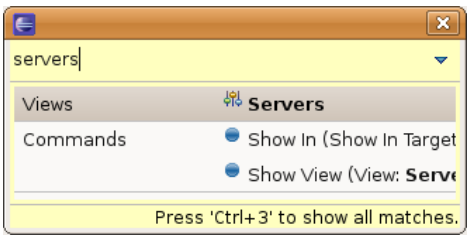
Para instalar o eclipse com WTP basta ir no site do Eclipse e:

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

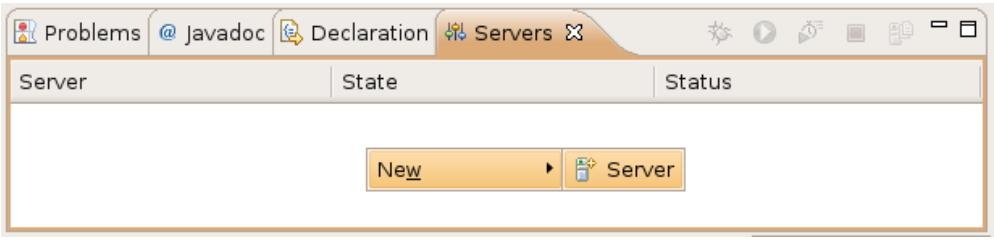
3.10 - Configurando o Tomcat no WTP

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

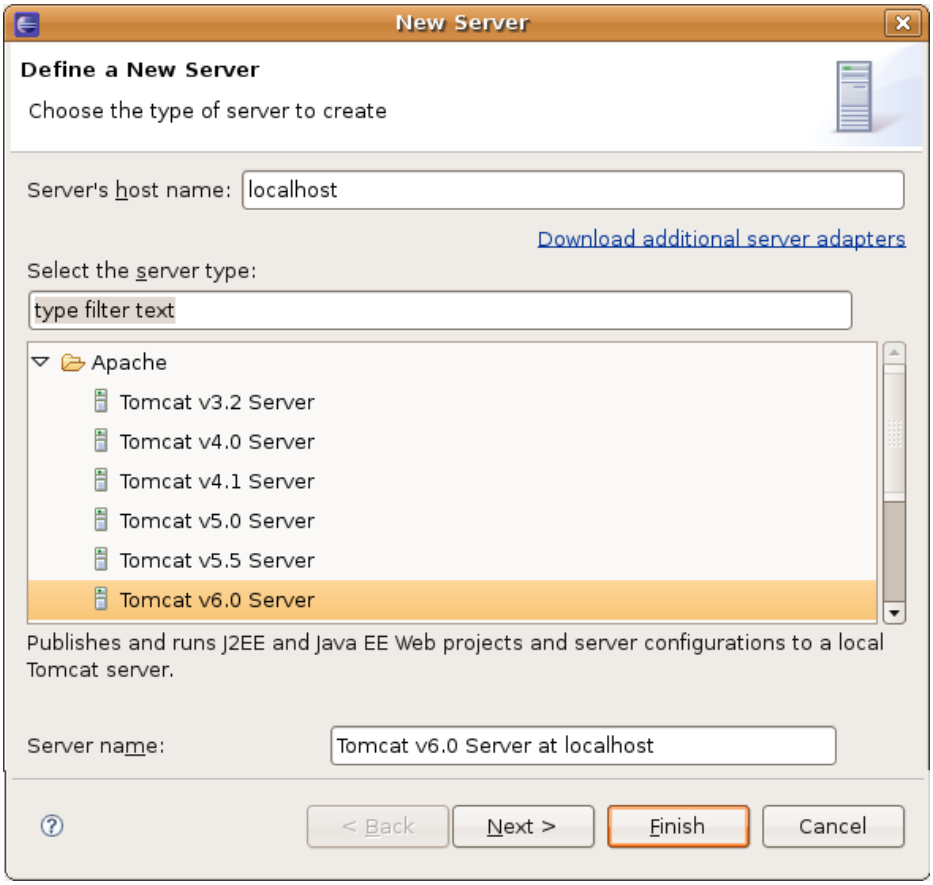
1) Mude a perspectiva do Eclipse para **Java** (e não Java EE, por enquanto). Para isso, vá no canto direito superior e selecione **Java**;
2) Abra a *View* de **Servers** na perspectiva atual. Aperte **Ctrl + 3** e digite **Servers**:



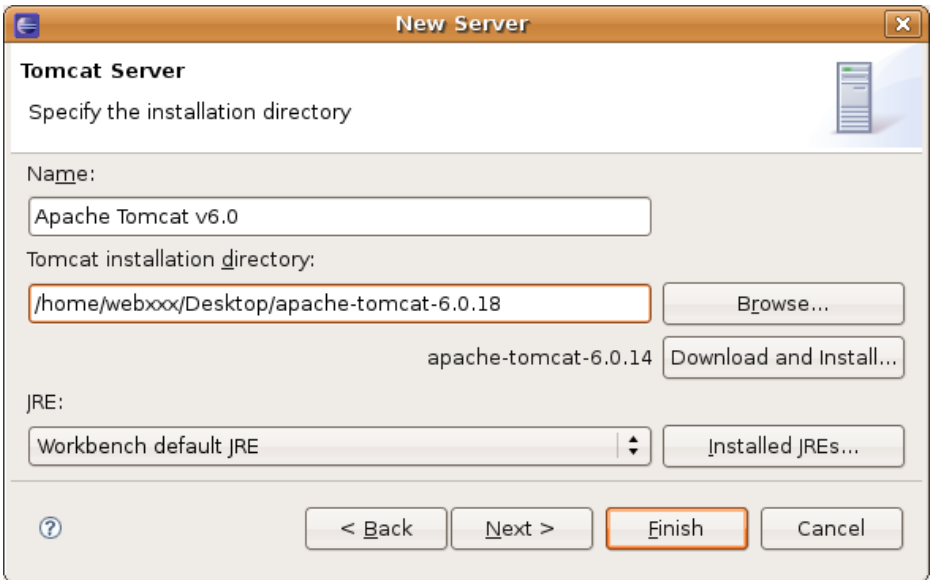
3) Clique com o botão direito dentro da aba Servers e vá em **New** > **Server**:



4) Selecione o **Apache Tomcat 6.0** e clique em **Next**:



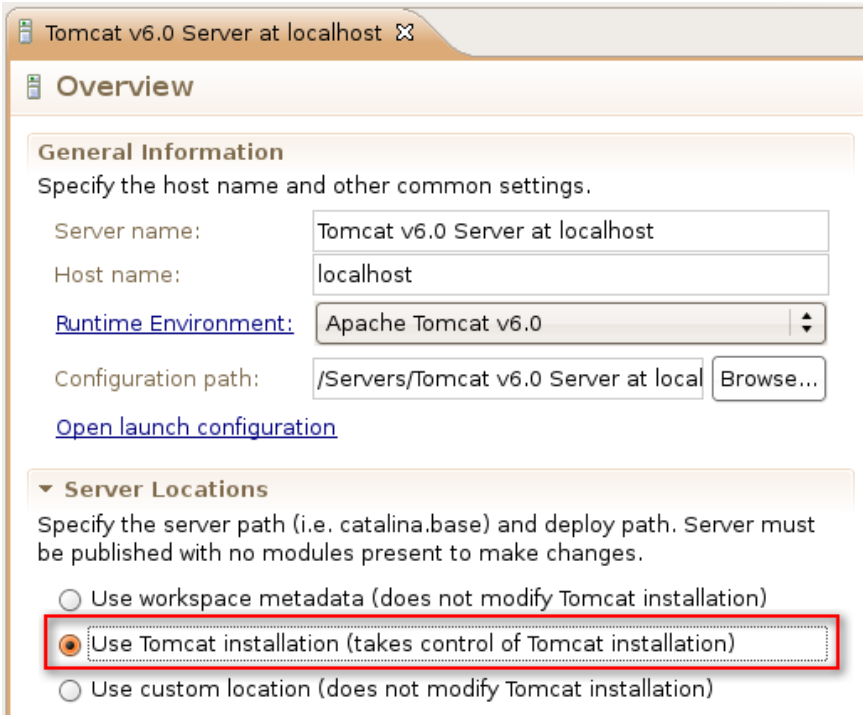
5) Na próxima tela, selecione o diretório onde você descompactou o Tomcat e clique em **Finish**:



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.

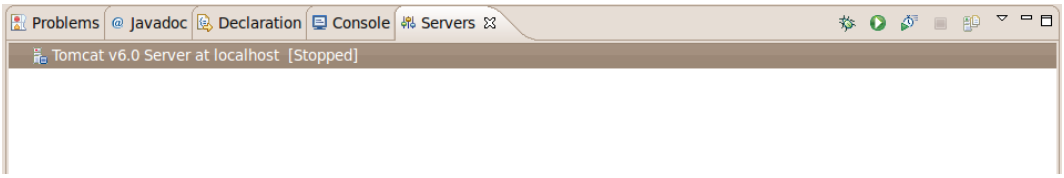
   Na aba Servers, dê dois cliques no servidor Tomcat que uma tela de configuração se abrirá. Localize a

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!

CAPÍTULO 4

Novo projeto Web usando Eclipse

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

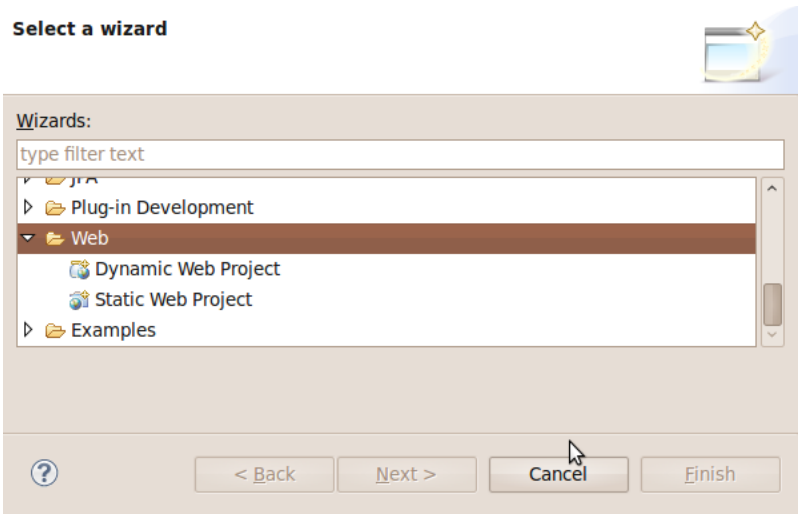
Ao término desse capítulo, você será capaz de:

- 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.

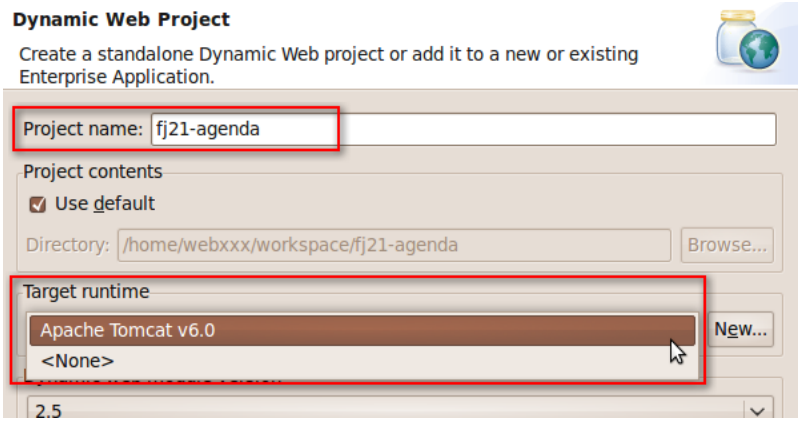
4.1 - Novo projeto

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:

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



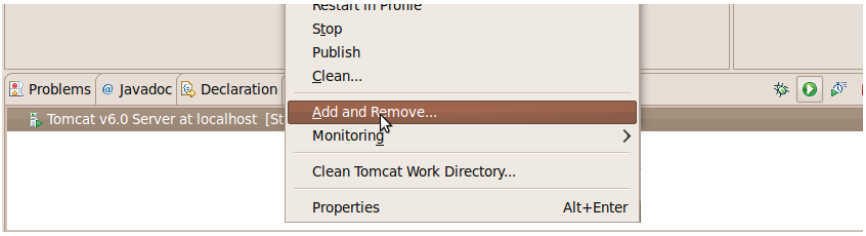
2) Coloque o nome do projeto como **fj21-agenda** e selecione como *Runtime Environment* a versão do Tomcat que acabamos de configurar:



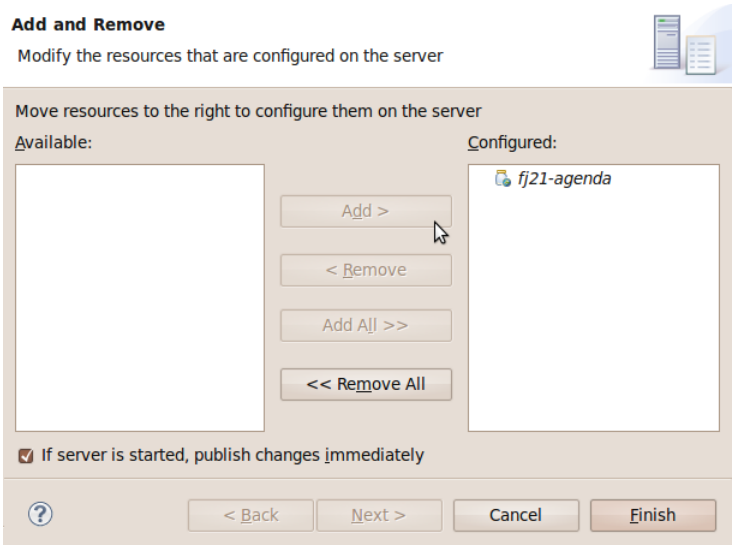
3) Clique em **Finish**. Se for perguntado sobre a mudança para a perspectiva Java EE, selecione **Não**.

O último passo é configurar o projeto para rodar no Tomcat que configuramos:

1) Na aba **Servers**, clique com o botão direito no Tomcat e vá em **Add and Remove...**:



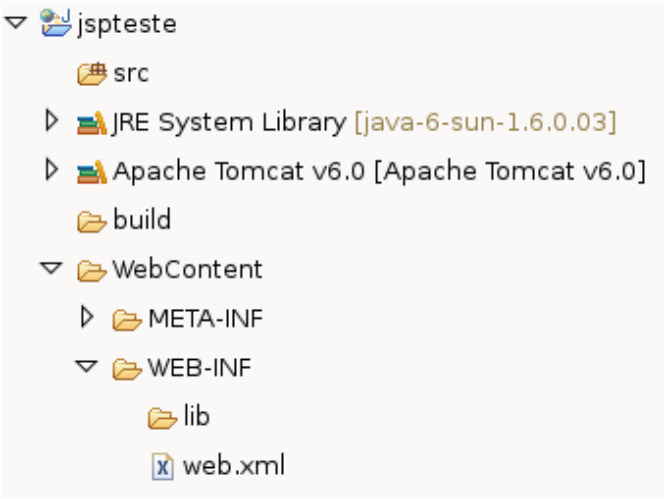
2) Agora basta selecionar o nosso projeto fj21-agenda e clicar em **Add**:



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

4.2 - 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
           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.

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`.

4.3 - Criando nossas páginas e HTML Básico

Para criarmos as nossas páginas, precisamos utilizar uma linguagem que consiga ser interpretada pelos nevagadores. Essa linguagem é a HTML (**H**yper**t**ext **M**arkup **L**anguage).

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>
```

4.4 - 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>
        Primeira página do projeto fj21-agenda
    </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: http://localhost:8080/fj21-agenda/

4.5 - Para saber mais: configurando o Tomcat sem o plugin

Se fosse o caso de criar uma aplicação web sem utilizar o plugin do tomcat deveríamos criar um arquivo de extensão xml com o nome de sua aplicação no diretório **tomcat/conf/Catalina/localhost**.

Para isso teríamos que configurar a url **/fj21-agenda** para o diretório **/home/usuario/workspace/fj21-agenda/WebContent/**. Queremos também permitir que o Tomcat faça o *restart* de sua aplicação sempre que julgar necessário.

1) Abra os seus diretórios;

2) Vá para o diretório **tomcat**;

3) Escolha o diretório **conf/Catalina/localhost**;

4) Crie um arquivo chamado **fj21-agenda.xml**;

5) Escreva o código a seguir no seu arquivo:

```
<Context path="/fj21-agenda"
         docBase="/home/usuario/workspace/fj21-agenda/WebContent/" reloadable="true" />
```

Importante! Não esqueça de trocar a palavra “usuario” pelo nome do seu usuário.

4.6 - 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 mais comuns tags da HTML:

- `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;
- `p`: define que o texto dentro dela estará em um parágrafo;
- `h1` - `h6`: define cabeçalhos, do `h1` ao `h6`, onde menor o número, maior o tamanho do texto;
- `a`: cria um link para outra página, ou para algum ponto da mesma página;

Para um tutorial completo sobre HTML, recomendamos uma visita ao site da w3schools, que possui referências das tags e exemplos de uso: http://www.w3schools.com/html/

CAPÍTULO 5

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 enviador 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, mas que ainda não tem adoção no mercado.

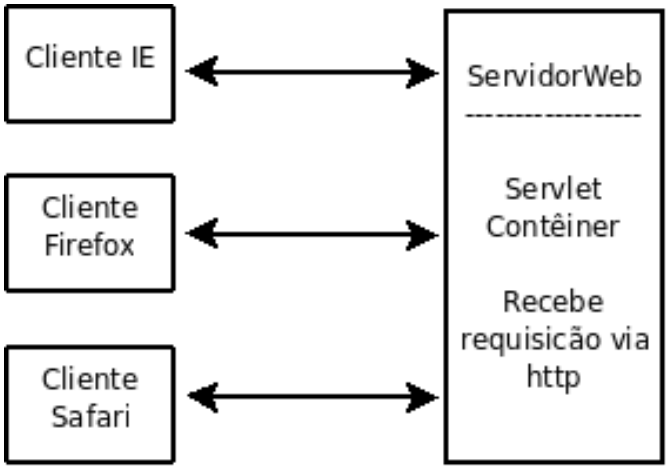
5.2 - Servlets

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 iremos 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 agora 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:

```
1 public class OiMundo extends HttpServlet {
2     protected void service (HttpServletRequest request, HttpServletResponse response)
3                         throws ServletException, IOException {
4         PrintWriter out = response.getWriter();
5
6         // escreve o texto
7         out.println("<html>");
8         out.println("<body>");
9         out.println("Primeira servlet");
10        out.println("</body>");
11        out.println("</html>");
12    }
13 }
```

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 reinstanciada;
- 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 iremos 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:

1) Definir o nome e classe da servlet;

2) Usando o nome da servlet, definir a URL.

Agora a servlet pode ser acessada através da seguinte URL:

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

Assim que o arquivo `web.xml` e a classe de 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`:

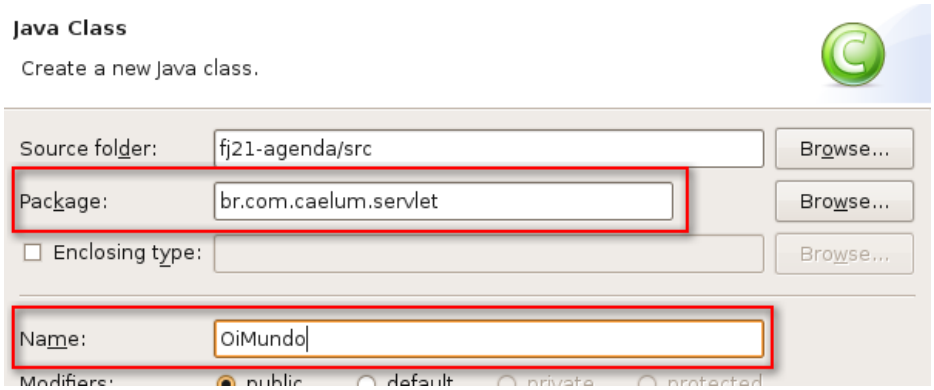
```
<servlet-mapping>
    <servlet-name>primeiraServlet</servlet-name>
    <url-pattern>*.php</url-pattern>
</servlet-mapping>
```

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).



a) Estenda `HttpServlet`:

```
public class OiMundo extends HttpServlet {

}
```

b) Utilize o CTRL+SHIFT+O para importar `HttpServlet`.

c) Para escrever a estrutura do método `service`, dentro da classe, escreva apenas **service** e dê **Ctrl+espaço**: o Eclipse gera pra você o método.

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 irá reclamar e você vai perceber o erro ainda em tempo de compilação.
O método gerado deve ser esse. **Troque os nomes dos parâmetros como abaixo.**

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

}
```

d) 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.

```
1  protected void service(HttpServletRequest request,
2          HttpServletResponse response) throws ServletException, IOException {
3      PrintWriter out = response.getWriter();
4
5      // escreve o texto
6      out.println("<html>");
7      out.println("<body>");
8      out.println("Oi mundo!");
9      out.println("</body>");
10     out.println("</html>");
11 }
```

2) Abra o arquivo **web.xml** e mapeie a URL /**oi** para a servlet `OiMundo`. Aproveite o auto-completar 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.

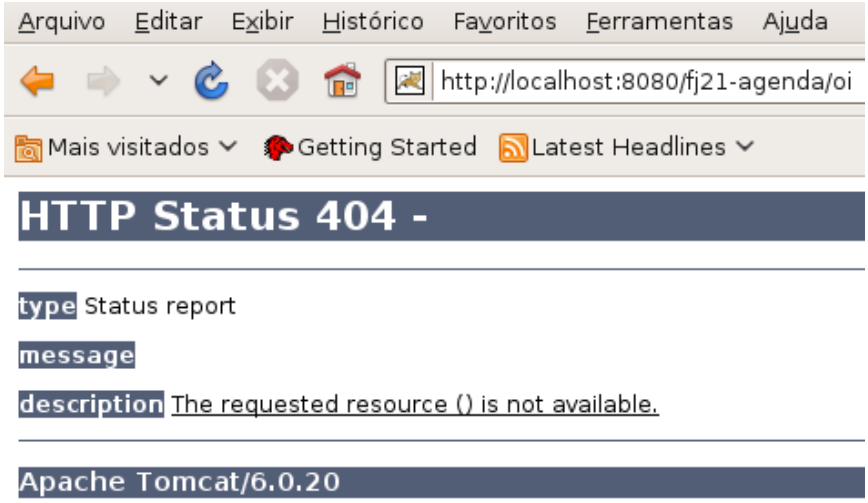
4) Teste a url http://localhost:8080/fj21-agenda/oi

5.6 - Erros comuns

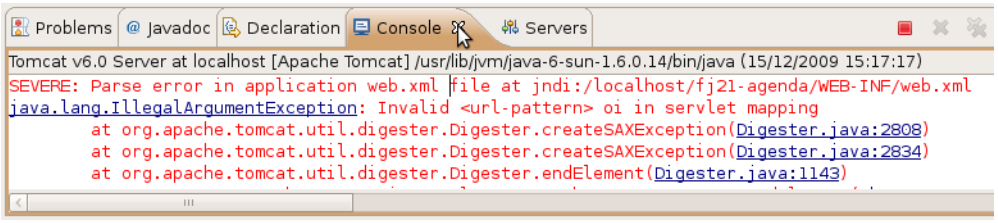
Existem diversos erros comuns nos exercícios anteriores. Aqui vão alguns deles:

1) Esquecer da barra inicial no URL pattern:

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

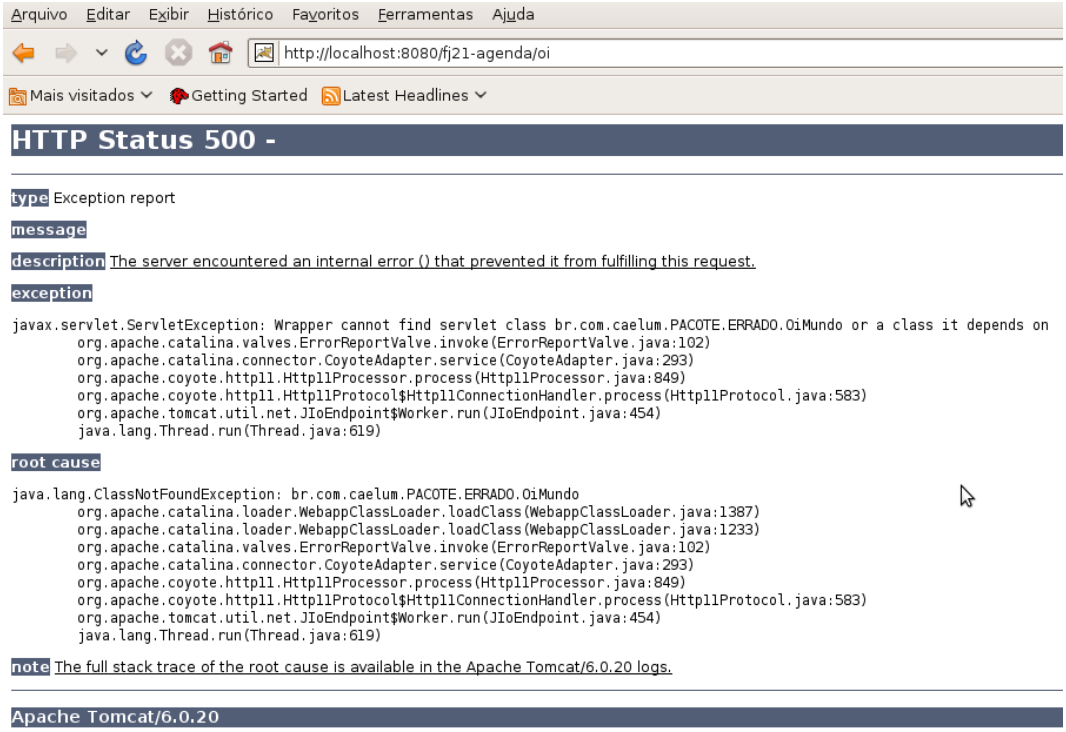


Nesse caso, uma exceção acontecerá no momento em que o tomcat for inicializado:



2) Digitar errado o nome do pacote da sua servlet:

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



3) Esquecer de colocar o nome da classe no mapeamento da servlet:

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

5.7 - Enviando parâmetros na requisição

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

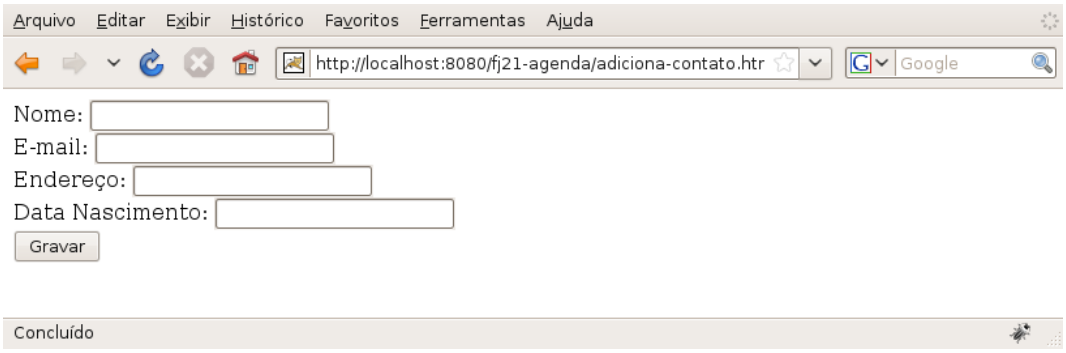
Por exemplo, para gravarmos um usuário no banco de dados, precisamos do nome, data de nascimento, e-mail e o endereço dele. Temos uma página com um formulário que o usuário possa preencher e ao clicar em um botão esses dados devem, de alguma forma, ser passados para uma servlet. Já sabemos que a servlet responde por uma determinada URL (através do *url-pattern*), 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:



5.8 - 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 parâmtro `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 irá 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`.

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

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

```
1  public class AdicionaContatoServlet extends HttpServlet {
2      protected void service(HttpServletRequest request, HttpServletResponse response)
3                           throws IOException, ServletException {
4
5          PrintWriter out = response.getWriter();
6
7          // pegando os parâmetros do request
8          String nome = request.getParameter("nome");
9          String endereco = request.getParameter("endereco");
10         String email = request.getParameter("email");
11         String dataEmTexto = request.getParameter("dataNascimento");
12         Calendar dataNascimento = null;
13
14         // fazendo a conversão da data
15         try {
16             Date date = new SimpleDateFormat("dd/MM/yyyy").parse(dataEmTexto);
17             dataNascimento = Calendar.getInstance();
18             dataNascimento.setTime(date);
19         } catch (ParseException e) {
20             out.println("Erro de conversão da data");
21             return; //para a execução do método
```

```
22          }
23
24          // monta um objeto contato
25          Contato contato = new Contato();
26          contato.setNome(nome);
27          contato.setEndereco(endereco);
28          contato.setEmail(email);
29          contato.setDataNascimento(dataNascimento);
30
31          // salva o contato
32          ContatoDAO dao = new ContatoDAO();
33          dao.adiciona(contato);
34
35          // imprime o nome do contato que foi adicionado
36          out.println("<html>");
37          out.println("<body>");
38          out.println("Contato " + contato.getNome() + " adicionado com sucesso");
39          out.println("</body>");
40          out.println("</html>");
41      }
42  }
```

5.9 - Exercícios: Criando funcionalidade para gravar contatos

1) 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.

   a) No Eclipse, selecione o projeto **fj21-agenda** e vá no menu **File -> Import**

   b) Dentro da janela de Import, escolha **General -> Archive File** e clique em **Next**:

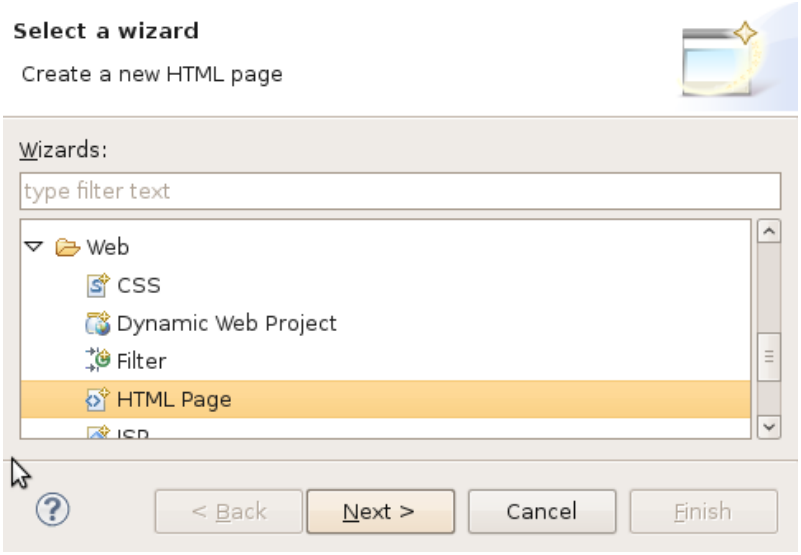
   c) No campo **From archive file** clique em **Browse**, selecione o arquivo **Desktop/caelum/21/dao-modelo.zip** e clique em **Finish**

> **Em casa**
>
> Caso você esteja fazendo em casa, você pode usar exatamente as mesmas classes criadas durante os exercícios do capítulo de JDBC. Não esqueça de copiar também o Driver do MySQL.

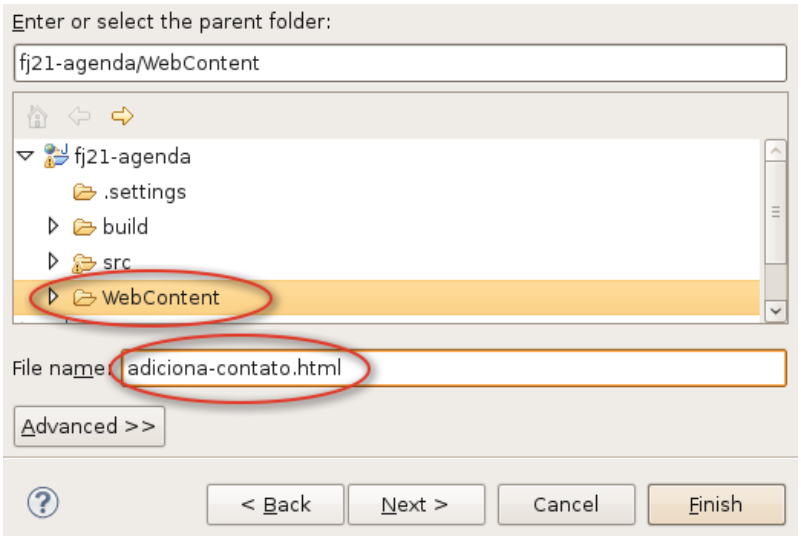
2) Temos agora que criar a página que permitirá aos usuários cadastrar os contatos

   a) Vá no menu **File -> New -> Other**.

   b) Escolha **Web -> HTML Page** e clique **Next**:



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



d) Esse arquivo HTML deverá ter o seguinte conteúdo:

```
<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>
```

e) Acesse no navegador o endereço:

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

3) Precisamos agora criar a `servlet` que gravará o contato no banco de dados:

a) Crie uma nova `Servlet` no pacote `br.com.caelum.agenda.servlet` chamada `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`.

```
1  public class AdicionaContatoServlet extends HttpServlet {
2
3      protected void service(HttpServletRequest request, HttpServletResponse response)
4                          throws IOException, ServletException {
5          // busca o writer
6          PrintWriter out = response.getWriter();
7
8          // buscando os parâmetros no request
9          String nome = request.getParameter("nome");
10         String endereco = request.getParameter("endereco");
11         String email = request.getParameter("email");
12         String dataEmTexto = request.getParameter("dataNascimento");
13         Calendar dataNascimento = null;
14
15         // fazendo a conversão da data
16         try {
17             Date date = new SimpleDateFormat("dd/MM/yyyy").parse(dataEmTexto);
18             dataNascimento = Calendar.getInstance();
19             dataNascimento.setTime(date);
20         } catch (ParseException e) {
21             out.println("Erro de conversão da data");
22             return; //para a execução do método
23         }
24
25         // monta um objeto contato
26         Contato contato = new Contato();
27         contato.setNome(nome);
28         contato.setEndereco(endereco);
29         contato.setEmail(email);
30         contato.setDataNascimento(dataNascimento);
31
32         // salva o contato
33         ContatoDAO dao = new ContatoDAO();
34         dao.adiciona(contato);
35
36         // imprime o nome do contato que foi adicionado
37
38         out.println("<html>");
39         out.println("<body>");
40         out.println("Contato " + contato.getNome() + " adicionado com sucesso");
41         out.println("</body>");
42         out.println("</html>");
43     }
}
```

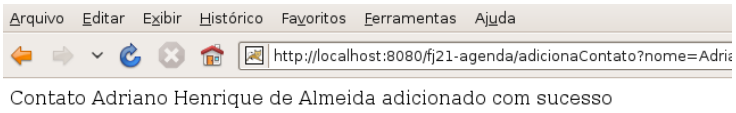
b) Mapeie-a no `web.xml`:

```
<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>
```

c) Reinicie o servidor, para que a nova Servlet seja reconhecida

d) Acesse novamente no navegador a URL http://localhost:8080/fj21-agenda/adiciona-contato.html

e) Preencha o formulário e clique em Gravar, o resultado deve ser semelhante com a imagem a seguir:



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

5.10 - 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 6 métodos: PUT, DELETE, HEAD, TRACE, CONNECT e OPTIONS.
Muitas pessoas conhecem apenas o GET e POST, pois, são os únicos que HTML 4 suporta.

5.11 - 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.

5.12 - Exercício: Tratando exceções e códigos HTTP

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

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

```
<html>
```

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

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

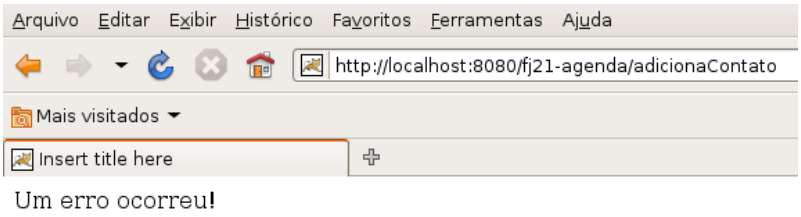
```
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/erro.html</location>
</error-page>
```

c) **Altere** o usuário de acesso ao banco na classe `ConnectionFactory` de `root` para algum outro usuário que não exista, por exemplo, `toor`.

d) Reinicie o servidor, para que as alterações tenham efeito

e) Acesse no navegador a URL http://localhost:8080/fj21-agenda/adiciona-contato.html

f) Preencha o formulário e clique em Gravar, o resultado deve ser semelhante com a imagem a seguir:



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

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

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

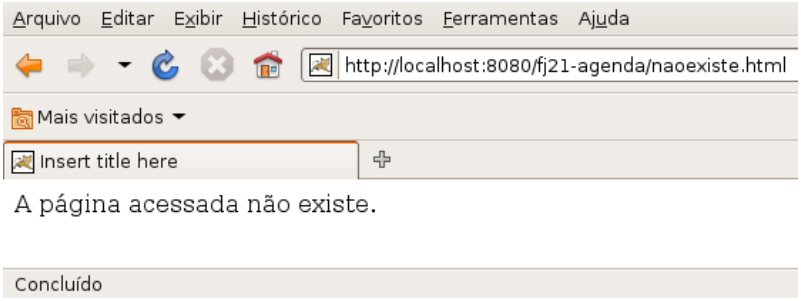
```
<html>
    <body>
        A página acessada não existe.
    </body>
</html>
```

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

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

c) Reinicie novamente o servidor;

d) Acesse no navegador uma URL inexistente no projeto, por exemplo, http://localhost:8080/fj21-agenda/naoexiste.html:



5.13 - Init e Destroy

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

É importante perceber que a sua Servlet será instanciada 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):

```
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
    }
}
```

5.14 - Uma única instância de cada servlet

De acordo com a especificação de servlets, por padrão, existe uma única instância de cada servlet declarada no `web.xml`. 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:

```
public class Contador extends HttpServlet {
    private int contador = 0; //variavel de instância

    protected void service(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {
        contador++; // a cada requisição a mesma variável 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. 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 concorrrência. 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 acessem 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**.

5.15 - 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.

5.16 - Para saber mais: Facilidades das Servlets 3.0

A última versão da API de Servlets, a 3.0, lançada em Dezembro de 2009 com o Java EE 6, traz algumas facilidades em relação às configurações das Servlets. De modo geral, não é mais preciso configurar no web.xml sendo suficiente usar a anotação `@WebServlet` apenas.

Veja alguns exemplos:

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

Ou ainda, com mais opções:

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

No final da apostila, você encontra um apêndice com as novidades do JavaEE 6, em especial o que há de novo na parte de Servlets 3.0.

5.17 - 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.

CAPÍTULO 6

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 **bem-vindo.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 implicitas 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 --%>
> ```

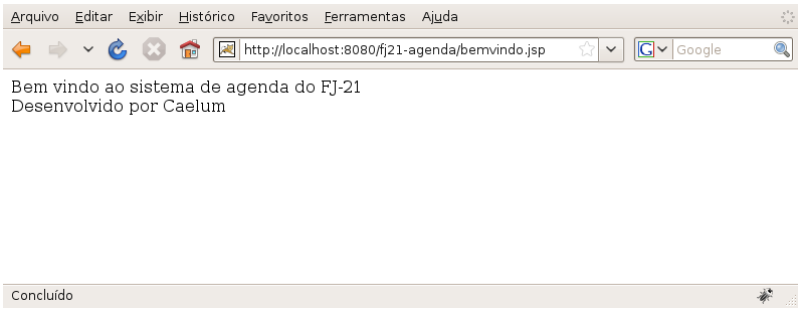
6.2 - Exercícios: Primeiro JSP

1) 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>
```

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



3) 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 %>`. Agora, só 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 serpará-los por vírgulas (vide o exercício).

6.4 - Exercícios: Lista de contatos com scriptlet

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

   a) Importe os pacotes necessários. Use o Ctrl+Espaço do WTP para ajudar a escrever os pacotes.

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

   b) 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>
```

c) 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) (Opcional) Coloque cabeçalhos para as colunas da tabela, descrevendo o que cada coluna significa.

6.5 - Exercícios opcionais

1) 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>
```

Reinicie o tomcat e acesse a URL: http://localhost:8080/fj21-agenda/

6.6 - 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, iremos evoluir nosso código até um ponto em que não faremos mais essa mistura.

6.7 - 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.

6.8 - Exercícios: parâmetros com a Expression Language

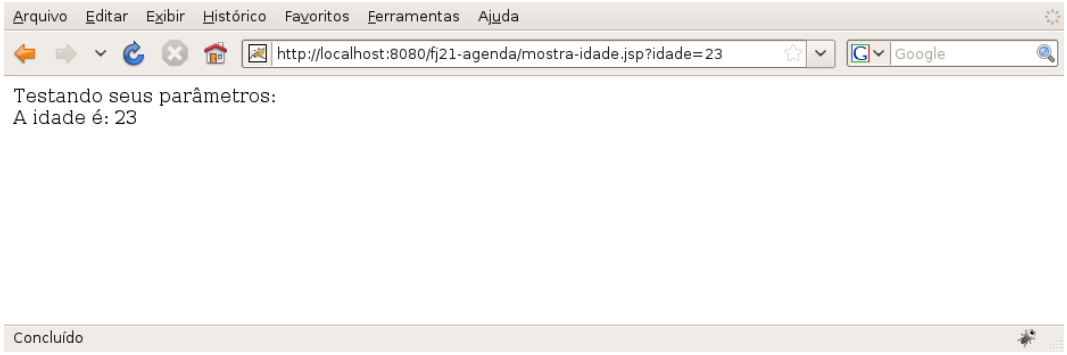
1) Crie uma página chamada **WebContent/digita-idade.jsp** com o conteúdo:

```
<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>

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

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



6.9 - 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 veremos adiante, 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.

CAPÍTULO 7

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.

E agora, instânciá-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"/>
```

Agora, 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 idéia 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 compreende 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 **https://jstl.dev.java.net/**.

Ao usar o JSTL em alguma página precisamos primeiro definir o cabeçalho. Existem quatro APIs básicas e iremos 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 iremos 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"/>
```

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

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

E agora 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.

   a) Primeiro, vá ao Desktop, e entre no diretório `Caelum/21/jstl`

   b) Haverá dois jars, jstl-impl-xx.jar e jstl-api-xx.jar

c) Copie-os (CTRL+C) e cole-os (CTRL+V) dentro de `workspace/fj21-agenda/WebContent/WEB-INF/lib`

d) No Eclipse, dê um F5 no seu projeto

> **Em casa**
>
> Caso você esteja em casa, pode fazer o download da JSTL API e da implementação em: http://jstl.dev.java.net/download.html

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

a) Crie o arquivo **WebContent/lista-contatos-elegante.jsp** usando o atalho de novo JSP no Eclipse;

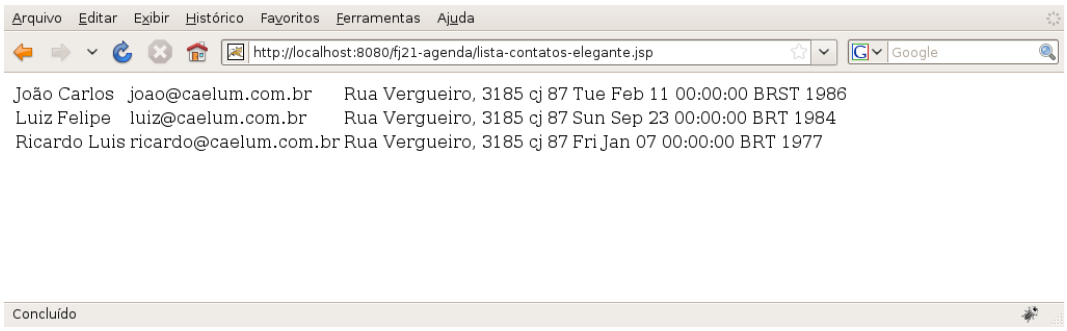
b) Antes de escrevermos nosso código, precisamos importar a taglib JSTL Core. Isso é feito com a diretiva `<%@ taglib %>` no topo do arquivo. Usando o recurso de auto-completar do Eclipse (inclusive na URL), declare a taglib no topo do arquivo:

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

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>
```

d) Acesse http://localhost:8080/fj21-agenda/lista-contatos-elegante.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 para o designer entender?

7.8 - Exercício opcional

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 o `box` imediatamente antes do exercício como auxílio)

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: Melhorando a lista de contatos com condicionais

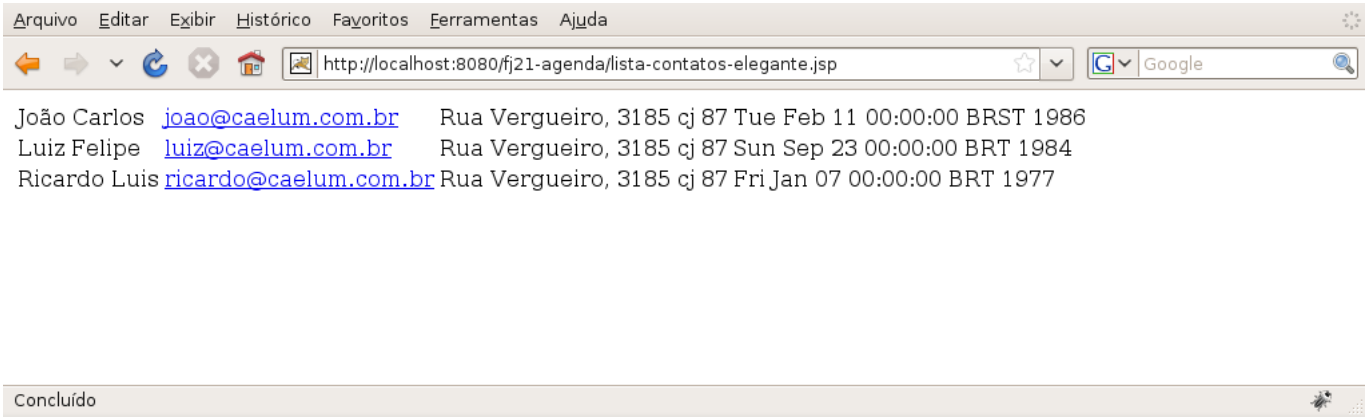
1) Vamos colocar em nossa listagem um link para envio de e-mail caso o mesmo tenha sido informado.

   a) Abra o arquivo `lista-contatos-elegante.jsp` no Eclipse;

   b) No momento de imprimir o e-mail do contato, adicione uma verificação para saber se o e-mail está pre-enchido 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>
```

   c) Acesse a página no navegador pelo endereço http://localhost:8080/fj21-agenda/lista-contatos-elegante.jsp



2) Agora vamos colocar a mensagem “E-mail não informado” caso o e-mail não tenha sido informado

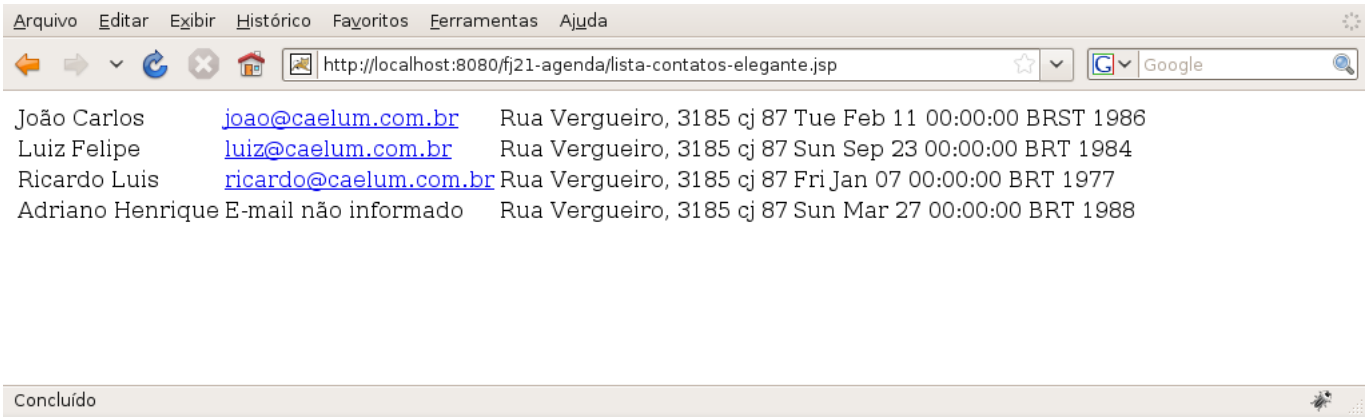
   a) Abaixo do novo `if` que fizemos no item anterior, vamos colocar mais um if, dessa vez com a verificação contrária, ou seja, queremos saber se está vazio:

```
<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>
```

b) Caso você não possua nenhum contato sem e-mail, cadastre algum agora.

c) Acesse a lista-contatos-elegante.jsp pelo navegador e veja o resultado;final



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`:

```
<html>
    <body>
        <img src="imagens/logomarca.jpg" /> Nome da empresa
```

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

```
Copyright 2010 - Todos os direitos reservados
    </body>
</html>
```

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

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

<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" />
```

7.13 - Exercícios: Adicionando cabeçalhos e rodapés

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

   a) Vá no Desktop, e entre em `Caelum/21/imagens` e copie esse diretório 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

   b) Crie dentro de `WebContent` um arquivo chamado `cabecalho.jsp` com o começo do nosso HTML e o logotipo do sistema:

```
<html>
    <body>
        <img src="imagens/caelum.png" />
        <h2>Agenda de Contatos do(a) (Seu nome aqui)</h2>
        <hr />
```

   c) Crie também a página `rodape.jsp`:

```
<hr />
        Copyright 2010 - Todos os direitos reservados
    </body>
</html>
```

d) Podemos agora importar as duas páginas (`cabecalho.jsp` e `rodape.jsp`), dentro da nossa `lista-contatos-elegante.jsp` usando a tag `<c:import/>` que vimos:

```
<%@ 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>

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

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

e) Repare que não temos mais a abertura do `<html>` nem o fechamento na `lista-contatos-elegante.jsp`. Isso porque as páginas que estão sendo inclúidas já estão fazendo esse serviço.

f) Visualize o resultado final acessando no navegador http://localhost:8080/fj21-agenda/lista-contatos-elegante.jsp

7.14 - Formatando as 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.

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

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

b) Troque a Expression Language que mostra a data de nascimento para passar pela tag `<fmt:formatDate/>`:

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

c) Acesse a página pelo navegador e veja o resultado final:

7.16 - Para saber mais: Trabalhando com links com <c:url>

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 `<img>` 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`. Nosso 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"/>
<img src="${imagem}"/>
```

Ou de uma forma ainda mais simples:

```
<img src="<c:url value="/imagens/caelum.png"/>"/>
```

O HTML gerado pelo exemplo seria: .

7.17 - Exercícios opcionais: Caminho absoluto

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

```
<img src="<c:url value="/imagens/caelum.png"/>" />
```

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)

b) Visualize o resultado acessando no navegador http://localhost:8080/fj21-agenda/lista-contatos-elegante.jsp

c) 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

Confira o uso de algumas delas no apêndice final de Servlet API.

CAPÍTULO 8

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 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 type="text/javascript">
    $("#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" type="text">

<script type="text/javascript">
        $("#dataNascimento").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 agora temos a capacidade de receber um valor quando formos usar a tag. Imagine usar essa tag nova no nosso formulário:

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

Agora 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 type="text/javascript">
    $("#${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 type="text/javascript" src="js/arquivo.js"></script>
</head>
```

Para saber mais sobre Javascript, recomendamos que acesse o site: http://www.w3schools.com/js

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

Para saber mais sobre *CSS*, recomendamos que acesse o site: http://www.w3schools.com/css/

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.

   a) Vá ao Desktop, e entre em `Caelum/21`;

   b) Copie os diretórios `js` e `css` e cole-os dentro de `WebContent` no seu projeto;

Em casa

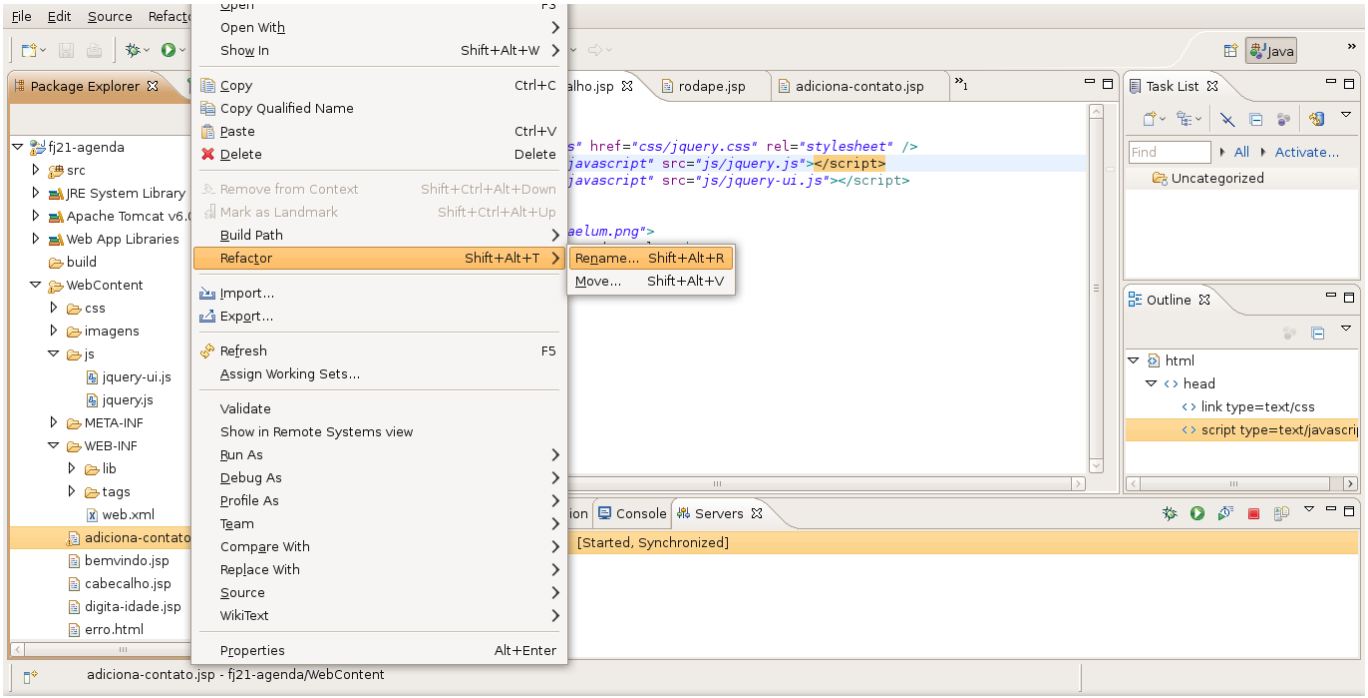
Se você estiver fazendo de casa, pode achar esses dois arquivos no pacote do JQueryUI em: http://jqueryui.com/download

   c) Precisamos importar o JQuery agora nos nossos arquivos. Para isso, no `cabecalho.jsp`, adicione dentro a tag `html`, dessa forma, nossas páginas que importarem `cabecalho.jsp` terão automaticamente acesso aos estilos *CSS* e aos *Javascripts*:

```
<html>
    <head>
        <link type="text/css" href="css/jquery.css" rel="stylesheet" />
        <script type="text/javascript" src="js/jquery.js"></script>
        <script type="text/javascript" src="js/jquery-ui.js"></script>
    </head>
    <body>

    <!-- Restante do cabeçalho aqui -->
```

2) a) Agora temos que importar nosso `cabecalho.jsp` na página de adicionar contatos, a `adiciona-contato.html`. Mas temos um problema. Não podemos utilizar a `c:import` em uma página `HTML`. Para resolvermos isso, vamos trocar sua extensão para `.jsp`

   b) 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 agora se chamará `adiciona-contato.jsp`.



c) Agora podemos importar o cabeçalho e o rodapé também nessa página, portanto, remove a declaração `<html>` e `<body>` do começo e do fim do arquivo e inclua:

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

    <!-- Aqui continua o formulário com os campos -->

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

Cuidado para não apagar o formulário, precisaremos dele mais para frente.

d) Acesse agora http://localhost:8080/fj21-agenda/adiciona-contato.jsp e verifique o resultado com o cabeçalho e rodapé.

3) Vamos criar a nossa tag para o calendário.

a) Dentro de `WEB-INF` crie um diretório chamado `tags`.

b) Agora crie um arquivo chamado **campoData.tag** dentro de `WEB-INF/tags/`:

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

<input type="text" id="${id}" name="${id}" />
<script type="text/javascript">
    $("#${id}").datepicker({dateFormat: 'dd/mm/yyyy'});
</script>
```

c) 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" %>
```

d) Agora, 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>
```

e) 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.

f) Verifique o código fonte gerado, e repare que o *Javascript* do calendário e o input foram gerados para nós através de nossa Taglib.

4) (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-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.

CAPÍTULO 9

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 agora 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>");

}
```

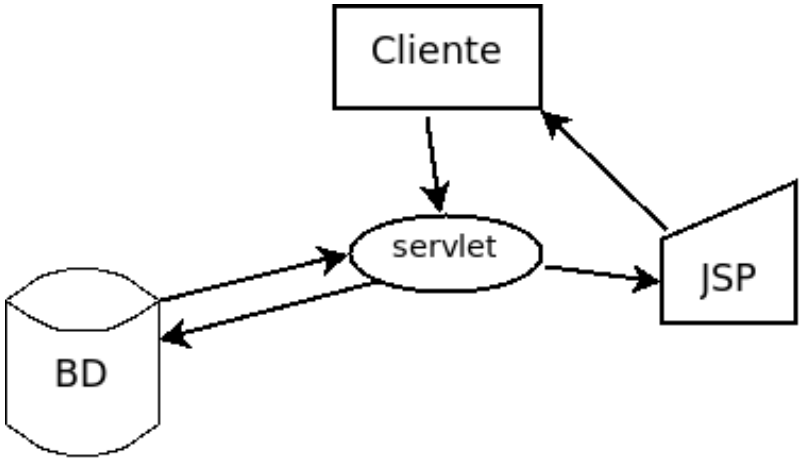
Repare que, no final do nosso método, misturamos o código HTML com Java. O que queremos extrair do código acima é justamente essas últimas linhas.

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);
```

Agora, 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 agora 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*:

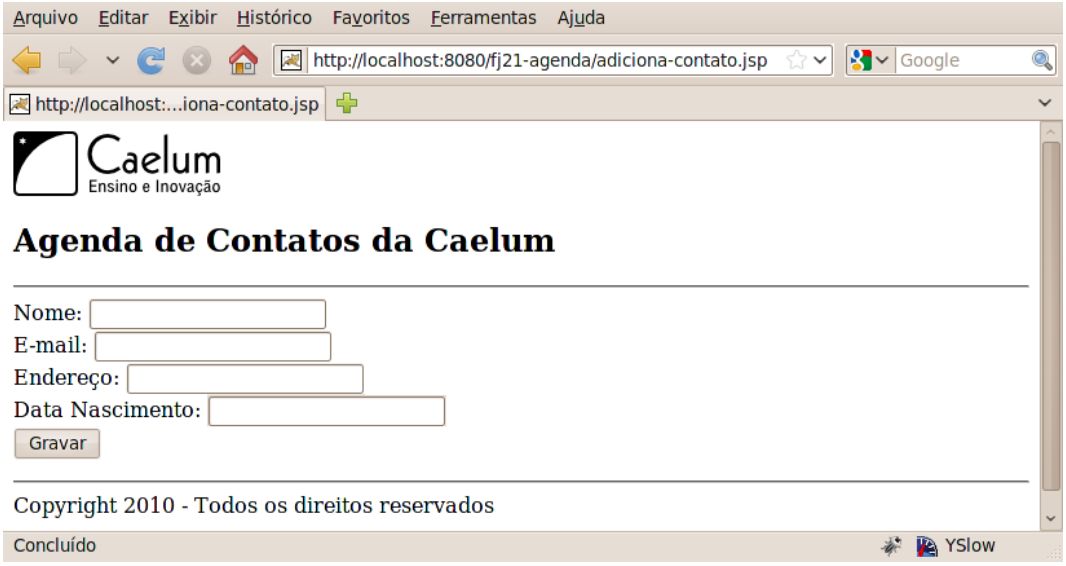
```
<html>
    <body>
        Contato ${param.nome} adicionado com sucesso
    </body>
</html>
```

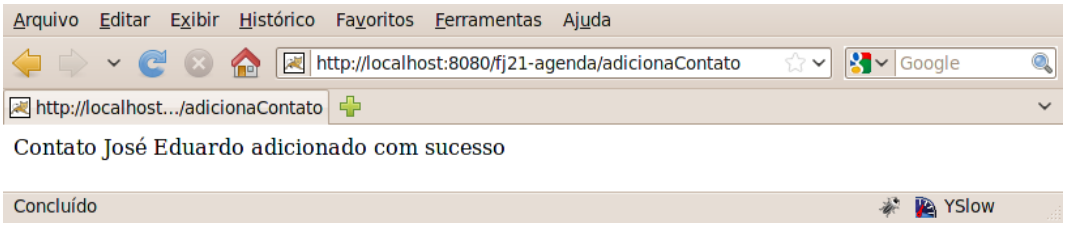
2) **Altere** sua servlet `AdicionaContatoServlet` para que, após a execução da lógica de negócios, o fluxo da requisição seja redirecionado para nosso novo JSP.

   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





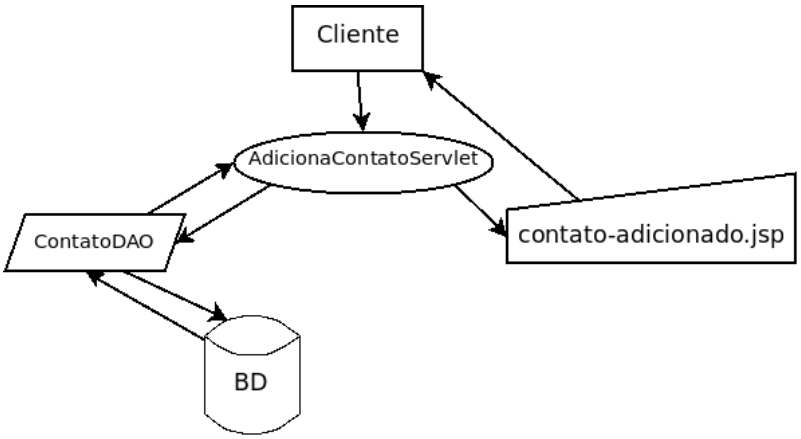
Resultado

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.



Agora, temos o problema de ter muitas servlets. Para cada lógica de negócios, teríamos uma servlet diferente, que significa oito linhas de código no `web.xml`... algo abominável em um projeto de verdade. Imagine dez classes de modelo, cinco lógicas diferentes, isso totaliza quatrocentas linhas de configuração.

Sabemos da existência de ferramentas para gerar tal código automaticamente, mas isso não resolve o problema da complexidade de administrar tantas servlets.

Utilizaremos uma idéia que diminuirá a configuração para apenas oito linhas: *colocar tudo em uma Servlet só* e, de acordo com que parâmetros o cliente nos chamar, decidimos o que executar. Teríamos aí uma Servlet para controlar essa parte, como o esboço abaixo:

```
// Todas as lógicas dentro de uma Servlet
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
            // dispacha para um jsp
        } else if (acao.equals("RemoveContato")) {
            // faz a remoção e redireciona para a lista
        }
    }
}
```

Configuraríamos essa Servlet no web.xml e 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 continuaríamos 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, `execute`) 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." + request.getParameter("logica");
Class classe = Class.forName(nomeDaClasse);
```

Ótimo, agora 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` de agora 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 `Logica`s 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;
}
```

E agora, podemos simplificar nossa `Servlet` para executar a lógica de forma polimórfica e, tudo aquilo que fazíamos em aproximadamente 8 linhas de código, agora 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.

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 codigo que isola a instanciação através de algum recurso diferente do construtor é uma **factory**.

9.6 - A configuração do web.xml

O único passo que falta para que tudo o que fizemos funcione é declarar a nossa única `Servlet` no `web.xml`. Note que agora temos **somente** 8 linhas de declaração no `web.xml`:

```
<web-app>
    <servlet>
        <servlet-name>controlador</servlet-name>
        <servlet-class>br.com.caelum.mvc.ControllerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>controlador</servlet-name>
        <url-pattern>/mvc</url-pattern>
    </servlet-mapping>
</web-app>
```

9.7 - Exercícios: Criando nossas lógicas e servlet de controle

1) Crie a sua interface no pacote `br.com.caelum.mvc.logica`:

```
public interface Logica {

    void executa(HttpServletRequest req, HttpServletResponse res) throws Exception;
}
```

2) Crie uma implementação da interface `Logica`, nossa classe `PrimeiraLogica`, também no pacote `br.com.caelum.mvc.logica`:

```
1 public class PrimeiraLogica implements Logica {

2     public void executa(HttpServletRequest req, HttpServletResponse res) throws Exception {
3         System.out.println("Executando a logica e redirecionando...");
4
5         RequestDispatcher rd = req.getRequestDispatcher("/primeira-logica.jsp");
```

```
6           rd.forward(req, res);
7       }
8   }
```

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) Agora vamos escrever nossa Servlet que coordenará o fluxo da nossa aplicação.
Crie sua servlet chamada `ControllerServlet` no pacote `br.com.caelum.mvc.servlet`:

```
1  public class ControllerServlet extends HttpServlet {

2      protected void service(HttpServletRequest request, HttpServletResponse response)
3                           throws ServletException, IOException {
4
5          String parametro = request.getParameter("logica");
6          String nomeDaClasse = "br.com.caelum.mvc.logica." + parametro;
7
8          try {
9              Class classe = Class.forName(nomeDaClasse);
10
11             Logica logica = (Logica) classe.newInstance();
12             logica.executa(request, response);
13
14         } catch (Exception e) {
15             throw new ServletException("A lógica de negócios causou uma exceção", e);
16         }
17     }
18 }
```

5) Mapeie a URL `/mvc` para essa servlet no arquivo `web.xml`.

```
<servlet>
    <servlet-name>controlador</servlet-name>
    <servlet-class>br.com.caelum.mvc.servlet.ControllerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>controlador</servlet-name>
    <url-pattern>/mvc</url-pattern>
</servlet-mapping>
```

6) Teste a url http://localhost:8080/fj21-agenda/mvc?logica=PrimeiraLogica

9.8 - Exercícios: Lógica para alterar contatos

1) Crie uma nova classe chamada `AlteraContatoLogic` no mesmo pacote `br.com.caelum.mvc.logica`. Devemos implementar a interface `Logica` e durante sua execução altere o contato no banco a partir do `id` indicado.

```
//import aqui CTRL + SHIFT + O


public class AlteraContatoLogic implements Logica {

    public void executa(HttpServletRequest request, HttpServletResponse response)
                                    throws Exception {

        Contato contato = new Contato();
        long id = Long.parseLong(request.getParameter("id"));
        contato.setId(id);
        contato.setNome(request.getParameter("nome"));
        contato.setEndereco(request.getParameter("endereco"));
        contato.setEmail(request.getParameter("email"));

        //Converte a data de String para Calendar
        String dataEmTexto = request.getParameter("dataNascimento");
        Date date = new SimpleDateFormat("dd/MM/yyyy").parse(dataEmTexto);
        Calendar dataNascimento = Calendar.getInstance();
        dataNascimento.setTime(date);

        contato.setDataNascimento(dataNascimento);

        ContatoDAO dao = new ContatoDAO();
        dao.atualiza(contato);

        RequestDispatcher rd = request.getRequestDispatcher("/lista-contatos-elegante.jsp");
        rd.forward(request, response);
        System.out.println("Alterando contato ..." + contato.getNome());
    }
}
```

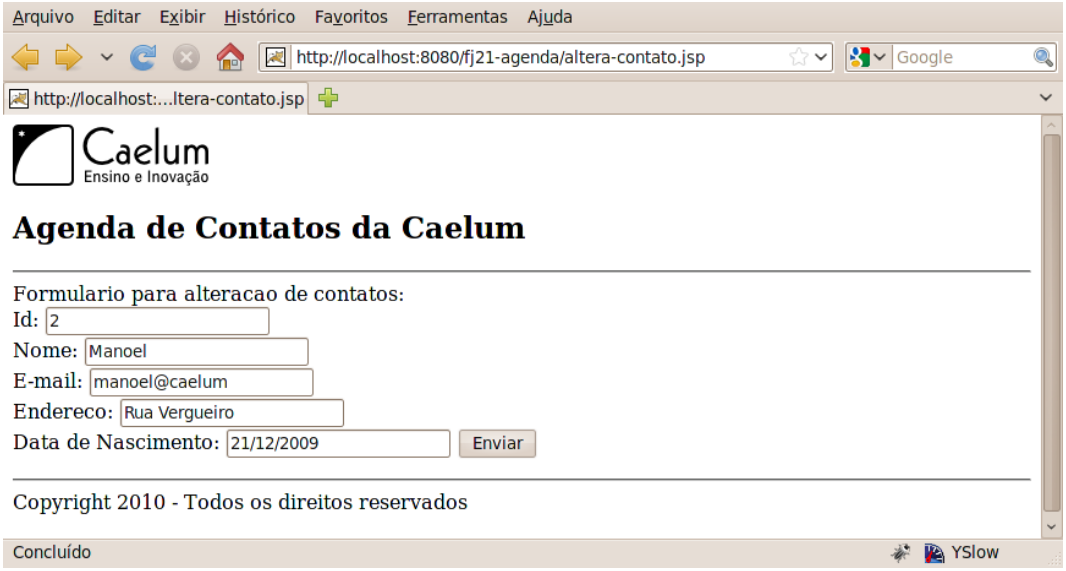
2) Crie uma nova página chamada `altera-contato.jsp` através do método `POST` que chama a lógica criada no exercício anterior.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib tagdir="/WEB-INF/tags" prefix="caelum" %>
    <c:import url="cabecalho.jsp" />
```

```
        Formulário para alteração de contatos:<br/>

        <form action="mvc" method="POST">
            Id:       <input type="text" name="id"/><br/>
            Nome:       <input type="text" name="nome"/><br/>
            E-mail:   <input type="text" name="email"/><br/>
            Endereço: <input type="text" name="endereco"/><br/>
            Data de Nascimento: <caelum:campoData id="dataNascimento" />
                      <input type="hidden" name="logica" value="AlteraContatoLogic"/>
                      <input type="submit" value="Enviar"/>
        </form>
<c:import url="rodape.jsp" />
```

Repare que passamos o parâmetro de qual lógica queremos executar através do `input` de tipo `hidden`. Isso cria um campo que não aparecerá na tela, mas também será um parâmetro enviado na requisição, ou seja, a `Servlet` controladora a receberá, para definir qual lógica será invocada.

3) Teste a url http://localhost:8080/fj21-agenda/altera-contato.jsp



9.9 - Exercícios opcionais

1) Crie uma lógica chamada `RemoveContatoLogic` e teste a mesma através de um link na listagem da `lista-contatos-elegante.jsp`.

2) Coloque um link na sua `lista-contatos-elegante.jsp` que abre a página `testa-altera-mvc.jsp` passando o Id do contato que você quer alterar. Deixe o campo Id visível no form mas não alterável. Não esqueça de passar o campo Id pela requisição. Faça com que os campos do form estejam populados com os dados do contato a ser editado.

3) 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-elegante.jsp`.

4) **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.

5) **Desafio**: Implemente outras facilidades no nosso framework. Por exemplo, ao invés de obrigar todas as Logicas a usarem o `RequestDispatcher`, faça o método `executa` devolver uma String com a página para onde redirecionar. E faça a Servlet controladora pegar esse retorno e executar o dispatcher.

9.10 - 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 dispatchs 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.

9.11 - Lista de tecnologias: camada de controle

Há diversas opções para a camada de controle no mercado. Veja um pouco sobre algumas delas:

1) **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.

2) **VRaptor 3** - desenvolvido inicialmente por profissionais da Caelum e baseado em diversas ideias dos controladores mencionados acima, o Vraptor 3 usa o conceito de favorecer Convenções em vez de Configurações para minimizar o uso de XML e anotações em sua aplicação Web.

3) **JSF** - JSF é uma especificação da Sun para frameworks MVC. Ele é baseado em componentes e possui várias facilidades para desenvolver a interface gráfica. Devido ao fato de ser um padrão da Sun ele é bastante adotado. O JSF é ensinado em detalhes no nosso curso FJ-26.

4) **JBoss Seam** - segue as idéias do Stripes na área de anotações e é desenvolvido pelo pessoal que trabalhou também no Hibernate. Trabalha muito bem com o Java Server Faces e EJB 3.

5) **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.

6) **Stripes** - um dos frameworks criados em 2005, abusa das anotações para facilitar a configuração.

7) **Wicket** - controlador baseado na idéia de que todas as suas telas deveriam ser criadas através de código Java. Essa linha de pensamento é famosa e existem diversos projetos similares, sendo que é comum ver código onde instanciamos um formulário, adicionamos botões, etc como se o trabalho estivesse sendo feito em uma aplicação `Swing`, mas na verdade é uma página html.

9.12 - 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 idéia 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 idéias 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 altenativa para as ferramentas anteriores mas sim uma maneira de criar templates para seu site, com uma idéia 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 irá 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.

9.13 - Discussão em aula: os padrões Command e Front Controller

CAPÍTULO 10

Recursos importantes: Filtros e WAR

"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;
- implantar sua aplicação em qualquer container.

10.1 - Reduzindo o acoplamento com Filtros

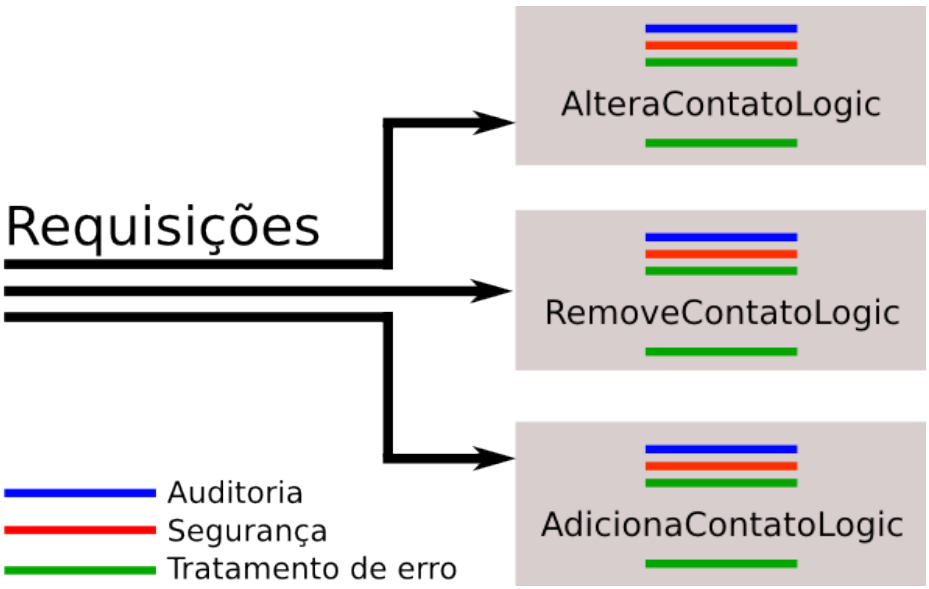
Em qualquer aplicação surgem requisitos que não são diretamente relacionados com a regra de negocios. Um exemplo clássico desses requisitos não funcionais é a auditoria (*Logging*). Queremos logar as chamadas de uma lógica da nossa aplicação. Outros exemplos são autorização, tratamento de erro ou criptografia. 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 pseudo-código as chamadas para fazer auditoria e autorização:

```
public class AlteraContatoLogic implements Logica {

  public void executa(HttpServletRequest request, HttpServletResponse response)
                            throws Exception {

    //auditoria
    Logger.info("acessando altera contato logic");

    //autorização
    if(!usuario.ehCliente()) {
        request.getRequestDispatcher("/acessoNegado.jsp").forward(request,response);
    }

    //toda lógica para alterar o contato aqui
    //...
  }
```

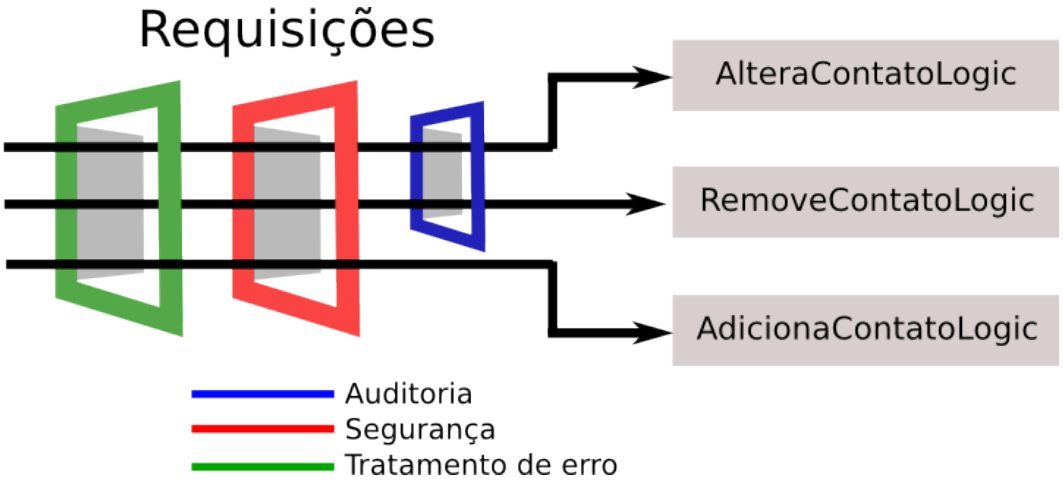
```
}
```

Podemos ver que além da lógica é preciso implementar os outros requistos, 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 responsibiliade, 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 um filtro, basta criarmos uma classe que implemente a interface `javax.servlet.Filter`. 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`.

```
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. Mais 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 através do `web.xml` declarando o filtro e quais URLs serão filtradas:

```
<filter>
    <filter-name>FiltroTempoDeExecucao</filter-name>
    <filter-class>br.com.caelum.agenda.filtro.FiltroTempoDeExecucao</filter-class>
</filter>

<filter-mapping>
    <filter-name>FiltroTempoDeExecucao</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Repare como usamos o *wildcard* no `<url-pattern>` para indicar que todas as requisições serão filtradas e, portanto, o filtro será aplicado em cada requisição.

É possível usar um `<url-pattern>` mais especifico. Por exemplo, podemos filtrar todas as requisições para paginas JSPs:

```
<url-pattern>*.jsp</url-pattern>
```

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.

   a) Crie uma nova classe chamada `FiltroTempoDeExecucao` no pacote `br.com.caelum.agenda.filtro` e faça ela implementar a interface `javax.servlet.Filter`

   b) Deixe os métodos `init` e `destroy` vazios e implemente o `doFilter`:

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
                throws IOException, ServletException {

    long tempoInicial = System.currentTimeMillis();

    chain.doFilter(request, response);

    long tempoFinal = System.currentTimeMillis();
    System.out.println("Tempo da requisição em millis: " + (tempoFinal - tempoInicial));
}
```

   c) Declare o filtro no `web.xml`, para fazer com que todas as requisições passem por ele:

```
<filter>
    <filter-name>cronometro</filter-name>
    <filter-class>br.com.caelum.agenda.filtro.FiltroTempoDeExecucao</filter-class>
</filter>
```

```
    <filter-mapping>
        <filter-name>cronometro</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

d) Reinicie o servidor e acesse **http://localhost:8080/fj21-agenda/altera-contato.jsp**
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 irá 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 idéia 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 iremos utilizar. Para isso, nossos `DAOs` agora 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 void 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();
    }
}
```

Isso já é uma grande evolução com relação ao que tínhamos no começo mas ainda não é uma solução muito boa. Agora, 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` agora 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.

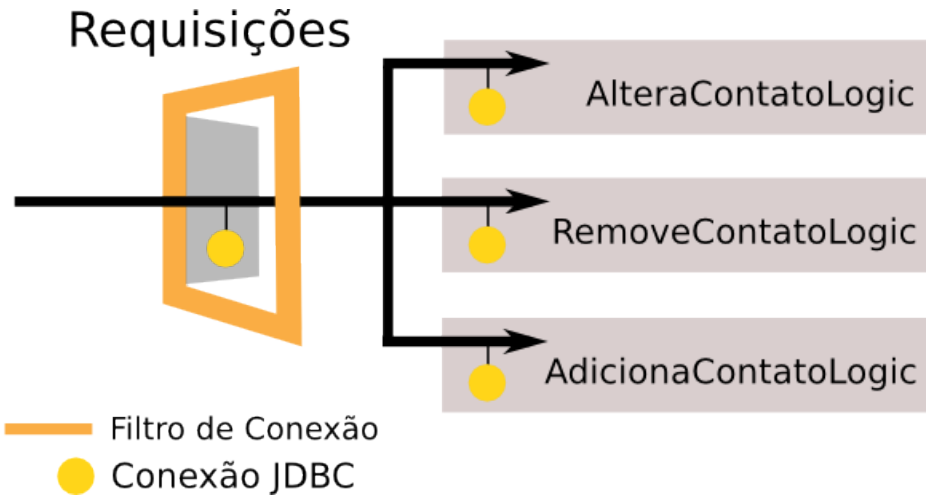
Vamos então implementar um filtro com esse comportamento:

```
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 apos 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 agora é o único ponto da nossa aplicação que criará conexões. 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 através do `web.xml` declarando o filtro e quais URLs serão filtradas:

```
<filter>
    <filter-name>FiltroConexao</filter-name>
    <filter-class>br.com.caelum.agenda.filtro.FiltroConexao</filter-class>
</filter>

<filter-mapping>
    <filter-name>FiltroConexao</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Repare como usamos o *wildcard* no `<url-pattern>` para indicar que todas as requisições serão filtradas e, portanto, terão uma conexão aberta já disponível.

Agora, 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 void 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 dispatch para o 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-16**.

10.6 - Exercícios: Filtros

1) Vamos criar o nosso filtro para abrir e fechar a conexão com o banco de dados

   a) Crie uma nova classe chamada `FiltroConexao` no pacote `br.com.caelum.agenda.filtro` e faça ela implementar a interface `javax.servlet.Filter`

   b) 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);
    }
}
```

   c) Declare o filtro no `web.xml`, para fazer com que todas as requisições passem por ele:

```
<filter>
    <filter-name>FiltroConexao</filter-name>
    <filter-class>br.com.caelum.agenda.filtro.FiltroConexao</filter-class>
</filter>

<filter-mapping>
    <filter-name>FiltroConexao</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

2) 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) Agora na sua `AlteraContatoLogic`, 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 AlteraContatoLogic implements Logica {

    public void executa(HttpServletRequest request, HttpServletResponse response)
                    throws Exception {

        // ....
        // ....

        // (procure o ContatoDAO no código existente)
        // busca a conexão pindurada na requisição
        Connection connection = (Connection) request.getAttribute("conexao");
        ContatoDAO dao = new ContatoDAO(connection); //passe a conexão no construtor

        // ....
        // ....
    }
}
```

Ou você pode fazer essa mesma modificação na nossa antiga `AdicionaContatoServlet`.

4) Altere um contato já existente na sua aplicação e verifique que tudo continua funcionando normalmente.

10.7 - Deploy do projeto em outros ambientes

Geralmente, ao desenvolvermos uma aplicação Web, possuímos um ambiente de desenvolvimento com um container que está em nossas máquinas locais. Depois que o desenvolvimento está finalizado e nossa aplicação está pronta para ir para o 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 `agenda.war`. Após compactar o diretório, efetuaremos o *deploy*.

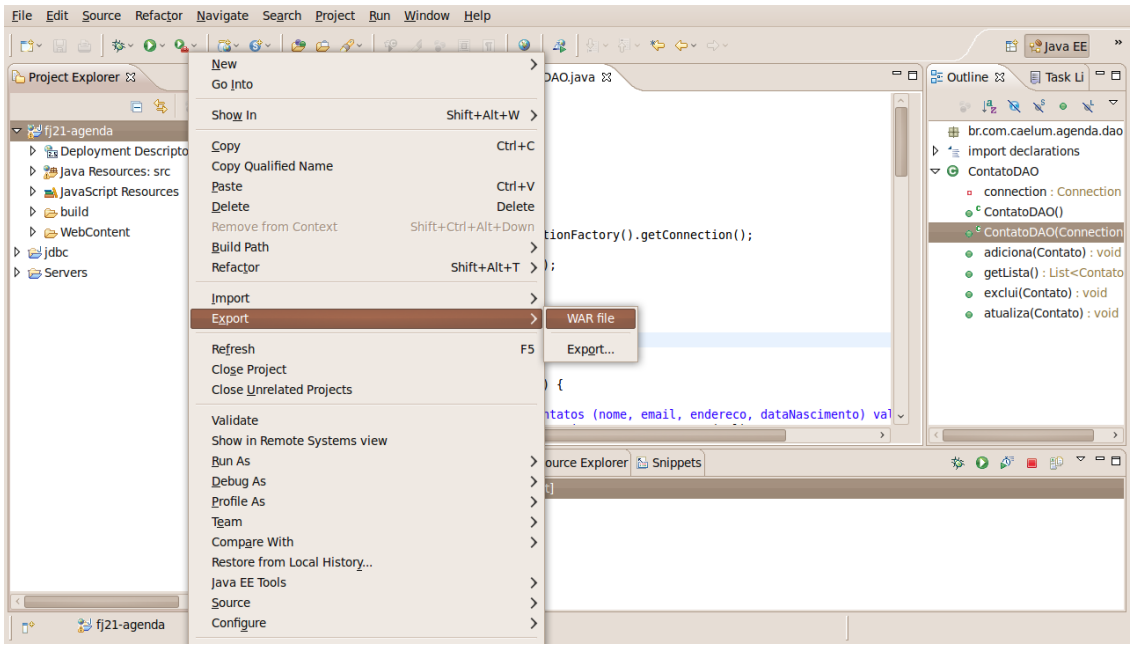
No Tomcat, basta copiar o arquivo .war no diretório **TOMCAT/webapps/**. Ele será descompactado pelo container e um novo contexto chamado **agenda** estará disponível. Repare que o novo contexto gerado pelo Tomcat adota o mesmo nome que o seu arquivo `.war`, ou seja, nosso arquivo chamava `agenda.war` e o contexto se chama `agenda`.

Ao colocarmos o `war` no Tomcat, podemos acessar nossa aplicação pelo navegador através do endereço: http://localhost:8080/agenda/bemvindo.jsp

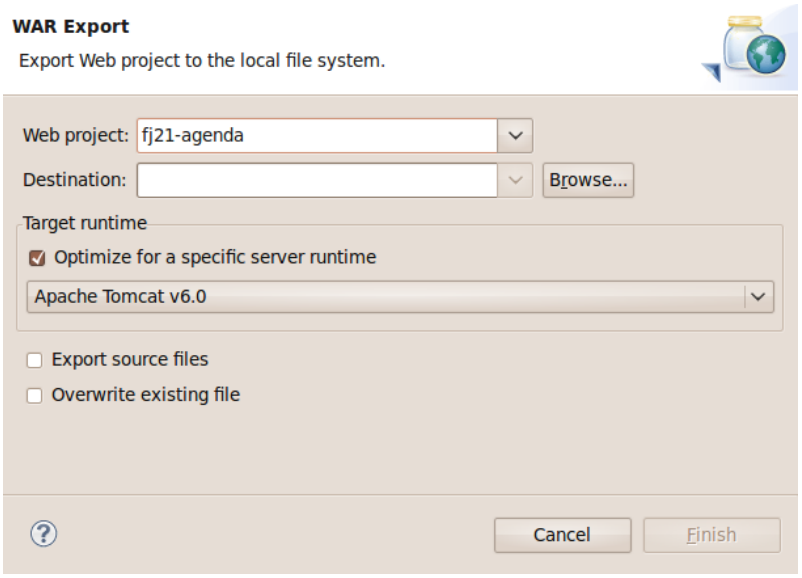
10.8 - Exercícios: Deploy com war

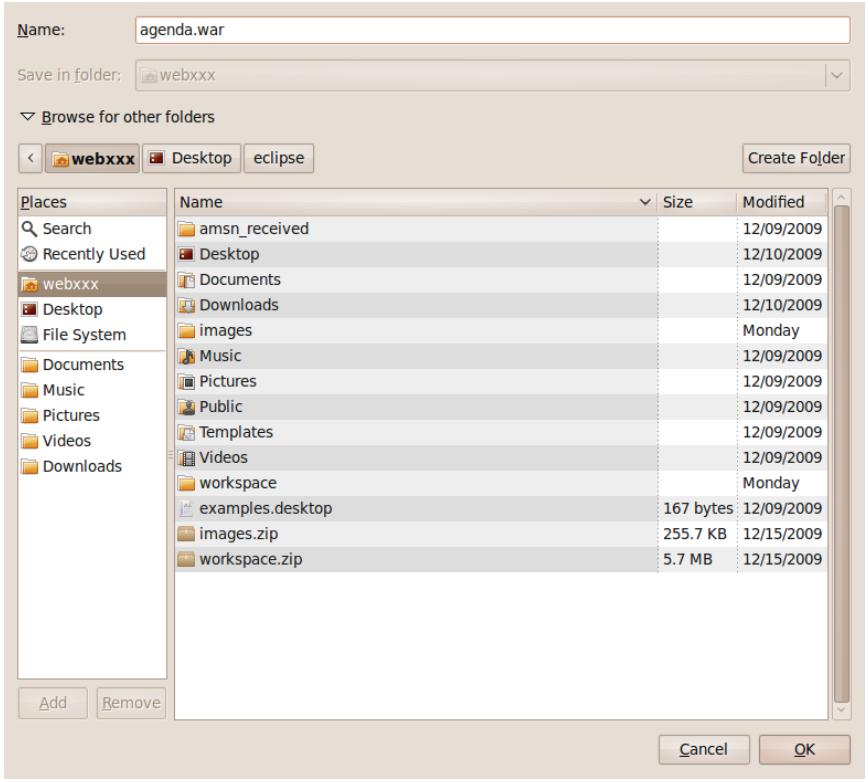
1) Vamos praticar criando um `war` e utilizando-o, mas antes de começarmos certifique-se de que o Tomcat esteja no ar.

   a) Clique com o botão direito no projeto e vá em Export -> WAR file



   b) Clique no botão Browse e escolha a pasta do seu usuário e o nome **agenda.war**.





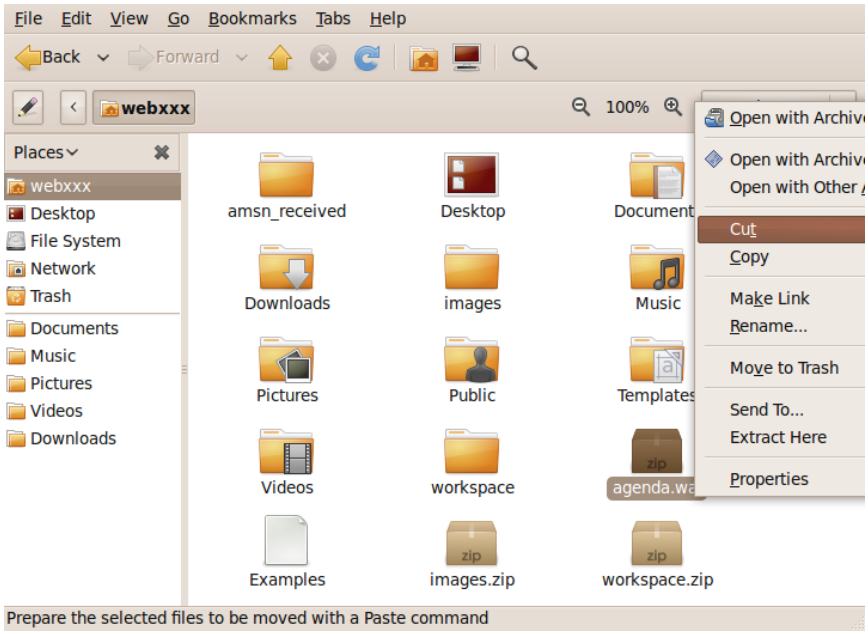
c) Clique em Finish

Pronto, nosso war está criado!

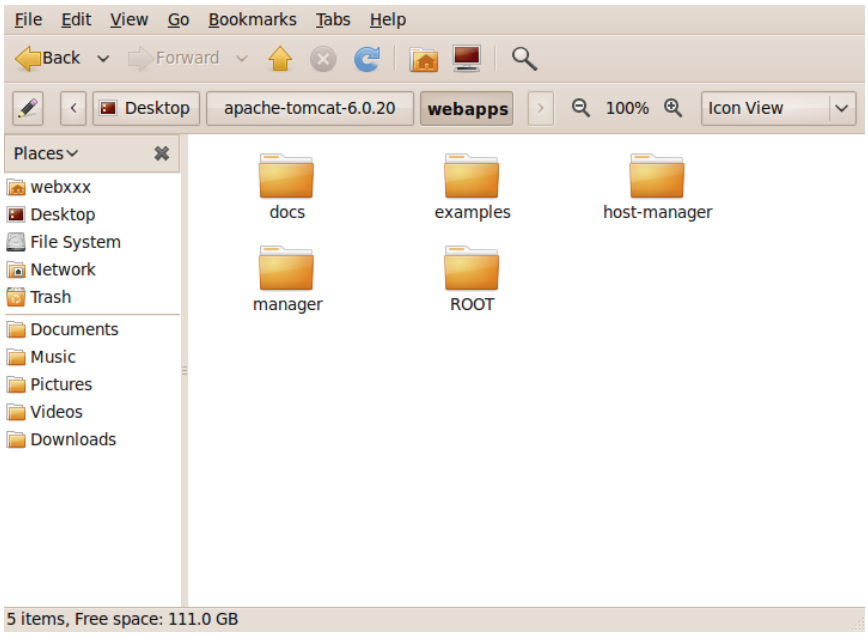
2) Vamos instalar nosso war!

a) Abra o File Browser

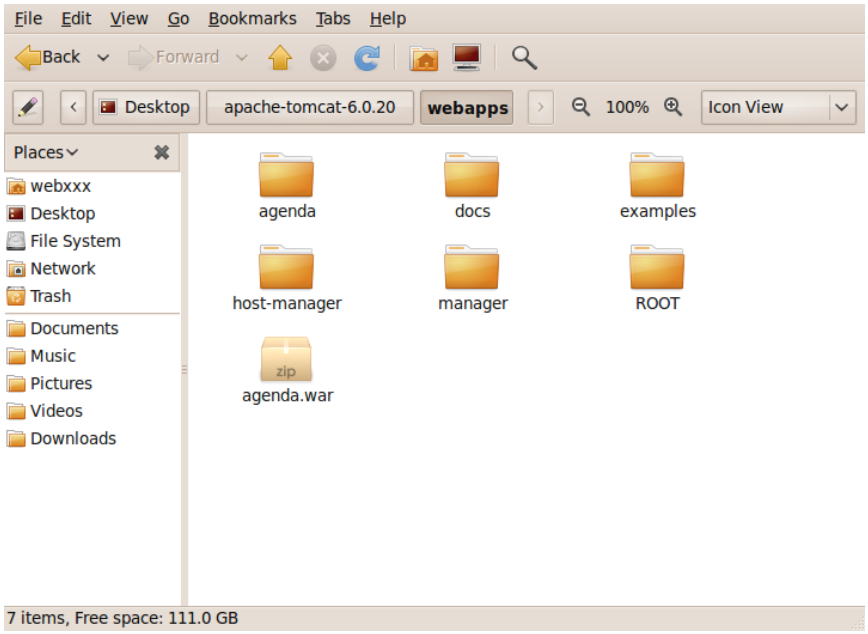
b) Clique da direita no arquivo `agenda.war` e escolha **Cut**.



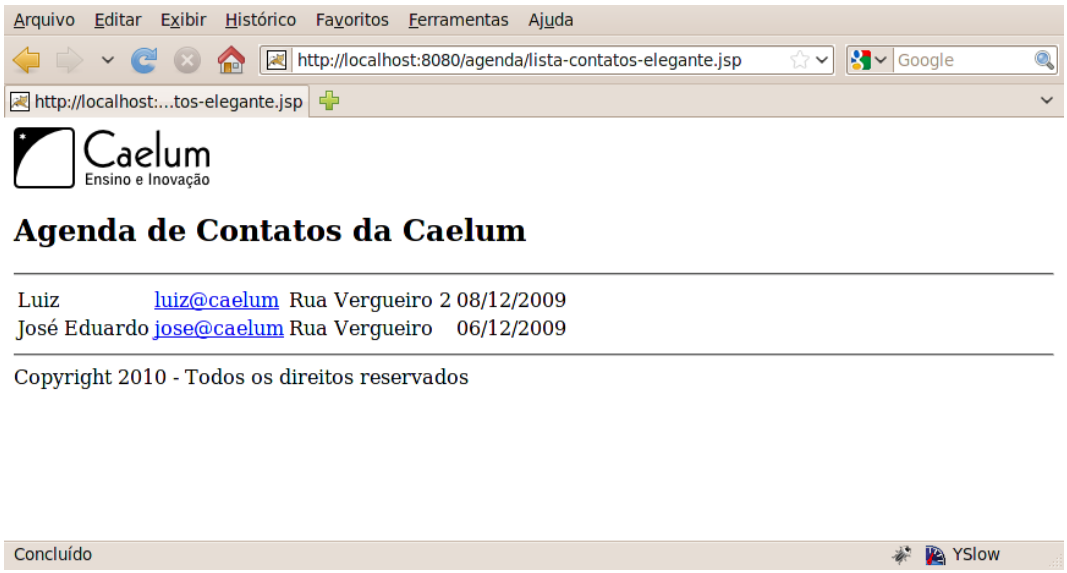
c) Vá para o diretório **apache-tomcat**, **webapps**. Certifique-se que seu tomcat está rodando.



d) Cole o seu arquivo aqui (**Edit**, **Paste**). Repare que o diretório `agenda` foi criado.



e) Agora podemos acessar o projeto através da URL: http://localhost:8080/agenda/lista-contatos-elegante.jsp



10.9 - Discussão em aula: lidando com diferentes nomes de contexto

CAPÍTULO 11

Struts 2

"Um homem pinta com seu cérebro e não com suas mãos."
– Michelangelo

Nesse capítulo, você aprenderá:

- Porque utilizar frameworks;
- Como funciona o framework Struts 2;
- As diferentes formas de se trabalhar com o Struts 2.

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 nos previnem de trabalharmos dessa forma, 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 e 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.

Para resolver o problema de ser trabalhoso, a comunidade do Struts 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.

Apesar de ser uma evolução com relação a primeira versão, o Struts 2 permite que o desenvolvedor possa trabalhar da mesma forma que na versão anterior. Tornando a migração entre as versões mais suave. Estudaremos ainda nesse capítulo as formas de se trabalhar com o Struts 2.

> **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. No fim dessa apostila, você encontra um apêndice sobre Struts 1 que mostra essas diferenças.

11.3 - Configurando o Struts 2

Para que possamos aprender o Struts 2, vamos criar um sistema de *lista de tarefas*. E o primeiro passo que precisamos fazer é ter o Struts 2 para adicionarmos em nossa aplicação. Podemos encontrá-lo no site http://struts.apache.org/2.x/. Lá, é possível encontrar diversas documentações e tutoriais, além dos JARs do projeto.

Uma vez que adicionamos os JARs do Struts 2 em nosso projeto dentro do diretório `WEB-INF/lib`, precisamos declarar o Filtro, que fará o papel de *Front Controller* da nossa aplicação, recebendo as requisições e as enviando às lógicas corretas. Para declararmos o Filtro do Struts 2, basta adicionarmos no `web.xml` da nossa aplicação:

```
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>
        org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Repare como é uma configuração normal de filtro, como as que fizemos antes.

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 Struts 2 permite criar as lógicas de 3 formas:

1) *Declaração em um arquivo XML*: Equivalente com a maneira em que se fazia no Struts 1;
2) *Utilizando convenções*: Não é necessário herdar nem escrever nenhum XML, desde que a sua lógica seja criada em um pacote pré-definido pelo framework. Essa abordagem só é conseguida através da utilização do plugin de convenções, que já vem no download do projeto;
3) *Utilizando anotações*: Foi a forma introduzida pelo WebWork que foi incorporada ao Struts 2 na fusão. Também é conseguido através do uso do plugin de convenções.

Vamos utilizar aqui o plugin de convenções para utilizarmos as anotações do Struts 2, mas explicaremos posteriormente como fazer o mesmo trabalho das outras duas formas.

11.5 - A lógica Olá Mundo!

Para criarmos nossa primeira lógica, vamos criar uma classe chamada `OlaMundoAction`. O sufixo `Action` é obrigatório para que o Struts 2 consiga carregar as classes que possuem lógicas a serem executadas. Como utilizaremos o plugin de convenções, **é obrigatório** que a sua `Action` esteja dentro de um sub-pacote com um dos nomes: **struts2**, **action** ou **actions**. No nosso caso, criaremos a classe dentro do pacote: `br.com.caelum.tarefas.action`.

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!". Podemos criar também um método de qualquer nome dentro dessa classe, desde que ele esteja com a anotação `@Action`. A anotação `@Action` recebe um atributo chamado `value` que indica qual será a URL utilizada para invocar a `Action`. Portanto, se colocarmos o valor `olaMundoStruts` acessaremos nossa `Action` pela URL http://localhost:8080/fj21-tarefas/olaMundoStruts.

Vamos utilizar uma outra convenção, que é criar um método chamado `execute`. Esse método deve retornar uma String que indica qual JSP deve ser executado após a lógica. Essa indicação se dá através de um sinal, por exemplo, podemos retornar "ok" e enviar o usuário para uma página chamada `ola.jsp`. Esse mapeamento é feito através da anotação `@Action` no atributo `results`.

O atributo `results` recebe um `Array` de anotações `@Result`, na qual indicamos o nome do resultado, no nosso caso "ok" e onde se localiza o JSP desse resultado, através do atributo `location`.

Dessa forma, teríamos a seguinte classe:

```
public class OlaMundoAction {

    @Action(value = "olaMundoStruts", results = {
```

```
        @Result(location = "/olaMundoStruts.jsp", name = "ok") }
    )
    public String execute() {
        System.out.println("Executando a lógica com Struts2");
        return "ok";
    }
}
```

Um ponto importante a se notar é que podemos criar vários métodos que respondem por URL, ou seja, `Actions` dentro dessa classe. Bastaria que nós utilizássemos novamente a anotação `@Action` nesse novo método.

Por fim, só precisamos criar o JSP que mostrará a mensagem "Olá mundo!". Basta criar o arquivo `olaMundoStruts.jsp`, que mapeamos anteriormente como um `@Result` da nossa `Action`, com o seguinte conteúdo:

```
<html>
    <body>
        <h2>Olá mundo com Struts2!</h2>
    </body>
</html>
```

Agora podemos acessar nossa `Action` pela URL http://localhost:8080/fj21-tarefas/olaMundoStruts. O que acontece é que após a execução da sua `Action` o Struts 2 verifica qual foi o resultado retornado pelo seu método e procura uma anotação `@Result` com o resultado equivalente e dispacha a requisição para a página indicada.

11.6 - Exercícios: Configurando o Struts 2 e testando a configuração

Configurando o Struts 2 em casa

Caso você esteja em casa, faça o download do Struts 2 e use apenas os seguintes jars na sua aplicação:

- commons-fileupload-1.x.x.jar
- commons-io-1.x.x.jar
- commons-logging-1.0.4.jar
- freemarker-2.3.15.jar
- ognl-2.7.3.jar
- struts2-convention-plugin-2.1.8.1.jar
- struts2-core-2.1.8.1.jar
- xwork-core-2.1.6.jar
- javassist-3.x.ga.jar

Se você colocar todos os jars que vem na distribuição do Struts 2, sua aplicação lançará uma exceção indicando que está faltando o Spring, vamos utilizar o Spring no curso FJ-27.

1) Vamos agora configurar o Struts 2 em um novo projeto.

   a) Crie um novo projeto web: **File** -> **New** -> **Project...** -> **Dynamic Web Project** chamado `fj21-tarefas`.

   b) Na aba **Servers**, clique com o botão direito no Tomcat e vá em **Add and Remove...**:

   c) Agora basta selecionar o nosso projeto fj21-tarefas e clicar em **Add**:

   d) 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/Caelum/21` e confirme a importação.

   e) No Desktop, entre no diretório Caelum -> 21 -> struts2-jars.

   f) Copie todos os arquivos para dentro de `WebContent/WEB-INF/lib` do projeto **fj21-tarefas** que estamos desenvolvendo.

   g) Por fim, abra o arquivo `web.xml` para fazermos a declaração do filtro do Struts 2:

```
<web-app...>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

2) Agora, vamos fazer um simples Olá Mundo, para testarmos nossa configuração:

   a) Crie uma nova classe chamada `OlaMundoAction` no pacote `br.com.caelum.tarefas.action`

   b) Adicione nessa classe o seguinte conteúdo:

```
public class OlaMundoAction {

    @Action(value = "olaMundoStruts", results = {
        @Result(location = "/olaMundoStruts.jsp", name = "ok") }
    )
    public String execute() {
        System.out.println("Executando a lógica com Struts2");
        return "ok";
    }
}
```

   c) Agora, precisamos criar o JSP que será exibido após a execução da nossa lógica. Crie o arquivo `olaMundoStruts.jsp` no diretório `WebContent` do projeto com o conteúdo:

```
<html>
    <body>
        <h2>Olá mundo com Struts2!</h2>
```

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

d) Acesse no seu navegador o endereço http://localhost:8080/fj21-tarefas/olaMundoStruts. O resultado deve ser algo parecido com:

11.7 - Adicionando tarefas e passando parâmetros à Action

Vamos agora 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-tarefas.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 `Action` que responde pela URL `adicionaTarefa`. Essa `Action`

precisa receber os dados da requisição e gravar a tarefa que o usuário informou na tela:

```
public class AdicionaTarefasAction {

    @Action(value="adicionaTarefa", results = {
        @Result(name="ok", location="/tarefa-adicionada.jsp")
    })
    public String execute() {
        new TarefaDAO().adiciona(tarefa);
        return "ok";
    }
}
```

Mas, como montaremos o objeto `tarefa` para passarmos ao nosso `DAO`? Dentro dessa nossa classe `AdicionaTarefasAction` 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 Struts 2 não é diferente.

Essa ligação é feita através da criação de um atributo dentro da classe da nossa `Action`. Esse atributo é o objeto que deverá ser populado pelo Struts 2 com os dados que vieram da requisição. Portanto, vamos criar na nossa classe um novo atributo chamado `tarefa`:

```
public class AdicionaTarefasAction {

    private Tarefa tarefa;

    @Action(value="adicionaTarefa", results = {
        @Result(name="ok", location="/tarefa-adicionada.jsp")
    })
    public String execute() {
        new TarefaDAO().adiciona(tarefa);
        return "ok";
    }
}
```

Agora, 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 `tarefa.descricao`. Dessa forma, nosso formulário ficaria agora com o seguinte código:

```
<html>
    <body>
        <h3>Adicionar tarefas</h3>
        <form action="adicionaTarefa" method="post">
            Descrição: <br />
            <textarea name="tarefa.descricao" rows="5" cols="100"></textarea><br />

            <input type="submit" value="Adicionar">
        </form>
```

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

No primeiro campo, para popular o objeto `tarefa`, o Struts precisará criar o objeto `tarefa` para nós. Como a classe `Tarefa` é um JavaBean e possui construtor sem argumentos isso não será problema, mas como ele guardará esse novo `objeto` dentro da nossa `Action`? Precisamos adicionar dentro da nossa `Action` um método `setter` para a propriedade `tarefa`. Esse setter será invocado pelo Struts para criar o seu objeto. Em seguida ele precisará desse objeto para popular a descrição e as demais propriedades que poderíamos atribuir algum valor. Para isso o Struts também precisará que criemos um método `getter`.

Portanto, agora nossa `Action` será:

```
public class AdicionaTarefasAction {

    private Tarefa tarefa;

    @Action(value="adicionaTarefa", results = {
        @Result(name="ok", location="/tarefa-adicionada.jsp")
    })
    public String execute() {
        new TarefaDAO().adiciona(tarefa);
        return "ok";
    }

    public void setTarefa(Tarefa tarefa) {
        this.tarefa = tarefa;
    }

    public Tarefa getTarefa() {
        return this.tarefa;
    }
}
```

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>
```

11.8 - Exercícios: Criando tarefas

Podemos agora criar o formulário e nossa `Action` para fazer a gravação das tarefas.

1) O primeiro passo é criar nosso formulário. Crie um novo arquivo **formulario-tarefas.jsp** na pasta *WebContent* com nosso formulário de cadastro de novas tarefas:

```
<html>
    <body>
        <h3>Adicionar tarefas</h3>
```

```
        <form action="adicionaTarefa" method="post">
            Descrição: <br />
            <textarea name="tarefa.descricao" rows="5" cols="100"></textarea><br />

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

2) Precisamos agora da Action que trata a adição de tarefas. Crie uma nova classe no pacote `br.com.caelum.tarefas.action` chamada `AdicionaTarefasAction`.

   Nossa classe precisa ter um atributo privado do tipo `Tarefa` chamado `tarefa`. Através dele vamos receber os dados populados no formulário. **Gere o getter e o setter para ele.**

   Além disso, nosso método `execute` deve chamar o DAO e adicionar a tarefa. Não esqueça de fazer o mapeamento na anotação `@Action` indicando a URL `adicionaTarefa` usada anteriormente no formulário e com a configuração do *result* indicando algum novo JSP de confirmação, como por exemplo **tarefa-adicionada.jsp**.

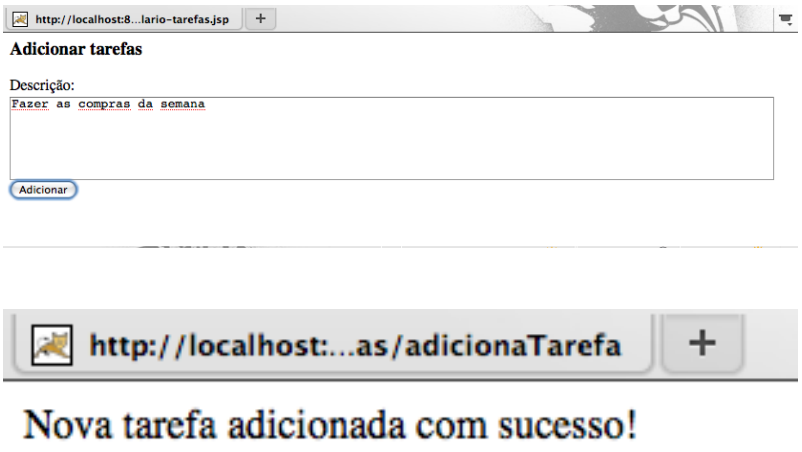
   O código final deve ser algo assim:

```
public class AdicionaTarefasAction {


    private Tarefa tarefa;

    @Action(value="adicionaTarefa", results = {
        @Result(name="ok", location="/tarefa-adicionada.jsp")
    })
    public String execute() {
        new TarefaDAO().adiciona(tarefa);
        return "ok";
    }

    // get e set aqui
}
```

   E, por fim, criamos o arquivo **tarefa-adicionada.jsp** que mostrará uma mensagem de confirmação de que a tarefa foi efetivamente adicionada:

```
<html>
    <body>
        Nova tarefa adicionada com sucesso!
    </body>
</html>
```

3) Acesse no seu navegador o endereço http://localhost:8080/fj21-tarefas/formulario-tarefas.jsp 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 um 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)
);
```

11.9 - Incluindo validação no cadastro de tarefas

Agora, já conseguimos adicionar novas tarefas em nossa aplicação. Porém, o que impede de algum usuário de tentar 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 action de adicionar tarefas.

O Struts2 possui um poderoso framework de validações, e a maneira mais fácil de ter acesso a este framework é fazendo nossa action estender a classe `ActionSupport`. Esta classe implementa as principais interfaces do framework Struts2, como, por exemplo, a `ValidationAware`. Por isso, existe uma recomendação de que as actions estendam a classe `ActionSupport`.

```
public class AdicionaTarefasAction extends ActionSupport {
  ...
```

```
}
```

A partir de agora, já podemos fazer o uso do framework de validação do Struts2. As validações podem ser incluídas através de configuração no XML, anotações ou programaticamente.

XML é certamente a opção mais trabalhosa, e a forma mais interessante é utilizando anotações. A anotação principal se chama `@Validations`. Ela pode ser usada no nosso modelo (`Tarefa`) e também na action.

Por exemplo, podemos definir que desejamos validar a `Tarefa`:

```
@Validations
public class Tarefa {
    ...
}
```

Mas qual seria a regra de validação? Falta definir as regras específicas, e isso também é feito através de anotações. A anotação `@RequiredStringValidator` indica que o valor é obrigatório (não nulo ou vazio), com `@StringLengthFieldValidator` podemos definir o mínimo (*minLength*) e máximo (*maxLength*) de um *String*. Por exemplo, para definir a regra do atributo `descricao` da classe `Tarefa`:

```
@RequiredStringValidator(message="Valor obrigatório", shortCircuit=true)
@StringLengthFieldValidator(message="Mínimo 10 caracteres", minLength="10")
public void setDescricao(String descricao) {
    this.descricao = descricao;
}
```

O atributo `shortCircuit` significa que queremos parar com o processo de validação no caso daquela regra de validação falhar. O atributo `message` define a mensagem de erro que poderá ser exibida na tela do usuário, desde que utilizemos a tag adequada em nosso JSP.

Existem várias outras anotações, como `@RegexFieldValidator` para aplicar uma expressão regular, `@EmailValidator` para validar um email ou `@DateRangeFieldValidator` que valida se a data está dentro de um range.

Mais informaões na documentação sobre as anotações do Struts2: http://struts.apache.org/2.0.14/struts2-core/apidocs/com/opensymphony/xwork2/validator/annotations/package-summary.html

Para habilitar o framework também precisamos usar a mesma anotação `@Validations` na action e indicar que queremos "visitar" o nosso modelo. Só assim as regras definidas no modelo serão verificadas:

```
@Validations
public class AdicionaTarefasAction extends ActionSupport {

    private Tarefa tarefa;

    @VisitorFieldValidator(key="tarefa")
    public void setTarefa(Tarefa tarefa) {
        this.tarefa = tarefa;
    }

    //getter e método execute omitidos
```

```
}
```

Pronto, conseguimos definir as regras de validação para o modelo `Tarefa`.

Existe uma maneira mais simples ainda de usar a validação com as mesmas anotações. As regras de validação podem ser definidas exclusivamente na action, ou seja, não o modelo `Tarefa` que possui as regras, apenas a action. Veja o exemplo:

```
@Validations(
 requiredStrings={
    @RequiredStringValidator(fieldName="tarefa.descricao",message="Valor obrigatório")
 },
 stringLengthFields={
    @StringLengthFieldValidator(fieldName="tarefa.descricao", minLength="5",
                                    message="Min. 5 car.")
 })
public class AdicionaTarefasAction extends ActionSupport {

}
```

Dessa maneira, apenas a action conhece as regras e, assim, a anotação `@VisitorFieldValidator` ficou desnecessária. Repare também que foi preciso avisar qual atributo no modelo deve ser validado (`fieldName="tarefa.descricao`

).

Agora que as regras de validação foram definidas, só falta informar ao struts qual é página que possui o nosso formulário para mostrar depois as mensagens. Assim quando um erro de validação acontecer, o struts irá redirecionar para essa página automaticamente. Isso é feito através da já conhecida anotação `@Result` dentro da action.

```
@Action(value="adicionaTarefa", results = {
    @Result(name="ok", location="/tarefa-adicionada.jsp"),
    @Result(name="input", location="/formulario-tarefas.jsp")
})
public String execute() {
    ...
}
```

É importante que o `@Result` use **input** como `name`, só assim o Struts irá redirecionar.

Mostrando as mensagens de validação

Para exibir as mensagens de validação no JSP usamos um tag especial que o Struts oferece. O tag se chama **s:fieldError**:

```
<s:fielderror fieldName="tarefa.descricao" />
```

O atributo *fieldName* indica com que atributo essa mensagem está relacionada.

Abaixo está o código completo do formulário *formulario-tarefas.jsp*. Repare que é preciso importar o taglib do struts:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<body>
    <h3>Adicionar tarefas</h3>
    <form action="adicionaTarefa" method="post">
        Descrição:
            <br/>
        <textarea rows="5" cols="100" name="tarefa.descricao"></textarea>
            <br/>
        <s:fielderror fieldName="tarefa.descricao"/>
            <br/>
        <input type="submit" value="Adicionar"/>
    </form>
</body>
</html>
```

Validando programaticamente

Quando estendemos a classe `ActionSupport`, ganhamos também a possibilidade de realizar validações de maneira programática, ao invés da forma declarativa com XML ou anotações. Isso pode ser necessário para definir regras de validação mais customizadas, regras que não foram previstas pelas anotações já existentes no Struts.

Para realizar validações de forma programática, precisamos sobrescrever o método `validate` que herdamos da classe `ActionSupport`. Dentro deste método, realizaremos nossas verificações. Em nosso exemplo, vamos verificar se o texto de descricao da tarefa é nulo ou vazio. E, para incluir as mensagens de erro, utilizaremos um outro método de `ActionSupport` chamado `addFieldError`.

```
public class AdicionaTarefasAction extends ActionSupport {

    // ... outros metodos omitidos ...

    @Override
    public void validate() {
        if (getTarefa().getDescricao() == null || getTarefa().getDescricao().isEmpty()){
            super.addFieldError("tarefa.descricao", "Descrição não pode ser vazia" );
        }
    }
}
```

Mensagens internacionalizadas

Para deixar as mensagens de validação mais fáceis de alterar, é comum criar um arquivo separado que possui apenas mensagens. Este arquivo é normalmente chamado *mensagens.properties* ou *messages.properties*.

Na pasta *src* do projeto podemos então criar este arquivo com o seguinte conteúdo:

```
tarefa.descricao.vazia=Descrição deve ser preenchida!
```

```
tarefa.adicionada.com.sucesso=Tarefa adicionada com sucesso!
```

Repare que definimos as mensagens em um estilo de: <chave>=<valor>.

O struts2 pode carregar automaticamente este arquivo, desde que a linha abaixo seja incluida no arquivo *struts.xml*:

```
<struts>
    <constant name="struts.custom.i18n.resources" value="mensagens" />
</struts>
```

Podemos carregar mais do que um arquivo de mensagens:

```
<struts>
    <constant name="struts.custom.i18n.resources" value="mensagens, erros" />
</struts>
```

O Struts2 então carrega o(s) arquivo(s) automaticamente. Dessa maneira podemos usar as chaves das mensagens dentro das anotações de validação pelo atributo `key`:

```
@RequiredFieldValidator(fieldName="tarefa.descricao", key="tarefa.descricao.vazia")
```

Também é possível definir a mensagem de erro programaticamente dentro do método `execute` da sua action. Para isso, a sua action deve estender `ActionSupport`:

```
super.addFieldError("tarefa.descricao", super.getText("tarefa.descricao.vazia"));
```

Ou para definir uma mensagem informativa ou genérica:

```
super.addActionMessage(getText("tarefa.adicionada.com.sucesso"));
```

11.10 - Exercícios: Validando tarefas

1) Abra a classe **AdicionaTarefasAction**. Coloque a anotação **@Validations** acima da classe E estenda a classe `ActionSupport`:

```
@Validations(requiredStrings={

    @RequiredStringValidator(fieldName="tarefa.descricao", message="Valor obrigatório")
})
public class AdicionaTarefasAction extends ActionSupport {
    ...
}
```

2) Dentro da classe **AdicionaTarefasAction**, procure o método **execute**. É preciso *alterar* a anotação `@Action`. Nela, coloque mais um `@Result` para definir o formulário que enviou a requisicão (*input*). Adicione apenas mais um `@Result`:

```
@Action(value="adicionaTarefa", results = {
```

```
        @Result(name="ok", location="/tarefa-adicionada.jsp"),
          @Result(name="input", location="/formulario-tarefas.jsp")
    })
public String execute() {
    ...
}
```

3) Abra o JSP **formulario-tarefas.jsp**. Adicione no início da página a declaração do taglib do Struts:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
```

Dentro do HTML adicione o tag **s:fielderror** acima do tag *form*. Adicione apenas o tag *s:fielderror*:

```
<s:fielderror fieldName="tarefa.descricao" />
<form action="adicionaTarefa" method="post">
```

4) Acesse no seu navegador o endereço http://localhost:8080/fj21-tarefas/formulario-tarefas.jsp Envia uma requisicão SEM preencher a descrição da tarefa, a mensagem de validação deve aparecer.

5) (opcional) Na classe `AdicionaTarefasAction` adicione mais uma regra de validação dentro da anotação `@Validations` para verificar o tamanho mínimo da descrição:

```
@Validations(requiredStrings={

    @RequiredStringValidator(fieldName="tarefa.descricao", message="Valor obrigatório")
  }, stringLengthFields={
    @StringLengthFieldValidator(fieldName="tarefa.descricao",
                minLength="5", message="Minimo de 5 caracteres")
})
```

11.11 - Listando as tarefas e disponibilizando objetos para a view

Agora que já conseguimos adicionar tarefas em nossa aplicação, precisamos saber o que foi adicionado. Para isso precisamos criar uma nova `Action`. A função dessa `Action` será invocar o `TarefaDAO` para conseguir a lista das tarefas que estão no banco de dados. Essa lista de tarefas deverá ser disponibilizada para o JSP fazer sua exibição.

Para que possamos disponibilizar um objeto para o JSP, basta criarmos o `getter` do objeto que queremos disponibilizar para a nossa View. Então, poderíamos criar uma nova `Action` chamada `ListaTarefasAction` que disponibiliza uma lista de tarefas para o JSP a ser executado em seguida:

```
public class ListaTarefasAction {
    private List<Tarefa> tarefas;

    @Action(value="listaTarefas", results= {
        @Result(name="ok", location="/lista-tarefas.jsp")
    })
    public String execute() {
        tarefas = new TarefaDAO().lista();
        return "ok";
    }

    public List<Tarefa> getTarefas() {
```

```
        return tarefas;
    }
}
```

Nesse caso, temos o método `getTarefas`. Dessa forma, será disponibilizado para o JSP que vai ser renderizado em seguida 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.

11.12 - Exercícios: Listando tarefas

1) Vamos criar a listagem das nossas tarefas mostrando se a mesma já foi finalizada ou não.

   a) Crie uma classe chamada `ListaTarefasAction` no pacote `br.com.caelum.tarefas.action` contendo o seguinte código:

```
private List<Tarefa> tarefas;


@Action(value="listaTarefas", results= {
    @Result(name="ok", location="/lista-tarefas.jsp")
})
public String execute() {
    tarefas = new TarefaDAO().lista();
    return "ok";
}

public List<Tarefa> getTarefas() {
    return tarefas;
}
```

   b) Para fazer a listagem, vamos precisar da JSTL (iremos fazer um `forEach`), portanto precisamos importá-la. Primeiro, vá ao Desktop, e entre no diretório `Caelum/21/jstl`.

   c) Haverá dois JARs, jstl-impl-xx.jar e jstl-api-xx.jar.

   d) Copie-os (CTRL+C) e cole-os (CTRL+V) dentro de `workspace/fj21-tarefas/WebContent/WEB-INF/lib`

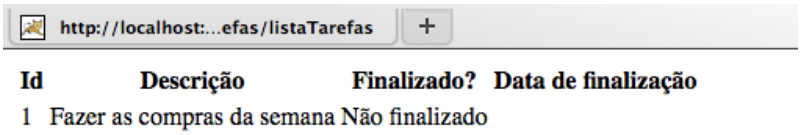
   e) No Eclipse, dê um F5 no seu projeto. Pronto, a JSTL já está em nosso projeto.

   f) Agora crie o JSP que fará a exibição das tarefas. Chame-o de `lista-tarefas.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>
    <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 false}">
                    <td>Finalizado</td>
                </c:if>
                <td>
                    <fmt:formatDate value="${tarefa.dataFinalizacao.time}" pattern="dd/MM/yyyy"/>
                </td>
            </tr>
        </c:forEach>
        </table>
    </body>
    </html>
```

g) Acesse o endereço http://localhost:8080/fj21-tarefas/listaTarefas e veja o resultado.



11.13 - Redirecionando a requisição para outra Action

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?tarefa.id=${tarefa.id}">Remover</a></td>
```

E, agora, podemos desenvolver a nossa `Action` 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.

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 `lista-tarefas.jsp` que fizemos a pouco. Mas, essa não é uma boa abordagem, porque teríamos que, outra vez, disponibilizar a lista das tarefas para o JSP, algo que já fazemos na `Action` de listar as tarefas, a `ListaTarefasAction`.

Já que a `ListaTarefasAction` faz esse trabalho, poderíamos ao invés de redirecionar a execução para o JSP, enviá-la para a `Action`. Para isso, na anotação `@Result` que usamos para fazer o redirecionamento, ao invés de usarmos o atributo `location`, passamos o nome da `Action` que queremos invocar através de um parâmetro. Fazemos isso no parâmetro `params` da anotação `@Result`.

Um outro detalhe é que precisamos indicar na anotação `@Result` que o redirecionamento é feito para uma outra ação, através do atributo `type` da anotação contendo o valor `redirectAction`.

Portanto, para redirecionarmos para a ação listaTarefas que já temos criada, basta fazermos:

```
@Result(name="ok", type="redirectAction", params= {"actionName", "listaTarefas"})
```

Dessa forma, a nossa `Action` para remoção de tarefas, que podemos chamar de `RemoveTarefaAction`, possuirá o seguinte código:

```
public class RemoveTarefaAction {

    private Tarefa tarefa;

    @Action(value="removeTarefa", results= {
        @Result(name="ok", type="redirectAction", params= {"actionName", "listaTarefas"})
    })
    public String execute() {
        new TarefaDAO().remove(tarefa);
        return "ok";
    }

    public void setTarefa(Tarefa tarefa) {
        this.tarefa = tarefa;
    }

    public Tarefa getTarefa() {
        return this.tarefa;
    }
}
```

Redirecionando para um JSP

Para redirecionar para um JSP, podemos usar uma síntaxe bem parecida:

```
@Action(value="minhaAcao", results= {

    @Result(name="ok", type="redirect", location="/pagina.jsp")
})
public String execute() { ... }
```

Com `type=redirect` indicamos ao Struts2 fazer um redirecionamento no lado do cliente.

11.14 - Exercícios: Removendo e alterando tarefas

1) Vamos fazer a funcionalidade de remoção de tarefas.

a) Adicione no `lista-tarefas.jsp` uma coluna com um link que ao ser clicado invocará a `Action` para remover contato.

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

b) Crie uma nova classe no pacote `br.com.caelum.tarefas.action` chamada `RemoveTarefaAction` com o código:

```
public class RemoveTarefaAction {


    private Tarefa tarefa;

    @Action(value="removeTarefa", results= {
        @Result(name="ok", type="redirectAction", params= {"actionName", "listaTarefas"})
    })
    public String execute() {
        new TarefaDAO().remove(tarefa);
        return "ok";
    }

    public void setTarefa(Tarefa tarefa) {
        this.tarefa = tarefa;
    }

    public Tarefa getTarefa() {
        return this.tarefa;
    }
}
```

c) Acesse a lista de tarefas em http://localhost:8080/fj21-tarefas/listaTarefas e remova algumas tarefas.

2) Agora criaremos a tela para fazer a alteração das tarefas, como por exemplo, marcá-la como finalizada e definirmos a data de finalização.

a) 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>
```

b) Vamos criar uma nova `Action` que dado um `id`, devolverá a `Tarefa` correspondente para um JSP, que mostrará os dados para que a alteração possa ser feita.
Crie a classe `MostraTarefaAction` no pacote `br.com.caelum.tarefas.action`:

```
public class MostraTarefaAction {


    private Long id;
    private Tarefa tarefa;

    @Action(value="mostraTarefa", results = {
        @Result(name="ok", location="/mostra-tarefa.jsp")
    })
    public String execute() {
        tarefa = new TarefaDAO().buscaPorId(id);
        return "ok";
    }
```

```
    public void setId(Long id) {
        this.id = id;
    }

    public Tarefa getTarefa() {
        return tarefa;
    }
}
```

c) Agora, crie o JSP `mostra-tarefa.jsp` 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="tarefa.id" value="${tarefa.id}" />

        Descrição:<br />
        <textarea name="tarefa.descricao" cols="100" rows="5">${tarefa.descricao}</textarea>
        <br />

        Finalizado? <input type="checkbox" name="tarefa.finalizado"
            value="true" ${tarefa.finalizado?'checked':''}/> <br />

        Data de finalização: <br />
        <input type="text" name="tarefa.dataFinalizacao"
          value="<fmt:formatDate value="${tarefa.dataFinalizacao.time}" pattern="dd/MM/yyyy" />"/>
        <br />

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

d) Vamos criar a `Action` que cuidará da alteração da tarefa. Crie uma classe no pacote `br.com.caelum.tarefas.action` chamada `AlteraTarefaAction` com o código:

```
public class AlteraTarefaAction {


    private Tarefa tarefa;

    @Action(value="alteraTarefa", results= {
        @Result(name="ok", type="redirectAction", params= {"actionName", "listaTarefas"})
    })
    public String execute() {
        new TarefaDAO().altera(tarefa);
        return "ok";
    }
```

```
        public void setTarefa(Tarefa tarefa) {
            this.tarefa = tarefa;
        }

        public Tarefa getTarefa() {
            return this.tarefa;
        }
    }
```

e) Acesse a lista de tarefas em http://localhost:8080/fj21-tarefas/listaTarefas e altere algumas tarefas.

11.15 - Desafio

1) Adicione o campo com calendário que fizemos no capítulo de criação de Tags em nosso projeto e utilize-o no formulário de alteração.

11.16 - Para saber mais: Outras formas de se trabalhar com o Struts 2

No começo do capítulo falamos que o Struts 2 possui 3 formas de se trabalhar, uma com o uso de anotações, outra com XML e a outra através de convenções. Já aprendemos as anotações, agora vamos dar uma breve olhada nas outras duas formas.

Utilizando XML

A utilização de XML é a forma com que era feita no Struts 1. É uma forma mais trabalhosa e que hoje em dia está caindo em desuso, pois a utilização de anotações é mais simples e menos verbosa.

Para criarmos uma `Action` e declará-la em XML, devemos criar um arquivo chamado `struts.xml` e declararmos nossas `Actions` nele, como a seguir:

```
<struts>
    <package name="default" extends="struts-default">

        <action name="listaTarefas" class="br.com.caelum.tarefas.action.ListaTarefasAction">
            <result name="ok">/lista-tarefas.jsp</result>
        </action>

    </package>
</struts>
```

Repare que o XML é bastante intuitivo. Dizemos que temos uma `action` que possui determinado `result`.

Utilizando convenções

O uso de convenções é bastante interessante, pois permite que criemos nossas ações sem fazer nenhuma declaração, seja XML ou anotações. No entanto, precisamos seguir regras impostas pelo Struts 2.

Para que o Struts encontre sua `Action` ela precisa estar criada dentro de um pacote de nome **struts2**, **action** ou **actions**. E sua classe que terá a lógica **somente** poderá ter uma única `Action` que será o método chamado `execute`. Portanto, poderíamos criar uma `Action` chamada `OlaMundo` no pacote `br.com.caelum.actions`. O JSP que será exibido deverá estar dentro do diretório `WEB-INF/content` e caso o retorno do método `execute`, por exemplo, seja "ok", a página deverá se chamar ola-mundo-**ok**.jsp

11.17 - 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 `Action` 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 *Assynchronous 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>
```

Agora, 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 `TarefaDAO` 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. Fazemos isso através do atributo `httpheader` da anotação `@Result` do Struts 2 e passamos como parâmetro qual é o estado da resposta através do código 200:

```
@Result(name="ok", type="httpheader", params={"status", "200"})
```

Dessa forma, poderíamos ter uma `Action` para fazer a finalização da tarefa com o seguinte código:

```
public class FinalizaTarefaAction {

    private Long id;

    @Action(value="finalizaTarefa", results= {
        @Result(name="ok", type="httpheader", params={"status", "200"})
    })
    public String execute() {
        new TarefaDAO().finaliza(id);
        return "ok";
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

11.18 - Utilizando AJAX para marcar tarefas como finalizadas

Através do JQuery, podemos enviar uma requisição AJAX para o servidor, e na Action sinalizamos ao controlador que queremos devolver apenas o status 200 como resposta. Também seria útil notificar o usuário da aplicação que a requisicão 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/

11.19 - Exercícios: Ajax

1) Vamos adicionar AJAX na nossa aplicação. Para isso, utilizaremos o JQuery que precisamos importar para nosso projeto e em nossas páginas.

   a) Vá ao Desktop, e entre em `Caelum/21`;

   b) Copie os diretório `js` e cole-os dentro de `WebContent` no seu projeto fj21-tarefas; Caso você esteja em casa, faça o download em http://jqueryui.com/download

   c) Agora precisamos importar o JQuery em nossa página de listagem. Para isso, adicione logo após a Tag `<html>` o seguinte código no arquivo `lista-tarefas.jsp`:

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

   d) Pronto, importamos o JQuery para nossa aplicação.

2) Caso a tarefa não esteja finalizada, queremos que ela possua uma nova coluna que se chamará “Finalizar agora”. Ao clicar, chamaremos via AJAX uma `Action` que marcará a tarefa como finalizada e a data de hoje será marcada como a data de finalização da mesma.

   a) Altere a coluna que mostra a tarefa como **não** finalizada no arquivo `lista-tarefas.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>
```

   b) Crie a função Javascript `finalizaAgora` para chamar a `Action` 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("Tarefa finalizada");
            });
        }
    </script>
    <table>
    <!-- Resto da página com a tabela -->
```

3) Vamos criar a `Action` para finalizar a tarefa. Após a mesma ser executada, ela não deverá nos redirecionar para lugar nenhum, apenas indicar que a execução ocorreu com sucesso.

   a) Crie no pacote `br.com.caelum.tarefas.action` uma classe chamada `FinalizaTarefaAction` com o conteúdo:

```
public class FinalizaTarefaAction {


    private Long id;

    @Action(value="finalizaTarefa", results= {
        @Result(name="ok", type="httpheader", params={"status", "200"})
    })
    public String execute() {
        new TarefaDAO().finaliza(id);
        return "ok";
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

   b) 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.

4) (Opcional, Avançado) No mesmo estilo do exercício anterior, use o JQuery para acionar a action `RemoveTarefaAction` 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 `RemoveTarefaAction`, 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.

CAPÍTULO 12

Struts 2: 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;
- A diferença entre Interceptadores e Filtros.

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 até agora 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 agora criar nossa `Action` que fará o login da aplicação. Caso o usuário informado na tela de login existir, guardaremos o mesmo na sessão e entraremos no sistema, e caso não exista vamos voltar para a página de login.

O Struts nos possibilita chegar até a sessão através de uma classe chamada `ActionContext` que chamamos seu método estático `getContext`. Com o contexto, podemos pedir a sessão através do método `getSession`.

```
ActionContext.getContext().getSession()
```

Isso nos devolve um objeto do tipo `Map<String, Object>` no qual podemos guardar o objeto que quisermos dando-lhes uma chave que é uma String. Portanto, para logarmos o usuário na aplicação poderíamos criar uma ação com o seguinte código guardando o usuário logado na sessão:

```
@Action(value="login", results= {
    @Result(name="ok", location="/menu.jsp"),
    @Result(name="invalido", location="/login.jsp")
})
public String login() {
```

```
    if (new UsuarioDAO().existeUsuario(usuario)) {
        ActionContext.getContext().getSession().put("usuarioLogado", usuario);
        return "ok";
    } else {
        return "invalido";
    }
}
```

12.6 - Exercício: Fazendo o login na aplicação

1) Vamos criar a página de Login e a `Action` para autenticar o usuário.

   a) Crie a página `login.jsp` dentro de `WebContent` com o conteúdo:

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

   b) Agora crie uma nova classe chamada `LoginAction` no pacote `br.com.caelum.tarefas.action` com o código:

```
public class LoginAction {


    private Usuario usuario;

    @Action(value="login", results= {
        @Result(name="ok", location="/menu.jsp"),
        @Result(name="invalido", location="/login.jsp")
    })
    public String login() {
        if (new UsuarioDAO().existeUsuario(usuario)) {
            ActionContext.getContext().getSession().put("usuarioLogado", usuario);
            return "ok";
        }

        return "invalido";
    }

    public Usuario getUsuario() {
        return usuario;
    }

    public void setUsuario(Usuario usuario) {
        this.usuario = usuario;
    }
```

```
}
```

c) 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` 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>
```

d) Acesse a página de login em http://localhost:8080/fj21-tarefas/login.jsp e se logue na aplicação.

e) 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;
```

f) Caso não exista usuários cadastrados, cadastre algum utilizando o mesmo terminal aberto da seguinte maneira:

```
insert into usuarios values('seu_usuario', 'sua_senha');
```

2) (Opcional) Faça o logout da aplicação. Crie um link no `menu.jsp` que invocará uma `Action` que removerá o usuário da sessão e redirecione a navegação para a página de login.

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 Struts 2, podemos utilizar o conceito de 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 `com.opensymphony.xwork2.interceptor.Interceptor`. Ao implementar essa interface, precisamos implementar 3 métodos: init, destroy e intercept.

Os métodos `init` e `destroy` possuem as mesmas finalidades dos vistos no capítulo de `Servlet`s. Enquanto o método `intercept` é o responsável por executar antes e depois da requisição ser processada.

A assinatura do método `intercept` é a seguinte:

```
String intercept(ActionInvocation invocation) throws Exception
```

O parâmetro `ActionInvocation` permite que tenhamos acesso aos escopos da aplicação, bem como continuar a execução da ação invocada através do método `invoke()`. A `String` retornada é o resultado de para qual lugar a requisição será enviada, da mesma forma que na `Action`.

Portanto, poderíamos ter a classe `AutorizadorInterceptor` que verificará se o usuário está ou não logado antes de invocar qualquer ação:

```
public class AutorizadorInterceptor implements Interceptor {
    public String intercept(ActionInvocation invocation) throws Exception {
        Usuario usuarioLogado = (Usuario)invocation.getInvocationContext().
                        getSession().get("usuarioLogado");
        if(usuarioLogado == null) {
            return "naoLogado";
        } else {
            return invocation.invoke();
        }
    }

    //init e destroy
}
```

Ainda precisamos registrar o nosso novo interceptador. Mas o Struts 2 não permite que façamos isso via anotações, então usaremos a configuração via XML nesse caso. No arquivo `struts.xml` precisamos criar um grupo de configurações, que é chamado pelo Struts de `package`. Cada `package` pode herdar de outro para ter suas configurações. Para nos aproveitarmos das configurações pré existentes no struts utilizamos o parâmetro `extends="struts-default`

, e então nossa configuração fica da seguinte maneira:

```
<package name="default" extends="struts-default">
</package>
```

Dentro desse nosso pacote de configurações, vamos registrar o nosso `AutorizadorInterceptor`. Vamos criar uma Tag chamada `<interceptors>` e dentro dela adicionaremos todos os que existirem na nossa aplicação pela Tag `<interceptor>`. Basta darmos um nome para ele e indicarmos qual é a sua classe:

```
<interceptors>
    <interceptor name="autorizador"
        class="br.com.caelum.tarefas.interceptor.AutorizadorInterceptor"></interceptor>
</interceptors>
```

Dessa forma nosso Interceptador está criado e declarado. Agora, basta fazermos com que as nossas ações sejam interceptadas. O grande problema nesse ponto é que nossas `Actions` por padrão não possuem acesso às configurações do `package` que criamos, ou seja, elas não sabem que o `AutorizadorInterceptor` existe. Para resolvermos isso, precisamos indicar às nossas `Actions` que elas também possuem as configurações do `package` que criamos, que se chama `default`. Fazemos isso adicionando em nossas `Actions` a anotação `@ParentPackage` indicando o nome do pacote.

Dessa forma, teríamos o seguinte na classe `ListaTarefasAction`:

```
@ParentPackage("default")
public class ListaTarefasAction {
    //método execute, getter e atributo
```

```
}
```

Agora podemos dizer ao método `execute` dessa `Action` que ele será Interceptado pelo `Interceptor` que registramos com o nome de `autorizador` no XML. Fazemos isso através da anotação `@Action` pelo parâmetro `interceptorRefs` que recebe um `Array` de `@InterceptorRef`.

```
@Action(value="listaTarefas", results= {
    @Result(name="ok", location="/lista-tarefas.jsp")
}, interceptorRefs= {
    @InterceptorRef("autorizador")
})
public String execute() {
    tarefas = new TarefaDAO().lista();
    return "ok";
}
```

Interceptors Padrões

Não podemos esquecer de que caso declaremos outro Interceptor com @InterceptorRef, o Struts 2 deixa de usar os interceptadors padrões. Por isso é necessário declará-los junto da sua anotação de intercptor, ou criar uma Stack de Interceptadores com seu intercpetor e os interceptadores padrões do struts, como no código abaixo.

```
<interceptors>
    <interceptor name="autorizador"
        class="br.com.caelum.tarefas.interceptor.AutorizadorInterceptor"/>
    <interceptor-stack name="seguranca">
        <interceptor-ref name="defaultStack" />
        <interceptor-ref name="autorizador" />
    </interceptor-stack>
</interceptors>
```

Para podermos finalizar, só está faltando registrarmos o resultado `naoLogado` que utilizamos no nosso Interceptador para indicar que o usuário tentou acessar uma `Action` mas não estava logado. Uma forma que teríamos de fazer isso é em todas as nossas `Actions` que são interceptadas adicionarmos um novo `@Result` definindo o resultado `naoLogado`. Mas imagina quando tivermos mais lógicas? Se um dia mudarmos o nome do JSP, precisaríamos mudar essa definição em todos os lugares.

O Struts 2 permite que nós cadastremos resultados que servem para a aplicação inteira, conhecidos como **global results**. Esses resultados são registrados no `struts.xml` e funcionará para todas as nossas `Actions` dentro do `package` indicado:

```
<global-results>
    <result name="naoLogado">/login.jsp</result>
</global-results>
```

12.8 - Exercícios: Interceptando as requisições

1) Vamos criar um `Interceptor` que não permitirá que o usuário acesse `Action`s sem antes ter logado na aplicação.

a) Crie a classe `AutorizadorInterceptor` no pacote `br.com.caelum.tarefas.interceptor` com o seguinte código:

b) Implemente a interface `Interceptor` de `com.opensymphony.xwork2.interceptor.Interceptor`

c) No método `intercept` adicione o código para verificar se o usuário está adicionado na sessão. Caso ele esteja, seguiremos o fluxo normalmente, caso contrário indicaremos que o usuário não está logado e que deverá ser redirecionado para a página de login.

```
public String intercept(ActionInvocation invocation) throws Exception {

    Usuario usuarioLogado = (Usuario)invocation.getInvocationContext().
                    getSession().get("usuarioLogado");
    if (usuarioLogado == null) {
        return "naoLogado";
    }

    return invocation.invoke();
}
```

d) Não esqueca de implementar os métodos `init` e `destroy` da interface `Interceptor`. Deixe-os em branco.

e) Agora, temos que registrar o nosso novo interceptador no struts 2. Abra o arquivo `struts.xml` que está no diretório `src` do seu projeto e dentro da tag `<struts>` adicione:

```
<package name="default" extends="struts-default">
    <interceptors>
        <interceptor name="autorizador"
            class="br.com.caelum.tarefas.interceptor.AutorizadorInterceptor"></interceptor>
    </interceptors>
</package>
```

f) Repare que chamamos nosso interceptador de **autorizador**. Precisaremos desse nome mais para frente.

2) Agora precisamos indicar para nossas `Actions` que elas deverão ser interceptadas pelo interceptador chamado **autorizador**.

a) Vamos configurar primeiro a nossa `ListaTarefasAction` para executar o filtro.

b) Primeiro, precisamos indicar que ela conhece as configurações que fizemos no `struts.xml` dentro do `package` chamado `default`. Para isso, adicionaremos na classe a anotação `@ParentPackage` indicando qual é o pacote de configurações a ser utilizado.

```
@ParentPackage("default")

public class ListaTarefasAction {
    //método execute, getter e atributo
}
```

c) Agora basta dizermos que o método `execute` passará por um interceptador. Fazemos do atributo `interceptorRefs` da anotação `Action`. Nesse atributo passaremos o interceptador **autorizador** que acabamos de criar e registrar no `struts.xml`. Portanto, adicione o atributo novo na anotação `@Action` na classe `ListaTarefasAction`:

```
@Action(value="listaTarefas", results= {

    @Result(name="ok", location="/lista-tarefas.jsp")
}, interceptorRefs= {
    @InterceptorRef("autorizador")
```

```
})
public String execute() {
    tarefas = new TarefaDAO().lista();
    return "ok";
}
```

3) A único passo que está faltando fazermos é criar o redirecionamento para o caso do usuário não estar logado. Chamamos esse resultado de **naoLogado**. Vamos aproveitar o `struts.xml` e criar um resultado global, que funcionará para todas as nossas ações. Para isso, adicione **dentro do pacote** `default` que criamos no `struts.xml`:

```
<!-- dentro da tag package -->
<global-results>
    <result name="naoLogado">/login.jsp</result>
</global-results>
```

4) Reinicie o servidor e tente acessar a lista de tarefas em http://localhost:8080/fj21-tarefas/listaTarefas. Você deverá ser redirecionado para a tela de login.

12.9 - Discussão: Qual a diferença entre Filtro e Interceptor?

CAPÍTULO 13

Uma introdução prática ao Hibernate

"É uma experiência eterna de que todos os homens com poder são tentados a abusar."
– Baron de Montesquieu

Neste capítulo, você aprenderá a:

- Usar a ferramenta de ORM Hibernate;
- Gerar as tabelas em um banco de dados qualquer a partir de suas classes de modelo;
- Automatizar o sistema de adicionar, listar, remover e procurar objetos no banco;
- Utilizar anotações para facilitar o mapeamento de classes para tabelas;
- Criar classes de DAO bem simples utilizando o hibernate.

13.1 - 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: onde devemos deixar nossas queries? Dentro de um arquivo .java? Dentro do DAO? O ideal é criar um arquivo onde essas queries sejam externalizadas.

Há ainda a mudança do paradigma. A programação orientada a objetos difere muito do esquema entidade relacional e agora precisamos pensar das duas maneiras para fazer um único sistema. Esse buraco entre esses dois paradigmas gera bastante trabalho: a todo momento devemos "transformar" objetos em linhas e linhas em objetos, sendo que essa relação não é um-para-um.

Ferramentas para auxiliar nesta tarefa tornaram-se popular entre os desenvolvedores Java e são conhecidas como ferramentas de **mapeamento objeto-relacional** (ORM). O Hibernate, pertencente hoje ao grupo JBoss, é uma ferramenta ORM open source e é a líder de mercado, sendo a inspiração para a especificação JPA.

O Hibernate abstrai o seu código SQL, que 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 SQL, já que isso fica de responsabilidade da ferramenta.

13.2 - Criando seu projeto para usar o Hibernate

Para criar seu projeto, é necessário baixar os arquivos `.jar` que são dependência do Hibernate e colocá-los no *classpath*.

O site oficial do hibernate é o **www.hibernate.org**, onde você baixa a última versão na seção Download. Após descompactar esse arquivo, basta copiar todas as dependências do Hibernate para o nosso projeto. Você também precisa, além do *hibernate-core*, do *hibernate-annotations*. A partir do Hibernate 3.5, esses dois pacotes vem juntos no *core*, facilitando bastante sua configuração.

Além dessas, você precisa de uma implementação do SL4J, biblioteca de logging que o Hibernate usa. Para isso baixe o SL4J em:

http://www.slf4j.org/download.html

E coloque um dos jars de implementação no seu projeto, como o `slf4j-log4j12-1.5.x`, já que o SL4J é uma casca para as várias APIs de logging existentes.

O Hibernate vai gerar o código SQL para qualquer banco de dados. Continuaremos utilizando o banco MySQL, portanto também precisamos copiar o arquivo `.jar` correspondente ao driver para o diretório `lib` de sua aplicação e adicioná-lo no *classpath*.

Caso você esteja fazendo esse procedimento em casa do zero, há um passo a passo detalhado no blog da Caelum para configurar o ambiente com o Hibernate 3.5: http://blog.caelum.com.br/2010/04/14/as-dependencias-do-hibernate-3-5/

13.3 - Mapeando uma classe Produto para nosso Banco de Dados

Para este capítulo, utilizaremos uma classe que representa um produto:

```
package br.com.caelum.hibernate;

public class Produto {

    private Long id;
    private String nome;
    private String descricao;
    private double preco;
    private Calendar dataInicioVenda;

    // adicione seus getters e setters aqui!
}
```

Aqui criaremos os getters e setters para manipular o objeto, mas fique atento que só devemos criar esses métodos se realmente houver necessidade.

Essa é uma classe como qualquer outra que aprendemos a escrever em Java. Agora precisamos configurar o Hibernate para que ele saiba da existência dessa classe, e desta forma saiba que deve inserir uma linha na tabela `Produto` toda vez que for requisitado que um objeto desse tipo seja salvo. Em vez de usarmos o termo "configurar", falamos em **mapear** uma classe ao banco.

Para mapear a classe `Produto`, 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 três anotações:

```
@Entity
public class Produto {

    @Id
    @GeneratedValue
    private Long id;

    private String nome;
    private String descricao;

    private double preco;

    private Calendar dataInicioVenda;
    // metodos...
}
```

`@Entity` indica que essa classe é uma das que queremos que o Hibernate torne "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). Essas anotações precisam dos devidos `import`s, 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 `Produto` será gravada na tabela de nome também `Produto`, e o atributo `preco` em uma coluna de nome `preco` também!

Se quisermos configurações diferentes das convenções, basta usarmos outras anotações, que são completamente opcionais. Por exemplo, para mapear o atributo `preco` numa coluna chamada `preco_total_prod` faríamos:

```
@Column(name = "preco_total_prod", nullable = true, length = 50)
private double preco;
```

De que tipo é a data?

Objetos do tipo `Calendar`, `java.util.Date` e `java.sql.Date` são tratados pelo Hibernate por padrão como sendo um TIMESTAMP, ou seja, no banco de dados será gravado além da data, os minutos, segundos, horas e assim por diante.
Esse padrão pode ser alterado através da anotação `@Temporal` que recebe o formato com que você quer trabalhar.
Por exemplo: `@Temporal(TemporalType.DATE)`.
Outras opções possíveis são:

- `TemporalType.TIME`
- `TemporalType.TIMESTAMP`

13.4 - Configurando o Hibernate com as propriedades do banco

Em qual banco de dados vamos gravar nossos `Produto`s? Qual é o login? Qual é a senha? O Hibernate necessita dessas configurações, e para isso criaremos o arquivo `hibernate.properties`.

Os 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-26.

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.

Uma das maneiras mais práticas é copiar o arquivo de mesmo nome que está no diretório `project/etc`, do hibernate descompactado que você baixou, para dentro do diretório `src` do seu projeto no Eclipse.

Por padrão a configuração está de tal maneira que o hibernate irá usar um banco de dados do tipo HypersonicSQL. Comente as linhas do mesmo (colocando # no começo). Se você copiar tal arquivo, descomente a parte que utiliza o mysql e configure corretamente a mesma, por exemplo:

```
hibernate.dialect = org.hibernate.dialect.MySQLInnoDBDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost/teste
hibernate.connection.username = root
hibernate.connection.password =
```

Hoje em dia é mais comum encontrarmos essa configuração dentro de um arquivo chamado `hibernate.cfg.xml`, formatado hierarquicamente. Você deve usar apenas uma das abordagens, e escolhemos aqui mostrar pelo `.properties` pela legibilidade da apostila, apesar da nossa preferência pelo XML por ter se tornado praticamente uma convenção.

13.5 - Criando nosso banco com o Hibernate

Para usar o Hibernate 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 Hibernate leia a nossa configuração: tanto o nosso arquivo `properties` quanto as anotações que colocamos na nossa entidade `Produto`. Para tal, usaremos a classe `AnnotationConfiguration` e registramos nossa entidade:

```
AnnotationConfiguration cfg = new AnnotationConfiguration();
cfg.addAnnotatedClass(Produto.class);
```

Há ainda a possibilidade de registrar nossa classe `Produto` para dentro do arquivo de configuração do Hibernate. Nesse caso, não podemos esquecer de invocar o método `configure` da `AnnotationConfiguration`.

Configurando o Hibernate com XML

Por padrão, o Hibernate procura o properties. Algumas pessoas preferem configurar o Hibernate através de um XML. Para carrega-lo, além de ter o XML, você precisa invocar o método `configure()` da sua `AnnotationConfiguration`.

Estamos prontos para usar o Hibernate. Antes de gravar um `Produto`, 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.

A classe `SchemaExport` é a responsável por isto. Ela possui um método `create` que recebe dois argumentos booleanos: o primeiro diz se desejamos ver o código do schema e o segundo se desejamos executá-lo:

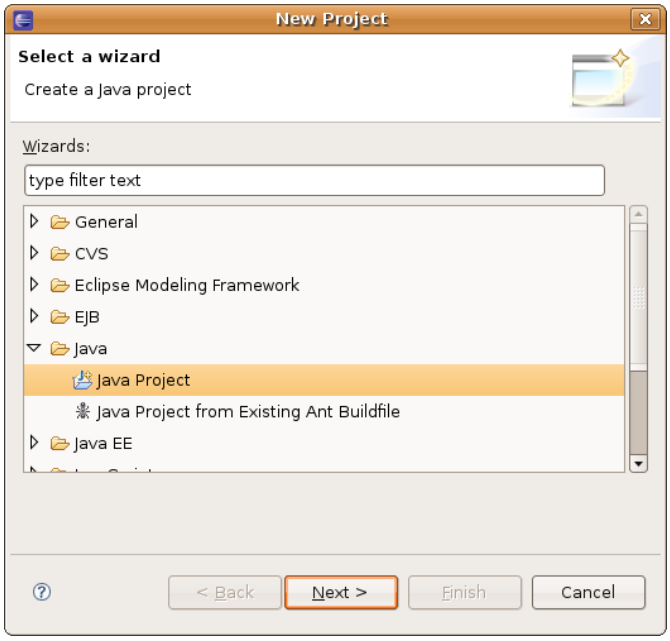
```
SchemaExport se = new SchemaExport(cfg);
se.create(true, true);
```

13.6 - Exercícios: preparando nosso projeto para o Hibernate

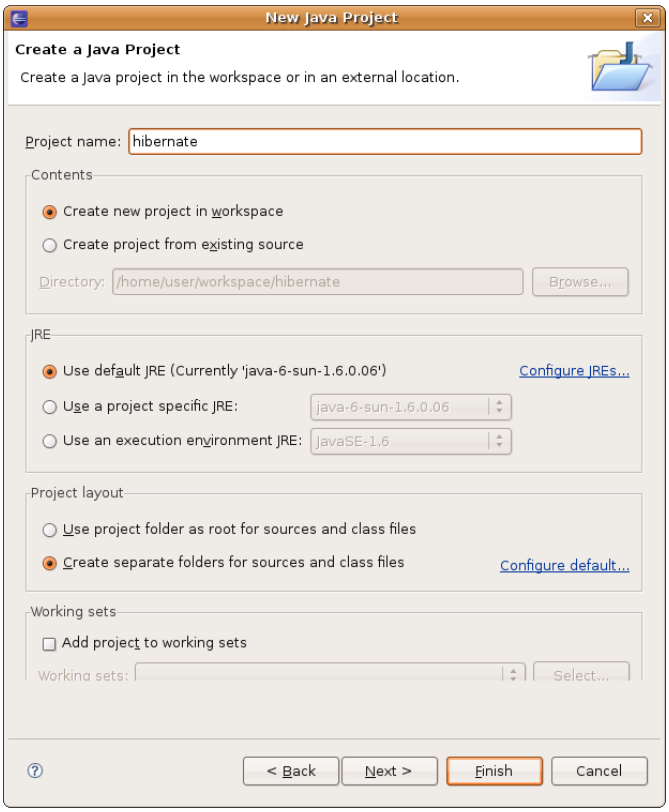
1) Crie um novo projeto:

   a) Vá em *File -> New -> Project*.

   b) Escolha a opção *Java Project*.



   c) Digite **hibernate** como nome do projeto e clique em *Finish*. Confirme a mudança de perspectiva.



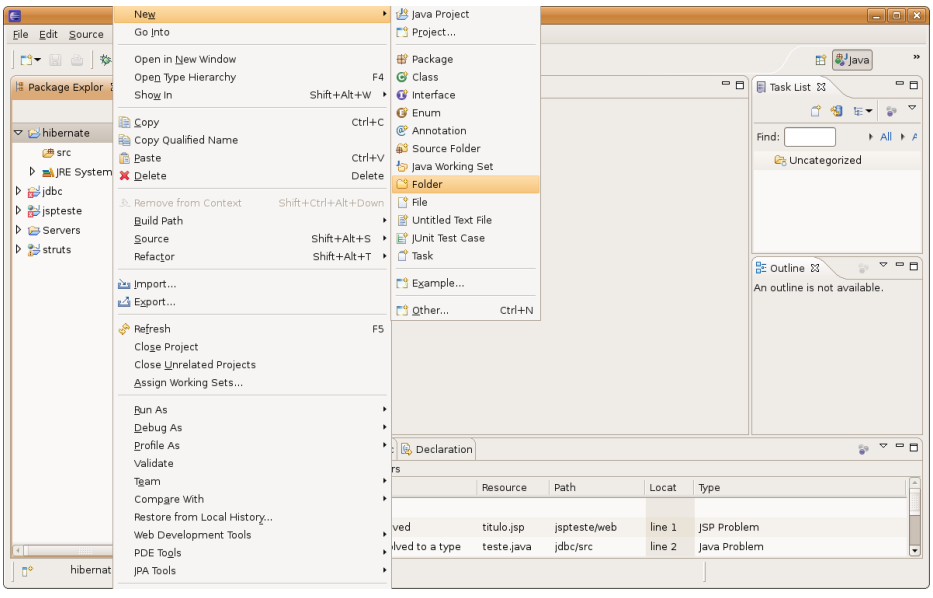
2) Vamos preparar nosso projeto com as dependências que o Hibernate precisa

   Caso você esteja fazendo esse passo de casa, pode seguir o passo a passo descrito no blog da Caelum:

   http://blog.caelum.com.br/2010/04/14/as-dependencias-do-hibernate-3-5/

   Na Caelum, Copie os jars do hibernate para a pasta **lib** do seu projeto dessa forma:

   a) Clique com o botão direito no nome do projeto do hibernate e escolha *New -> Folder*. Escolha **lib** como nome dessa pasta.



   b) Entre no diretório caelum, clicando no icone da caelum no seu Desktop;

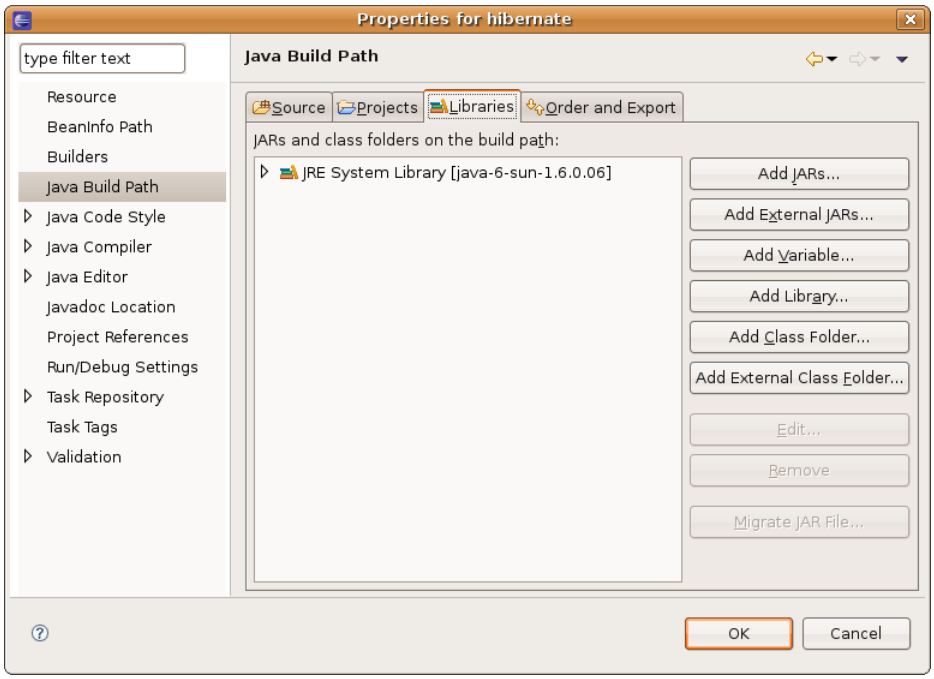
c) Entre no diretório 21 e depois no diretório Hibernate:
d) Selecione todos os arquivos, clique com o botão direito e escolha *Copy*;
e) Cole todos os jars na pasta *workspace/hibernate/lib*

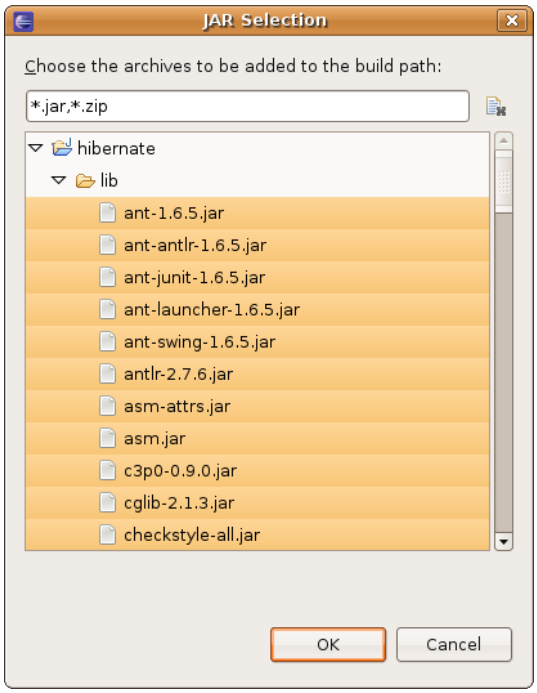
3) Copie o jar do driver do mysql para a pasta lib do projeto também.

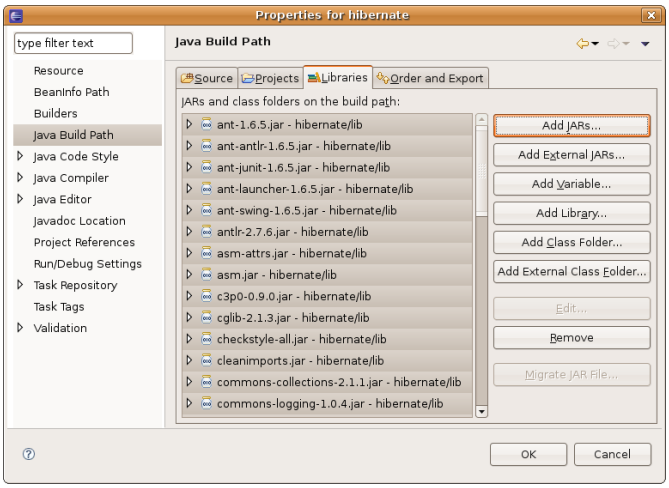
a) Entre no diretório caelum/21;
b) Selecione o driver mais novo do mysql, clique com o botão direito e escolha *Copy*;
c) Cole-o na pasta *workspace/hibernate/lib*

4) Vamos adicionar os jars no classpath do eclipse:

a) Clique da direita no nome do seu projeto, escolha Refresh.
b) Clique novamente da direita, escolha o menu *Properties*;
c) Escolha a opção *Java Build Path*;
d) Escolha a aba *Libraries*;
e) Escolha a opção *Add JARs* e selecione todos os jars do diretório `lib`;







5) Crie uma classe chamada `Produto` no pacote `br.com.caelum.hibernate`.

6) Adicione as seguintes variáveis membro:

```
private Long id;

private String nome;
private String descricao;
private double preco;
private Calendar dataInicioVenda;
```

7) Gere os getters e setters usando o eclipse. (*Source -> Generate Getters and Setters* ou *Ctrl + 3 -> ggas*)

8) Anote a sua classe como uma entidade de banco de dados. Lembre-se que essa é uma anotação do pacote `javax.persistence`.

```
@Entity
```

```
public class Produto {

}
```

9) Anote seu atributo `id` como chave primária e como campo de geração automática:

```
@Id

@GeneratedValue
private Long id;
```

13.7 - Exercícios: configurando e gerando o schema do banco

1) Crie o arquivo **hibernate.properties** no seu diretorio **src**, com as configurações necessárias:

```
hibernate.dialect = org.hibernate.dialect.MySQLInnoDBDialect

hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost/fj21
hibernate.connection.username = root
hibernate.connection.password =
```

2) Crie a classe `GeraTabelas` no pacote `br.com.caelum.hibernate`.

```
package br.com.caelum.hibernate;


// imports omitidos

public class GeraTabelas {
    public static void main(String[] args) {
        AnnotationConfiguration cfg = new AnnotationConfiguration();
        cfg.addAnnotatedClass(Produto.class);

        SchemaExport se = new SchemaExport(cfg);
        se.create(true, true);
    }
}
```

Caso você opte por usar a configuração do hibernate via XML, deve invocar o `cfg.configure()`.

3) Adicione as seguintes linhas no seu arquivo `hibernate.properties`.

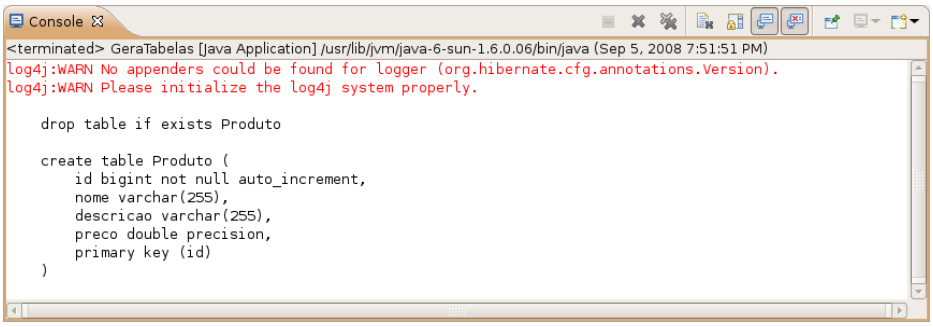
```
hibernate.show_sql = true

hibernate.format_sql = true
```

Essas linhas fazem com que todo sql gerado pelo hibernate apareça no console.

4) Se você não está usando um banco de dados que já existe, como fizemos no começo do curso com o `fj21`, crie o banco no mysql logando no cliente e executando `create database fj21`.

Agora crie suas tabelas executando o código anterior. Clique da direita no código e vá em *Run As -> Java Application*.

O Hibernate deve reclamar que não configuramos nenhum arquivo de log para ele (dois warnings) e mostrar o código SQL que ele executou no banco.



5) 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, crie o 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.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{HH:mm:ss} %5p [%c{1}] %m%n

log4j.rootLogger=warn, stdout
log4j.logger.org.hibernate=info
```

   Ou você pode copiar o `log4j.properties` que já vem na pasta da Caelum com essa configuração.

   Para conhecedores do Log4J, se preferir você pode optar pelo uso da configuração através do `log4j.xml`

13.8 - Trabalhando com os objetos: a Session

Para se comunicar com o Hibernate, precisamos de uma instância de um objeto do tipo `Session`. Adquirimos uma `Session` através de uma fábrica. Essa por sua vez é criada a partir da `AnnotationConfiguration`:

```
AnnotationConfiguration cfg = new AnnotationConfiguration();
cfg.addAnnotatedClass(Produto.class);

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

Esse código seria muito frequente na nossa aplicação, então vamos encapsulá-lo numa classe `HibernateUtil` que cuidará de obter uma única `SessionFactory` e disponibilizar um método que abra as `Session`s quando necessitarmos:

```
1 public class HibernateUtil {
2
```

```
3     private static SessionFactory factory;
4
5     static {
6         AnnotationConfiguration cfg = new AnnotationConfiguration();
7         cfg.addAnnotatedClass(Produto.class);
8         factory = cfg.buildSessionFactory();
9     }
10
11    public Session getSession() {
12        return factory.openSession();
13    }
14 }
```

O bloco estático das linhas 5 a 9 cuidará de configurar o Hibernate e pegar uma `SessionFactory`. Lembre-se que o bloco estático é executado automaticamente quando a classe é carregada pela JVM. O método `getSession` devolverá uma `Session`.

É muito importante saber que uma `Session` deve ter seu ciclo de vida tratado com extremo cuidado. Ela é um objeto “caro": consome recursos importantes, como a conexão do banco, caches, etc. É fundamental tomar cuidado com ela, como exemplo lembre-se sempre de fechar suas sessões, verificar se as transações foram comitadas com sucesso e assim por diante. Por isso no dia a dia o indicado é usar um framework que já tenha filtros para trabalhar com a `Session`, muitos deles vão até *injetar* esse objeto para você, facilitando ainda mais o seu trabalho, como é o caso do VRaptor e do Spring.

> **Cuidados ao usar o Hibernate**
>
> Ao usar o Hibernate profissionalmente, fique atento com alguns cuidados que são simples, mas não dada a devida atenção podem gerar gargalos de performance. Abrir `Session`s indiscriminadamente é um desses problemas. Esse post da Caelum indica outros:
> http://blog.caelum.com.br/2008/01/28/os-7-habitos-dos-desenvolvedores-hibernate-e-jpa-altamente-e
> Esses detalhes do dia a dia assim como Hibernate avançado são vistos no curso FJ-26.

Salvando novos objetos

Através de um objeto do tipo `Session` é possível gravar novos objetos no banco. Para isto basta utilizar o método `save` dentro de uma transação:

```
Produto p = new Produto();
p.setNome("Nome aqui");
p.setDescricao("Descrição aqui");
p.setPreco(100.50);

Session session = new HibernateUtil().getSession();
session.beginTransaction();

session.save(p);

session.getTransaction().commit();
System.out.println("ID do produto: " + p.getId());
session.close();
```

13.9 - Exercícios: o HibernateUtil e gravando objetos

1) Crie a sua classe `HibernateUtil` no pacote `br.com.caelum.hibernate`. No momento de importar `Session` lembre-se que **não** é a *classic*!

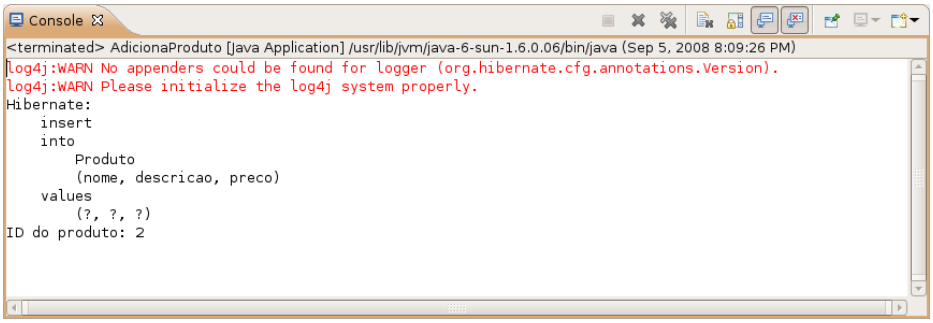
```
package br.com.caelum.hibernate;


// imports omitidos

public class HibernateUtil {

    private static SessionFactory factory;

    static {
        AnnotationConfiguration cfg = new AnnotationConfiguration();
        cfg.addAnnotatedClass(Produto.class);
        factory = cfg.buildSessionFactory();
    }

    public Session getSession() {
        return factory.openSession();
    }
}
```

2) Crie uma classe chamada `AdicionaProduto` no pacote `br.com.caelum.hibernate`, ela vai criar um objeto e adicioná-lo ao banco:

```
package br.com.caelum.hibernate;


// imports omitidos

public class AdicionaProduto {

    public static void main(String[] args) {

        Produto p = new Produto();
        p.setNome("Nome aqui");
        p.setDescricao("Descricao aqui");

        Calendar data = Calendar.getInstance();
        data.set(Calendar.DATE, 11);
        data.set(Calendar.MONTH, Calendar.FEBRUARY);
        data.set(Calendar.YEAR, 2004);

        p.setDataInicioVenda(data);
        p.setPreco(100.50);

        Session session = new HibernateUtil().getSession();
        session.beginTransaction();

        session.save(p);

        session.getTransaction().commit();
```

```
            System.out.println("ID do produto: " + p.getId());
            session.close();
        }
    }
```

3) Rode a classe e adicione alguns produtos no banco. Saída possível:



13.10 - Criando um DAO para o Hibernate

Com a mesma facilidade que gravamos um objeto usando o Hibernate, podemos executar as operações básicas de leitura, atualização e remoção.

Para buscar um objeto dada sua chave primária, no caso o seu `id`, utilizamos o método `load`, conforme o exemplo a seguir:

```
Session session = new HibernateUtil().getSession();
Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println(encontrado.getNome());
```

Com os métodos que já conhecemos, podemos criar uma classe `DAO` para o nosso `Produto` que utiliza o Hibernate:

```
public class ProdutoDAO {

    private Session session;

    public ProdutoDAO (Session session) {
        this.session = session;
    }

    public void salva (Produto p) {
        this.session.save(p);
    }

    public void remove (Produto p) {
        this.session.delete(p);
    }

    public Produto procura(Long id) {
        return (Produto) this.session.load(Produto.class, id);
    }
```

```
    public void atualiza (Produto p) {
        this.session.update(p);
    }
}
```

Através desse `DAO`, podemos salvar, remover, atualizar e procurar Produtos. Repare como ele está muito mais simples que se fossemos utilizar JDBC diretamente. Mais ainda: se adicionarmos novos atributos a classe `Produto`, não precisamos alterar nada nosso DAO, pois as queries são dinamicamente criadas pelo Hibernate.

13.11 - Exercícios: criando o DAO

1) Crie uma classe chamada `ProdutoDAO` dentro do pacote `br.com.caelum.hibernate.dao`

```
public class ProdutoDAO {


    private Session session;

    public ProdutoDAO(Session session) {
        this.session = session;
    }

    public void salva(Produto p) {
        this.session.save(p);
    }

    public void remove(Produto p) {
        this.session.delete(p);
    }

    public Produto procura(Long id) {
        return (Produto) this.session.load(Produto.class, id);
    }

    public void atualiza(Produto p) {
        this.session.update(p);
    }
}
```

2) Crie uma classe `TestaProdutoDAO` no pacote `br.com.caelum.hibernate`, e adicione diversos objetos diferentes no banco de dados usando sua classe `ProdutoDAO`.

   Esse trecho de código precisa estar dentro de um método para testa-lo. Vamos colocar dentro do método `main`, mas varemos no FJ-16 que existem bibliotecas próprias para facilitar a criação de testes (e de menor granularidade) como o JUnit.

```
Session session = new HibernateUtil().getSession();


ProdutoDAO dao = new ProdutoDAO(session);
Produto produto = new Produto();
```

```
//... popule os dados do produto aqui!

session.beginTransaction();
dao.salva(produto);
session.getTransaction().commit();

session.close();
```

3) Busque um `id` inválido, que não existe no banco de dados e descubra qual o erro que o Hibernate gera. Ele retorna `null`? Ele joga uma `Exception`? Qual?

13.12 - Buscando com uma cláusula where

O Hibernate possui uma linguagem própria de queries para facilitar a busca de objetos chamada de **HQL**. Por exemplo, o código a seguir mostra uma pesquisa que retorna todos os produtos com `preco` maior que `10`:

```
Session session = new HibernateUtil().getSession();
List<Produto> lista =
    session.createQuery("select p from Produto as p where p.preco > 10").list();

for (Produto p : lista) {
    System.out.println(p.getNome());
}
```

Vamos incluir mais dois métodos na nossa classe `ProdutoDAO` para conhecer um pouco mais do Hibernate. Em vez de usar a HQL, usaremos o outro mecanismo de consulta do Hibernate, chamado de `Criteria`. Ele tem o mesmo objetivo que a Hql, porém usa uma abordagem totalmente orientada a objetos, em vez de ser baseado em uma única String que define a query.

Primeiro, vamos listar todos os produtos existentes no banco de dados. Para isso, vamos usar o método `createCriteria` de `Session` que cria um `Criteria`. Através de um `Criteria`, temos acesso a diversas operações no banco de dados; uma delas é listar tudo com o método `list()`.

Nosso método `listaTudo()` fica assim:

```
public List<Produto> listaTudo() {
    return this.session.createCriteria(Produto.class).list();
}
```

Mas o método acima devolve a lista com todos os produtos no banco de dados. Em um sistema com listagens longas, normalmente apresentamos a lista por páginas. Para implementar paginação, precisamos determinar que a listagem deve começar em um determinado ponto e ser de um determinado tamanho.

Usando o `Criteria`, como no `listaTudo` anteriormente, isso é bastante simples. Nosso método página fica assim:

```
public List<Produto> pagina(int inicio, int quantia) {
    return this.session.createCriteria(Produto.class)
        .setMaxResults(quantia).setFirstResult(inicio).list();
}
```

O método `setMaxResults` determina o tamanho da lista (resultados por página) e o método `setFirstResult` determina em que ponto a listagem deve ter início. Por fim, basta chamar o método `list()` e a listagem devolvida será apenas daquela página!

Este é apenas o começo, o Hibernate é muito poderoso e no curso FJ-26 veremos como fazer queries complexas, com joins, agrupamentos, projeções, de maneira muito mais simples do que se tivessemos de escrever as queries. Além disso o Hibernate é muito customizável: podemos configura-lo para que ele 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 feito a primeira vista é pensar que o 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.

13.13 - Exercícios

1) Modifique a sua classe `ProdutoDAO` e acrescente os métodos `listaTudo()` e `pagina()`

```
public List<Produto> listaTudo() {

    return this.session.createCriteria(Produto.class).list();
}

public List<Produto> pagina (int inicio, int quantia) {
    return this.session.createCriteria(Produto.class)
            .setMaxResults(quantia).setFirstResult(inicio).list();
}

public List<Produto> precoMaiorQue(double preco) {
    Query query = this.session.createQuery("from Produto where preco > :preco");
    query.setDouble("preco", preco);
    return query.list();
}
```

2) Crie uma classe chamada `TestaBuscas` no pacote `br.com.caelum.hibernate`:

```
public class TestaBuscas {


    public static void main(String [] args){
        Session session = new HibernateUtil().getSession();
        ProdutoDAO produtoDao = new ProdutoDAO(session);

        System.out.println("*****Listando Tudo*******");
        for (Produto p : produtoDao.listaTudo()) {
            System.out.println(p.getNome());
        }

        System.out.println("*****Listando Paginado*******");
        for (Produto p : produtoDao.pagina(2,3)) {
            System.out.println(p.getNome());
```

```
        }

        System.out.println("*****Preços maiores que*******");
        for (Produto p : produtoDao.precoMaiorQue(2.10)) {
            System.out.println(p.getNome());
        }
    }
}
```

13.14 - Exercícios opcionais: para testar o LAZY

1) Mude a propriedade `hibernate.show_sql` para `true` no arquivo `hibernate.properties` e rode a classe acima.

2) Teste um programa que faz somente o seguinte: busca um produto por id. O código deve somente buscar o produto e não imprimir nada! Qual o resultado?

```
Session session = new HibernateUtil().getSession();

Produto encontrado = (Produto) session.load(Produto.class, 1L);
```

3) Tente imprimir o nome do produto do teste anterior, o que acontece?

```
Session session = new HibernateUtil().getSession();

Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println(encontrado.getNome());
```

4) Antes de imprimir o nome do produto, tente imprmir uma mensagem qualquer, do tipo: “O select já foi feito”. E agora? Como isso é possível?

```
Session session = new HibernateUtil().getSession();

Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println("O select já foi feito");
System.out.println(encontrado.getNome());
```

Então, onde está o código do select? Ele deve estar no método `getNome()`, certo?

5) Imprima o nome da classe do objeto referenciado pela variável encontrado:

```
Session session = new HibernateUtil().getSession();

Produto encontrado = (Produto) session.load(Produto.class, 1L);
System.out.println("O select já foi feito");
System.out.println(encontrado.getNome());
System.out.println(encontrado.getClass().getName());
```

O Hibernate retorna um objeto cujo tipo estende `Produto`: ele não deixa de ser um `Produto` mas não é somente um `Produto`.

O método `getNome` foi sobrescrito nessa classe para fazer a busca na primeira vez que é chamado, economizando tempo de processamento.

É claro que para fazer o *fine-tuning* do Hibernate é interessante conhecer muito mais a fundo o que o Hibernate faz e como ele faz isso, veremos mais durante o FJ-26.

13.15 - Exercício opcional

1) Crie um sistema para cadastro e listagem de produtos na web. Siga o padrão que utilizamos para cadastro e listagem de contatos, mas dessa vez usando o Hibernate. Isole seu código no DAO.

CAPÍTULO 14

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?

14.1 - 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:

- **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 também ajudar se estiver pensando na certificação SCWCD (veja mais sobre ela abaixo).
- **Struts 1** - Embora a última versão do Struts seja a 2, com muitas facilidades e melhorias, boa parte do mercado brasileiro ainda está preso ao Struts 1. Este apêndice mostra o uso do Struts 1 e é interessante caso você enfrente algum projeto com essa versão do framework.

14.2 - Certificação SCWCD

Entrar em detalhes nos assuntos contidos até agora iriam no mínimo tornar cada capítulo quatro vezes maior do que já é. Os tópicos abordados (com a adição e remoção de alguns) constituem boa parte do que é cobrado na certificação oficial para desenvolvedores Web da Sun, a **SCWCD**.

Para maiores informações sobre certificações consulte a própria Sun, o http://www.javaranch.com ou o http://www.guj.com.br, que possui diversas informações sobre o assunto.

Você pode ver o conteúdo completo da última versão disponível da prova em:

http://www.sun.com/training/certification/java/scwcd.xml

Note que, com o lançamento do Java EE 6, a Sun já iniciou o processo de atualização dos conteúdos da certificação. Ainda vai demorar um pouco para isso acontecer, mas é bom ficar atento às possíveis mudanças.

14.3 - 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 MVC é tratado no curso **FJ-27**:

http://www.caelum.com.br/curso/fj26

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 Struts 2 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.

14.4 - 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 Hibernate avançado no curso **FJ-26**:

http://www.caelum.com.br/curso/fj26

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.

14.5 - 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.

Muitos programadores com o mínimo ou máximo de conhecimento se reúnem online para a troca de dúvidas, informações e idéias sobre projetos, bibliotecas e muito mais. Um dos mais importantes e famosos no Brasil é o GUJ – http://www.guj.com.br

Bons estudos!

CAPÍTULO 15

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 idéias;
- 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*.

15.1 - 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!

15.2 - Vantagens de um codigo 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. Agora 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);

    }
}
```

15.3 - 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 Estátistica 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.

15.4 - 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.

15.5 - Minha primeira lógica de negócios

Vamos agora 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.

15.6 - 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.

15.7 - 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.

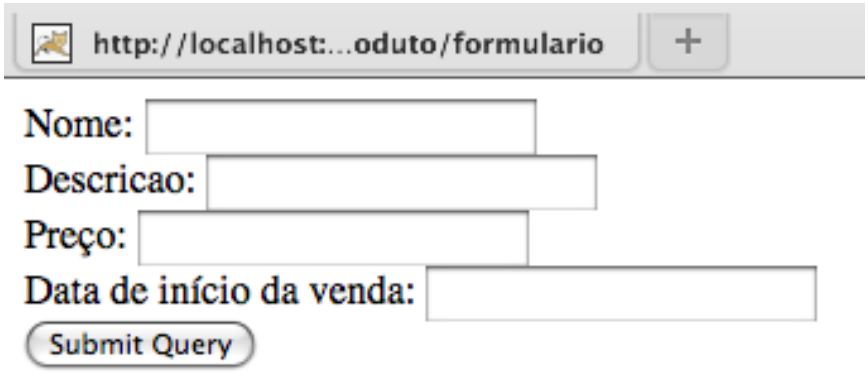
Vamos tomar uma abordagem diferente, pensando desde o começo que futuramente precisaremos de uma lógica executando antes do formulário, vamos criar uma lógica vazia, que nada mais fará do que repassar a execução ao formulário através da convenção do `VRaptor`.

```
@Resource
public class ProdutoController {
    public void formulario() {
    }
}
```

Agora 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>
```



15.8 - 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 unitários 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:

15.9 - 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) Associe o projeto com o Tomcat e acesse no seu navegador a URL: http://localhost:8080/controle-produtos

4) Vamos criar a parte de listagem dos produtos.

   a) 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();
    }

}
```

   b) Crie o arquivo `lista.jsp` no diretório `WebContent/WEB-INF/jsp/produto`

```
<%@ 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>
```

c) Teste a sua lógica: http://localhost:8080/controle-produtos/produto/lista.

5) Vamos criar a funcionalidade de adicionar produtos.

a) Na classe `ProdutoController` crie o método `adiciona`:

```
@Resource

public class ProdutoController {

    // a ação adiciona
    public void adiciona(Produto produto) {
        new ProdutoDao().adiciona(produto);
    }

}
```

b) 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();
    }

}
```

c) Vamos criar a lógica para o formulário

```
@Resource

public class ProdutoController {

    public void formulario() {
    }

}
```

d) 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="<c:url value="/js/jquery.js"/>"></script>
            <script type="text/javascript" src="<c:url value="/js/jquery-ui.js"/>"></script>
            <link type="text/css" href="<c:url value="/css/jquery.css"/>" rel="stylesheet" />
        </head>
        <body>
            <form action="<c:url value="/produto/adiciona"/>">
                Nome: <input name="produto.nome"/><br/>
                Descricao: <input name="produto.descricao"/><br/>
                Preço: <input name="produto.preco"/><br/>
                Data de início da venda:
                <caelum:campoData id="dataInicioVenda" name="produto.dataInicioVenda"/>
                <br />
                <input type="submit"/>
            </form>
        </body>
    </html>
```

e) 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`

f) Você importou alguma classe do pacote `javax.servlet`? É importante percebermos agora que não estamos atrelados com nenhuma API estranha. Apenas escrevemos código Java normal.

g) Teste o seu formulário: http://localhost:8080/controle-produtos/produto/formulario. Adicione alguns produtos diferentes.

6) Agora vamos fazer a exclusão de produtos.

a) 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="${produto.dataInicioVenda.time}"</td>
        <td><a href="<c:url value="/produto/remove"/>?produto.id=${produto.id}">Remover</a></td>
    </tr>
</c:forEach>
```

b) 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();
}
```

c) Acesse a lista de produtos em http://localhost:8080/controle-produtos/produto/lista e remova alguns produtos.

7) (Opcional) Faça a funcionalidade de alteração do produto da forma que achar conveniente

15.10 - 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 unitários, assunto coberto no curso **FJ-16**.

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 instânciá-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-dia.

15.11 - 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 agora nossa aplicação irá parar de funcionar. Isso porque precisamos dizer para o `VRaptor` que `ProdutoDao` é uma classe que ele irá 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 testa-lo, como veremos no FJ-16 e FJ-28.

```
@Component
public class ProdutoDao {

    //construtor e métodos do Dao
}
```

15.12 - 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 instânciado 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 idéias do VRaptor é o de não re-inventar a roda e re-utilizar 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.

15.13 - Exercícios: Usando Injeção de Dependências para o DAO

1) Vamos diminuir o acoplamento do `ProdutoController` com o `ProdutoDao` recebendo-o no construtor.

   a) 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
}
```

   b) Agora, 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
    }
}
```

c) Reinicie sua aplicação e verifique que tudo continua funcionando normalmente.

15.14 - 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();
    }
}
```

Agora que 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 agora 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();
        }
    }
}
```

Agora 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!

15.15 - 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, agora temos nossa funcionalidade de autenticação e também a parte de autorização finalizadas.

15.16 - Exercícios: Construindo a autenticação e a autorização

1) Vamos permitir que nossos usuários possam efetuar Login na aplicação.

a) 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;
    }
}
```

b) 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
}
```

c) Agora 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() {
    }
}
```

d) 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="<c:url value="/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>
```

e) Agora 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);
    }
}
```

f) Agora 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.

g) 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');
```

15.17 - 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 re-gerar a tela inteira. Não seria mais leve apenas removermos a linha da tabela que acabamos de excluir ao invés de re-criarmos 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.

15.18 - Para saber mais: Requsições: 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 es-

tamos acostumados é o que chamamos de *Requisições Síncronas*, na qual o navegador ao enviar a requisição interrompe a navegação e re-exibe 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 re-carregar toda a página).

15.19 - 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 a complexidade nos fornecem uma API agradável para trabalhar, como por exemplo, JQuery, YUI (Yahoo User Interface), ExtJS e assim por diante.

15.20 - 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>
    <!-- continuacao da pagina -->
```

Agora podemos alterar o link para disparar um evento ao ser clicado, e não mais chamar uma URL. O nosso link agora não nos enviará para lugar nenhum, será mais um artifício visual para exibí-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>
```

Agora 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.

Agora 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 composto 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>
```

Agora, em nossa função *Javascript* para remover o produto, podemos também remover a linha da tabela com o seguinte código:

```
$('#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.

15.21 - Exercícios opcionais: Adicionando AJAX na nossa aplicação

1) a) 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="<c:url value="/js/jquery.js"/>"></script>
    </head>
    <body>
    <!-- continuacao da pagina -->
```

b) Altere os links para chamar uma função chamada `removeProduto`:

```
<td><a href="#" onclick="return removeProduto(${produto.id})">Remover</a></td>
```

c) Agora 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>

    <!-- continuacao da pagina -->
</body>
```

d) 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 -->
```

e) Dê um `id` para os `<tr>`, dessa forma você poderá apagá-los:

```
<tr id="produto${produto.id}">
```

f) Remova o redirecionamento do método `remove` do `ProdutoController`, de forma que ele fique como a seguir:

```
public void remove(Produto produto) {

    produtoDao.remove(produto);
}
```

g) Por fim, crie no diretório `WEB-INF/jsp/produto` o arquivo `remove.jsp` com o seguinte conteúdo:

```
Produto removido com sucesso
```

h) Acesse novamente a listagem dos produtos e faça a remoção deles. Perceba que a página não é recarregada quando você clica no *link*.

CAPÍTULO 16

Apêndice - Servlets 3.0 e Java EE 6

"Nove pessoas não fazem um bebê em 1 mês"
– Fred Brooks

16.1 - 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 um 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 Entrepise 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.

16.2 - Suporte a anotações: @WebServlet

Como foi visto durante o curso, criar Servlets utilizando o Java EE 5 é um processo muito trabalhoso. Um dos grandes problemas é que temos que configurar cada um de nossos servlets no web.xml e se quisermos acessar esse servlet de maneiras diferentes, temos que criar vários mapeamentos para o mesmo servlet, o que pode com o tempo se tornar um problema devido a difícil manutenção.

Configuração de um servlet no web.xml:

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

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

Na nova especificação Servlets 3.0, que faz parte do Java EE 6, podemos configurar a maneira como vamos acessar o nosso servlet de maneira programática, utilizando anotações simples:

```
@WebServlet("/oi")
public class OlaServlet 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 o servlet (equivalente ao `servlet-name`). Se não definirmos esse atributo, por padrão, o nome do servlet é o nome completo da classe do seu servlet, também conhecido como *Fully Qualified Name*.

Para definirmos como iremos acessar aquele servlet, existem duas maneiras. Se quisermos que nosso servlet seja acessado atráves de apenas uma URL, é recomendado definirmos a URL diretamente no atributo `value` como no exemplo acima. Se precisarmos definir mais de uma URL para acessar nosso servlet, colocamos no atributo `urlPatterns`:

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

Não é permitido que ambos atributos sejam usados em conjunto, porém, pelo menos um deles **deve** ser configurado.

Arquivo web.xml

Dentro da tag `<web-app>` no web.xml, existe agora um novo atributo chamado `metadata-complete`. Nesse atributo podemos configurar se nossas classes anotadas com `@WebServlet` ou `@WebFilter` 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.

16.3 - Suporte a anotações: @WebFilter

Para criarmos filtros utilizando a API de Servlets do Java EE 5, temos as mesmas dificuldades que temos quando vamos definir `Servlet`s. 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>
```

Agora, 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`:

```
@WebFilter(“/oi”)
public class MeuFiltro implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) {
        //...
    }
}
```

Se não definirmos um atributo `name` para nosso filtro, o nome dele será o nome completo da classe, da mesma forma que acontece com as `Servlet`s. 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, a passamos atráves do atributo `name` na anotação `@WebFilter` como foi feito no exemplo acima. Mas, se quisermos definir que mais de uma URL será filtrada, podemos usar o atributo `urlPatterns`. Porém, assim como na `@WebServlet` na anotação `@WebFilter` não podemos definir ambos os atributos.

```
@WebFilter(name = "MeuFiltro", ulrPatterns = {"/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(name = "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 **<listener>** 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 **<init-param>** de hoje;
- **@MultipartConfig** - Utilizada para definir que um determinado `Servlet` receberá uma requisição do tipo `mime/multipart`.

16.4 - Preparando o Glassfish v3.0 em casa

Baixe o Glassfish v3.0 em https://glassfish.dev.java.net/public/downloadsindex.html, você também precisará de uma versão "Binary Distribution". Baixe a versão zipada por questões de comodidade, afinal ele é feito inteiramente em java, portanto, a mesma versão roda em qualquer plataforma.

O Glassfish v3.0 é a implementação de referência da Sun para o Java EE 6. Para conseguirmos fazer os exercícios devemos utilizá-lo. Depois de baixar o zip, basta descompactá-lo e pronto! Já instalamos o Glassfish.

Para iniciá-lo entre no diretório de instalação, depois em na pasta glassfish/bin e execute o script `startserv`:

```
cd glassfishv3/glassfish/bin
./startserv
```

Para pará-lo entre no diretório de instalação, depois em na pasta glassfish/bin e execute o script `stopserv`:

```
cd glassfishv3/glassfish/bin
./stopserv
```

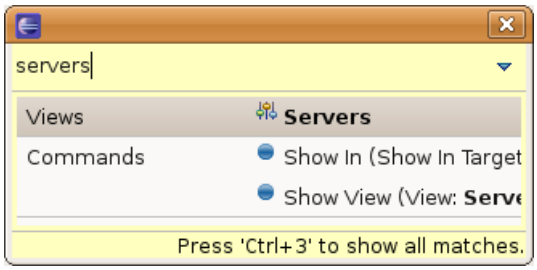
Como já vimos durante o curso não precisaremos fazer isso manualmente. Podemos nos beneficiar do uso dos plugins do WTP.

16.5 - Preparando o Glassfish v3.0 no WTP

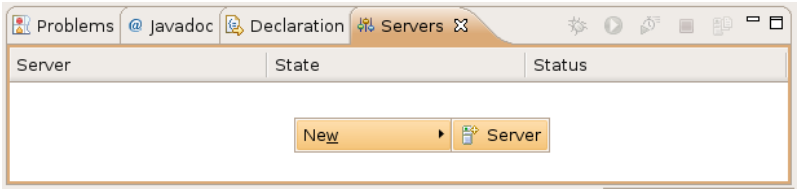
Vamos agora configurar no WTP o servidor Glassfish que acabamos de descompactar.

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

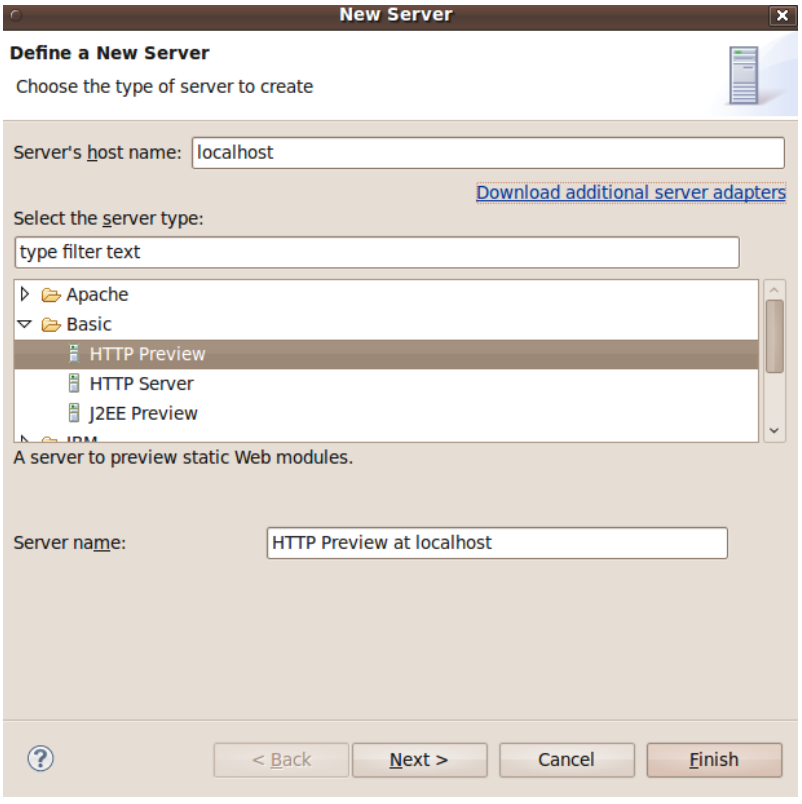
2) Abra a *View* de **Servers** na perspectiva atual. Aperte **Ctrl + 3** e digite **Servers**:



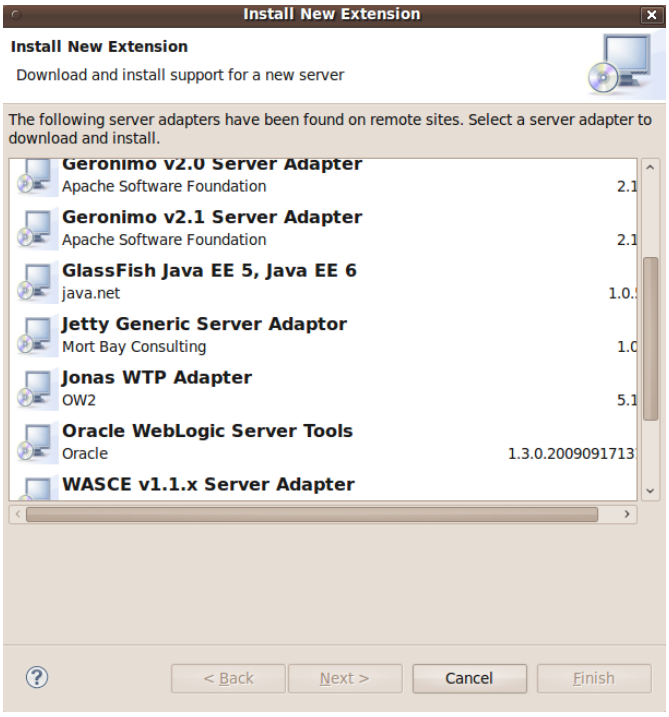
3) Clique com o botão direito dentro da aba Servers e vá em **New** > **Server**:



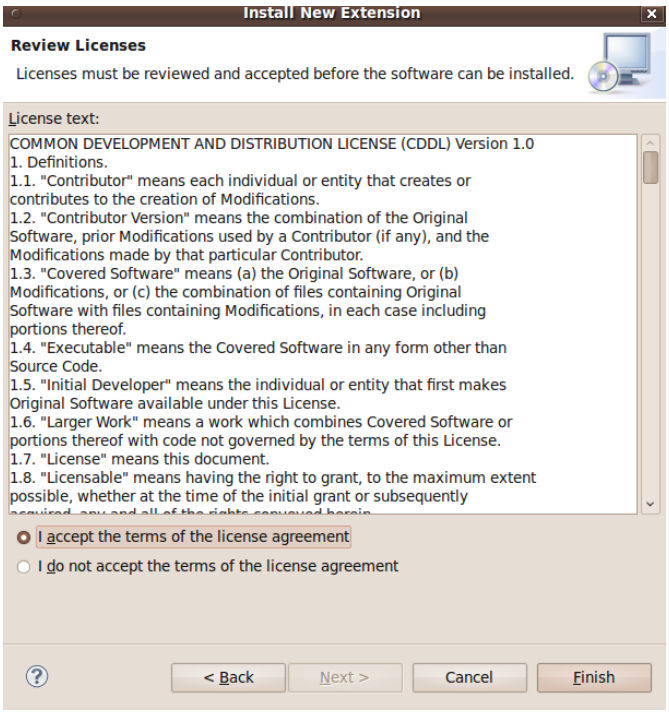
4) Não temos a opção de adicionar um servidor do tipo Glassfish, então vamos clicar em **Download additional server adapters**:



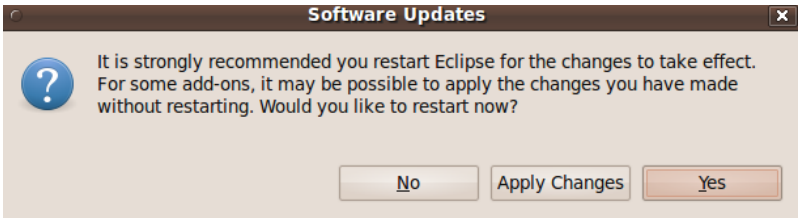
5) Na próxima tela, selecione **Glassfish Java EE 5, Java EE 6** e clique em **Next**:



6) Agora aceite os termos e clique em **Finish**:

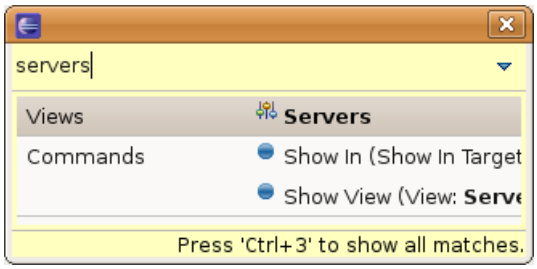


7) Ao final da instalação clique em **Restart**:

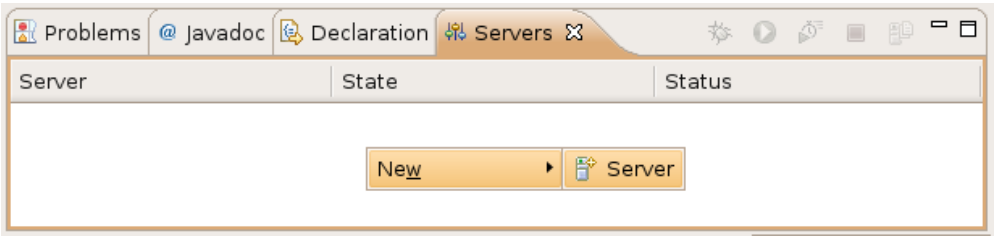


Pronto! Agora estamos aptos a adicionar servidores do tipo Glassfish.

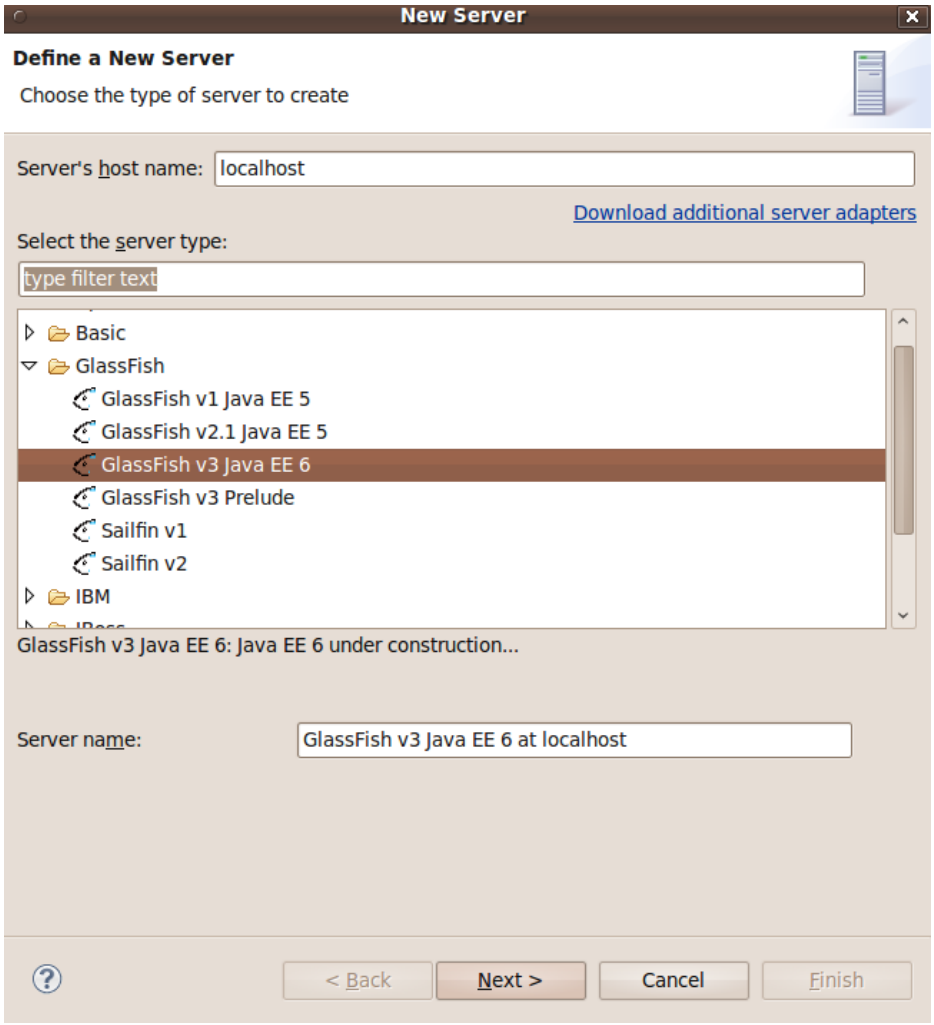
8) Abra novamente a *View* de **Servers** na perspectiva atual. Aperte **Ctrl + 3** e digite **Servers**:



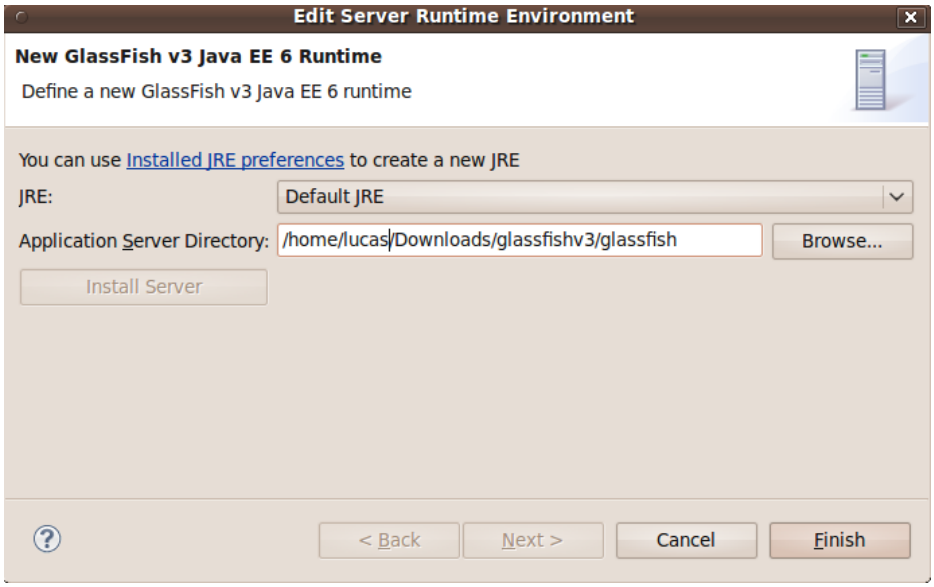
9) Clique com o botão direito dentro da aba Servers e vá em **New** > **Server**:



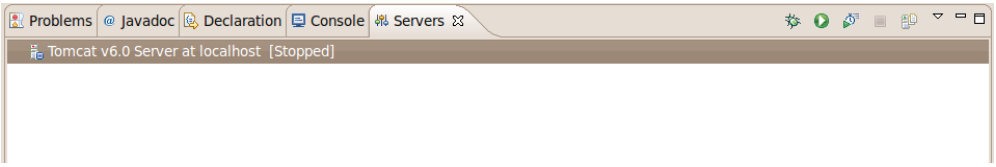
10) Selecione o **GlassFish v3 Java EE 6** e clique em **Next**:



11) Na próxima tela, selecione o diretório **glassfish** dentro da pasta onde você descompactou o Glassfish e clique em **Finish**:



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



13) Abra o navegador e acesse a URL http://localhost:8080/ Deve aparecer uma tela do glassfish.

Pronto! Conseguimos agora gerenciar o Glassfish através do Eclipse!

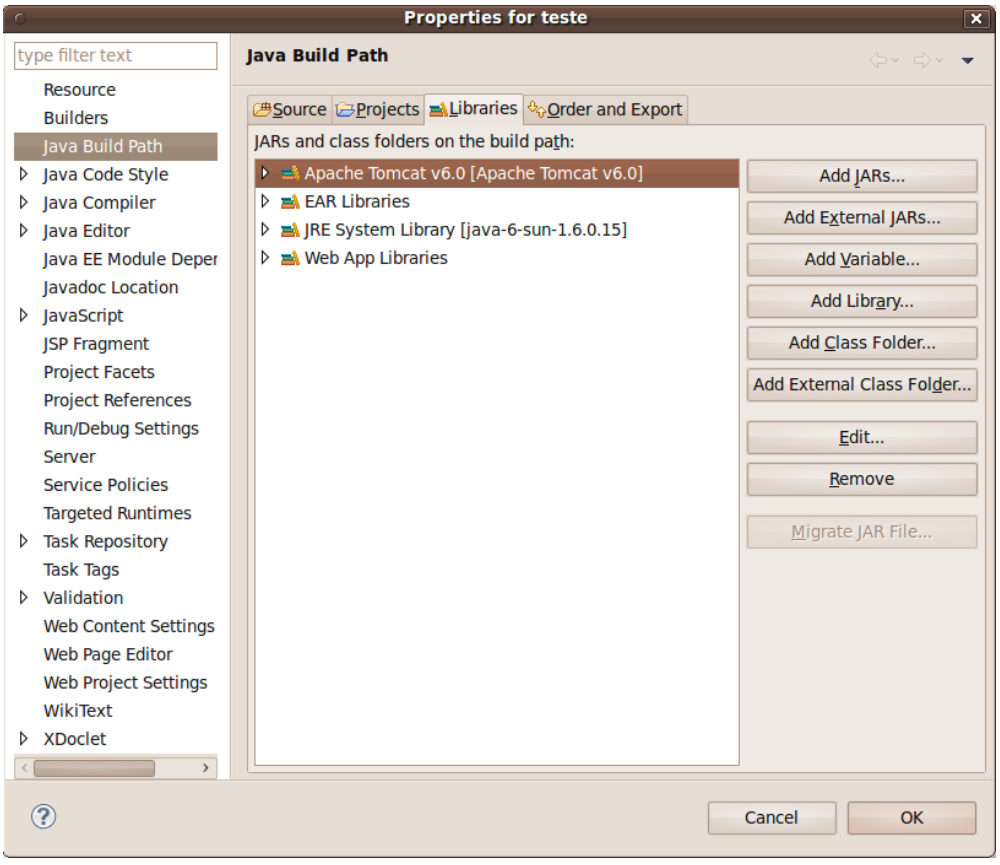
16.6 - Nossa aplicação usando o Glassfish

Até o momento estamos utilizando implementações de Servlets provenientes do `Tomcat`, no entanto ele ainda não implementa a versão 3.0 da especificação de Servlets.

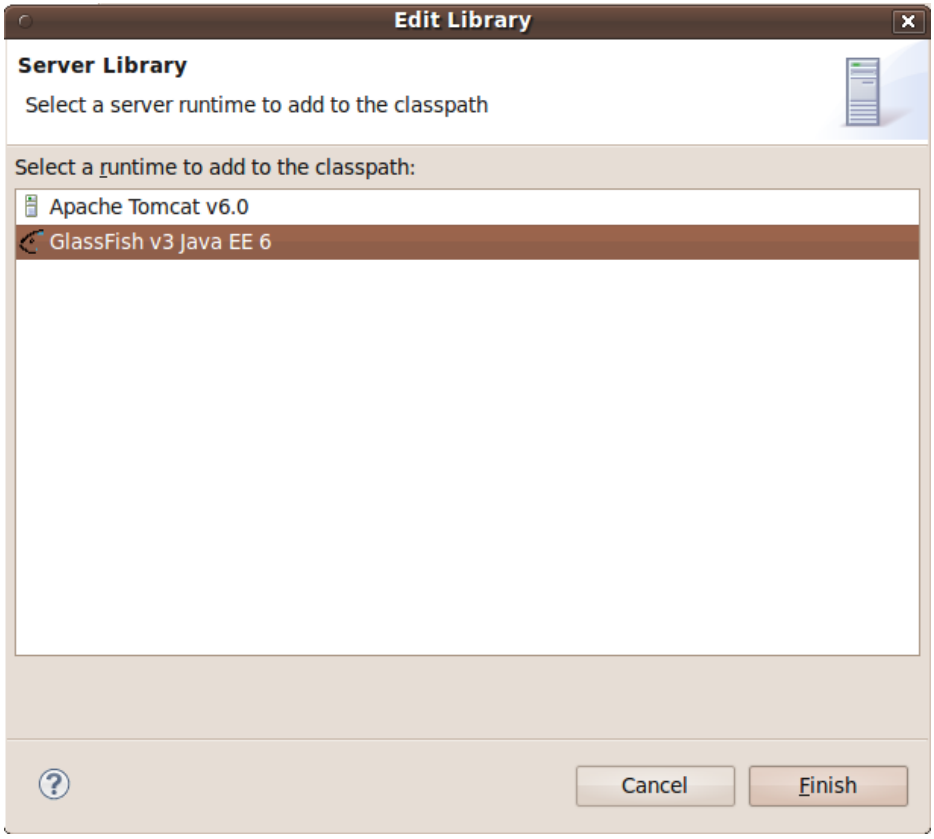
Para fazermos os próximos exercícios utilizando a API Servlets 3.0 vamos utilizar o Glassfish v3.0 que implementa a nova especificação.

1) Selecione o seu projeto **fj21-agenda**, clique da direita e escolha Build Path, Configure Build Path.

2) Clique na aba **Libraries**, selecione a biblioteca **Apache Tomcat v6.0** e clique em **Edit...**:



3) Agora selecione a bilioteca **Glassfish v3 Java EE 6** e clique em **Finish**:



4) Voltando a tela de configuração do build path, clique em **OK**;

A partir de agora estamos prontos para fazermos as mudanças para Servlets 3.0.

16.7 - Exercício: Usando anotação @WebServlet

1) Escolha o menu **File**, **New**, **Class**. Crie a servlet `OiMundoServlets3` no pacote `br.com.caelum.servlet`.

a) Estenda `HttpServlet`:

```
public class OiMundoServlets3 extends HttpServlet {

}
```

b) Coloque a anotação `@WebServlet` acima da declaração da classe.

```
@WebServlet

public class OiMundoServlets3 extends HttpServlet {
}
```

c) Na anotação adicionada, especifique o atributo `value`, que define como o servlet será acessado.

```
@WebServlet(value = "/olamundo-servlets3")

public class OiMundoServlets3 extends HttpServlet {
}
```

d) Sobrescreva o método `service`

```
@Override

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

}
```

e) Escreva a implementação do método `service` **dentro** dele.

```
1 protected void service(HttpServletRequest request,

2         HttpServletResponse response) throws ServletException, IOException {
3     // recebe o writer
4     PrintWriter out = response.getWriter();
5
6     // escreve o texto
7     out.println("<html>");
8     out.println("<body>");
9     out.println("Oi mundo usando Servlets 3.0!");
10    out.println("</body>");
11    out.println("</html>");
12 }
```

2) Reinicie o Glassfish clicando no botão verde na aba Servers.

3) Teste a url http://localhost:8080/fj21-agenda/olamundo-servlets3

* Sem nenhuma configuração em nosso arquivo web.xml, conseguimos acessar e criar um `Servlet` através de simples anotações.

16.8 - Exercício: Alterando nosso framework MVC

1) Vamos alterar a servlet `ControllerServlet` do nosso pequeno framework MVC para que ele funcione a partir de anotações.

a) Abra a classe `ControllerServlet` (utilize o atalho **ctrl+shift+t** para abrí-la) e adicione a anotação `@WebServlet` acima da declaração da classe setando os parâmetros `name` e `value`:

```
@WebServlet(name = "controlador", value = "/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();
                logica.executa(request, response);

            } catch (Exception e) {
                throw new ServletException("A lógica de negócios causou uma exceção", e);
            }
        }
    }
```

b) Não precisamos mais da configuração do `ControllerServlet` em nosso `web.xml`, portanto, podemos remover as seguintes linhas:

```
<servlet>
    <servlet-name>controlador</servlet-name>
    <servlet-class>br.com.caelum.mvc.servlet.ControllerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>controlador</servlet-name>
    <url-pattern>/mvc</url-pattern>
</servlet-mapping>
```

2) Reinicie o Glassfish clicando no botão verde na aba Servers.

3) Feito isso, teste a url http://localhost:8080/fj21-agenda/mvc?logica=PrimeiraLogica e nosso framework continua funcionando normalmente.

16.9 - Exercício: Alterando nosso FiltroConexao

1) Vamos refatorar o `FiltroConexao` utilizando recursos de servlets 3.0 como a anotação `@WebFilter`.

a) Abra a classe `FiltroConexao` e adicione a anotação `@WebFilter` acima da declaração do nosso filtro:

```
@WebFilter(filterName = "FiltroConexao", value = "/*")

public class FiltroConexao implements Filter {
    ...
}
```

b) Neste caso também não necessitamos mais da configuração do `FiltroConexao` em nosso `web.xml` veja que na anotação que adicionamos especificamos o nome do filtro através do atributo `filterName` e também quais URLs serão interceptadas pelo filtro através do atributo `value`

c) Consequentemente, podemos remover as seguintes linhas:

```
<filter>
    <filter-name>FiltroConexao</filter-name>
    <filter-class>br.com.caelum.agenda.filtro.FiltroConexao</filter-class>
</filter>

<filter-mapping>
    <filter-name>FiltroConexao</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

2) Reinicie o Glassfish clicando no botão verde na aba Servers.

3) Faça alguma requisição que seja interceptada pelo `FiltroConexao`, por exemplo, altere um contato já existente em nossa aplicação e veja que nosso filtro continua funcionando como anteriormente, porém agora com 8 linhas a menos de configuração em nosso `web.xml`. Fantástico!

16.10 - 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, agora, 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 agora 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 excedeu o tempo máximo que ela tinha para ser executada sobrescrevemos o método `onTimeout`. Para tratarmos um erro podemos sobrescrever o método `onError`.

Podemos prosseguir configurando nossa 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.

16.11 - 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 agora quando desejamos sobrescrever 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 poupados 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 contâiner 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 descritor `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:

1) `web.xml` - Sempre será processado primeiro

2) Fragmento1

3) 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 irá procurar por essas implementações quando ele for iniciado.

A anotação @HandlesTypes serve para especificar quais classes podem ser manipuladas pela implementação de `ServletContainerInitializer`

16.12 - 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 agora 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.

CAPÍTULO 17

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.

17.1 - 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.

17.2 - 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 irá 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>
```

17.3 - 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>
```

17.4 - 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" %>
```

17.5 - 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"/>
...
```

17.6 - 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()) {
    // faca algo com o parametro
}
```

17.7 - 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.

17.8 - 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>
```

17.9 - O ServletContext e o escopo de aplicação

As aplicações Web em Java têm 3 escopos possíveis. Ja 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.

17.10 - 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 atrubutos 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.

CAPÍTULO 18

Apêndice - Struts 1

"A qualidade é a quantidade de amanhã"
– Henri Bergson

Ao término desse capítulo, você será capaz de:

- Utilizar o Struts para controlar sua lógica de negócios;
- Criar atalhos para sua camada de visualização;
- Criar e configurar mapeamentos de ações e templates;
- Utilizar form beans para facilitar a leitura de formulários;
- Validar seu formulário de forma simples;
- Controlar os erros de validação do mesmo.

18.1 - Struts 1 e o mercado

No curso, abordamos a versão 2.0 do Struts que é a mais recente. Mas a primeira versão do Struts, da série 1.x foi lançado em 2000 e rapidamente dominou o mercado. Trabalhar com servlets e JSPs puros era tão improdutivo nesta época que as facilidades do Struts o tornaram a ferramenta mais utilizada no mercado Web em Java.

A ideia era aplicar o padrão MVC no desenvolvimento Web provendo um controlador genérico já pronto e com muitas funcionalidades implementadas (conversão, validação, popular parâmetros, etc). Mas, nessa época, o uso extensivo de herança e XMLs imensos ainda não era visto como um problema. Hoje vemos muitas deficiências no Struts 1, principalmente quando comparamos com frameworks mais modernos que evoluíram as ideias originais do Struts. Mas é importante reconhecer o papel histórico e divisor de águas do Struts 1 no ano 2000.

Até hoje, principalmente no Brasil, o Struts 1 ainda é muito utilizado. Não é recomendado criar um novo projeto usando Struts 1, mas muitos projetos antigos usam essa tecnologia. Hoje os frameworks mais comuns de se usar são o JSF, o Struts 2 e o Spring MVC.

Este apêndice apresenta as principais características do Struts 1 caso você precise utilizá-lo em algum projeto.

18.2 - Exercícios: Configurando o Struts

1) Crie um projeto chamado `struts` no Eclipse, seguindo os passos abaixo:

   a) No Eclipse, vá em *File > New > Project*
   b) Selecione *Dynamic Web project* e clique *Next*
   c) Preencha *Project name* com **struts** e clique em *Finish*. Deixe o tomcat 6 marcado como opção

   Isso cria um novo projeto web no WTP preparado com o contexto /**struts** (isso quer dizer que acessaremos via http://localhost:8080/struts/)

2) Baixe a última versão da série 1.x do Struts em:

   http://struts.apache.org

   Para fazer o Struts funcionar, precisamos colocar os JARs que baixamos direto do site deles dentro da pasta **WEB-INF/lib**. Além disso, copie as classes anteriores do projeto JDBC (DAO etc) e coloque os JARs do driver do MySQL e da JSTL etc. Enfim, tudo que fizemos até agora mais as configurações do Struts.

3) Configure o **web.xml** com a servlet controladora do Struts. Repare que isso é muito parecido com nosso framework MVC anterior. Como antes, não vamos precisar ficar configurando mais nada nesse XML.

```
<web-app>
    <servlet>
        <servlet-name>testeDeStruts</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>testeDeStruts</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

   Crie também outro XML, o **struts-config.xml** dentro da pasta **WEB-INF**. Esse arquivo é um arquivo específico do Struts e é lá que colocaremos as configurações do framework. Por enquanto, coloque apenas com o esqueleto:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
      "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
      "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">

<struts-config>
</struts-config>
```

4) Crie um arquivo **index.jsp** na pasta **WebContent** com algum conteúdo de teste.

   Clique da direita no nome do seu projeto, escolha *Run As > Run on Server*. Escolha o servidor que você já tem configurado no seu Eclipse e clique em *Finish*

   O Web Browser do Eclipse irá aparecer. Teste a URL http://localhost:8080/struts/ e você verá sua página de teste.

18.3 - Uma ação Struts

No nosso exemplo anterior de MVC utilizávamos uma interface comum a todas as nossas lógicas de negócio. Com o Struts temos uma classe chamada `Action` que iremos estender para implementar nossas lógicas.

Muitos programadores recomendam como boa prática não colocar a lógica de negócio na `Action`, e sim em uma nova classe que é chamada por ela.

Você deve reescrever o método `execute`, como no exemplo abaixo:

```
package  br.com.caelum.struts.action;
// imports....

public class TesteSimplesAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response)
                                    throws Exception {
            // ...
    }
}
```

Por enquanto, temos quatro parâmetros, dois que já conhecemos, e um retorno. Esses três itens serão explicados em breve, passo a passo para evitar complicações uma vez que é nesse ponto que os programadores sentem uma certa dificuldade inicial com o Struts.

Primeiro, precisamos devolver um objeto do tipo `ActionForward`, que indica para onde o usuário deve ser redirecionado ao término da execução daquela ação. Pode ser para outra ação, ou, normalmente, para um `.jsp`.

Poderíamos retornar algo do tipo:

```
return new ActionForward("/exemplo.jsp");
```

Mas isso não é recomendado. Por quê? Uma vez que colocamos o nome do arquivo jsp na nossa camada de lógica de negócios, estamos criando uma ligação muito forte entre as mesmas. Qualquer alteração no nome do arquivo resulta em um grande esforço para encontrar todos os pontos que se referenciam a tal lugar.

Portanto, essa não será a solução que utilizaremos. Partimos para algo que o Struts facilita desde o início de seu projeto: desejamos retornar “ok"! Lembre-se disso.

É muito importante ressaltar que o Struts pode instanciar apenas uma Action de cada tipo, fazendo com que você ainda tenha de se preocupar com problemas de sincronismo, já que existe a possibilidade de existir mais de uma `Thread` acessando o mesmo objeto `Action` ao mesmo tempo.

O nosso JSP final será algo bem simples para esse exemplo:

```
<html>
    Minha primeira página usando o Struts!
</html>
```

18.4 - Configurando a ação no struts-config.xml

Voltemos ao arquivo `struts-config.xml`: esse arquivo irá mapear as URLs que os usuários acessarem (chamados de *path*) e as classes (chamadas de *type*). Sendo assim, nada mais natural e elegante do que mapearmos o path `/teste` para a classe `TesteSimples`.

Atenção: o nome do path não precisa ser igual ao nome da classe! Isto é um mapeamento!

```
<struts-config>

    <action-mappings>
        <action path="/teste" type="br.com.caelum.struts.action.TesteSimples">
            <forward name="ok" path="/exemplo.jsp"/>
        </action>
    </action-mappings>

</struts-config>
```

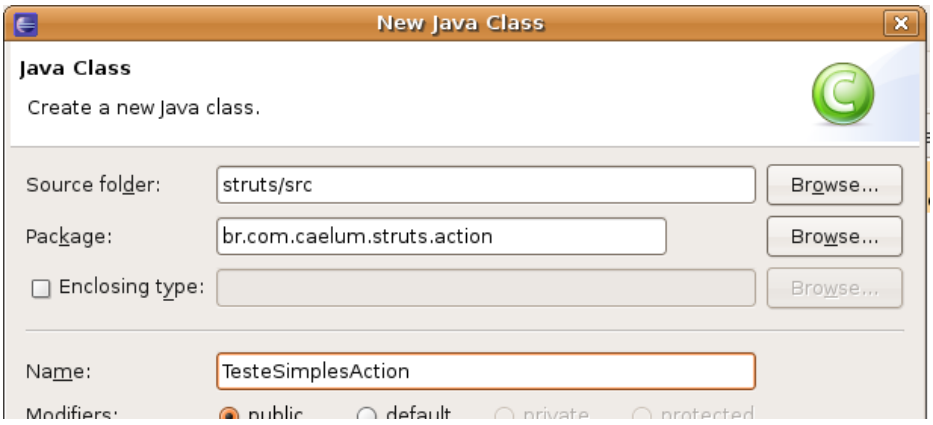
Dentro da tag `action`, colocamos uma tag `forward`. Essa tag define um redirecionamento com um apelido (atributo `name`) e o caminho (`path`). No código, quando fazemos `mapping.findForward("ok")`, o Struts procura um forward com apelido "ok" e devolve o objeto `ActionForward` correspondente (no caso, redirecionando para `exemplo.jsp`). Desta forma, podemos trabalhar com nossas lógicas sem nos atrelarmos muito à camada de visualização.

O parâmetro *path* de action indica qual URL vai acionar essa ação, no caso será o `/teste.do`, pois para acessar o struts precisamos da terminação `.do` no nosso caso.

18.5 - Exercícios: TesteSimplesAction

1) Crie sua primeira ação do Struts.

   a) Crie uma classe chamada `TesteSimplesAction` no pacote `br.com.caelum.struts.action`.



   b) Faça sua classe estender Action (do Struts!).

   c) Utilize CTRL+SHIFT+O para importar a classe.

   d) Escreva o método `execute` e implemente o conteúdo do mesmo, retornando o resultado exemplo

Você pode fazer o Eclipse gerar para você a *assinatura* do método. Basta escrever **execute** dentro da classe e apertar **Ctrl+espaço**. **Cuidado para gerar o método que recebe exatamente os parâmetros como mostrados abaixo**.
Agora implemente o conteúdo do método como abaixo:

```
public class TesteSimplesAction extends Action {

    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        System.out.println("Executando o código da lógica de negócios...");
        return mapping.findForward("ok");
    }
}
```

2) Crie seu arquivo `exemplo.jsp` dentro do diretório WebContent.

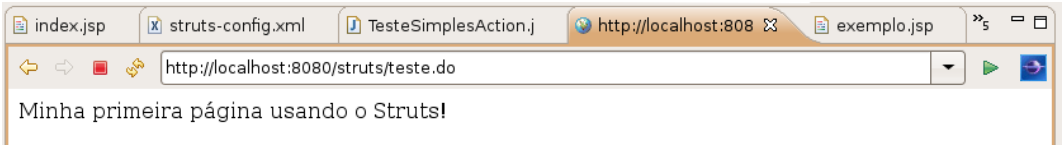
```
<html>
    Minha primeira página usando o Struts!
</html>
```

3) Abra o seu arquivo `struts-config.xml` e configure sua ação dentro da tag `action-mappings`, que vem antes do `message-resources`.

```
<struts-config>
    <action-mappings>
        <action path="/teste" type="br.com.caelum.struts.action.TesteSimplesAction">
            <forward name="ok" path="/exemplo.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

4) Reinicie o Tomcat.

5) Teste a URL http://localhost:8080/struts/teste.do



Reload automático do struts-config.xml

O Struts **não** faz o reload automático do arquivo `struts-config.xml`.
Um truque para fazer isso funcionar (que só é útil durante o desenvolvimento da sua aplicação) é colocar esse arquivo no seu diretório `src`, portanto será jogado no diretório classes, o *classpath*.
Já no seu arquivo `web.xml` configure o struts com um parâmetro de inicialização de servlet para ler o arquivo dentro do diretório classes (`/WEB-INF/classes/struts-config.xml`):

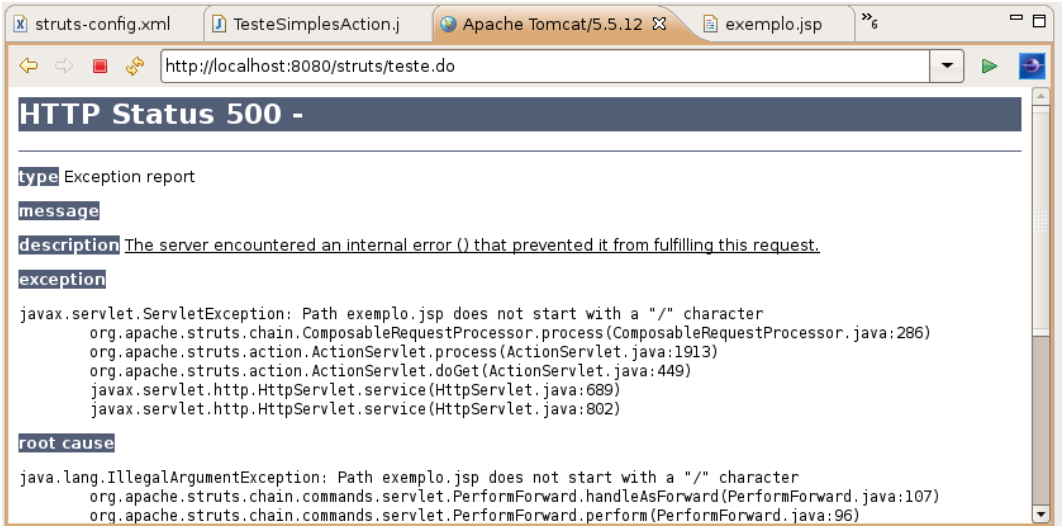
```
 <init-param>

    <param-name>config</param-name>
    <param-value>/WEB-INF/classes/struts-config.xml</param-value>
</init-param>

Agora, toda vez que o arquivo for alterado, o Tomcat percebe uma mudança no classpath
do projeto e reinicia a sua aplicação web.
Cuidado pois essa funcionalidade de reinicialização de contextos pode nem sempre
funcionar como você espera. Um caso simples é iniciar threads separadas e deixá-las
rodando no background, o que acontece?
```

18.6 - Erros comuns

1) O erro mais famoso nos primeiros exemplos de uma Action do Struts é colocar o nome do forward de maneira inválida, por exemplo, em minúsculo no `struts-config.xml` e em maiúsculo na sua classe. Lembre-se, o Java é case-sensitive e assim será a maior parte de suas bibliotecas!

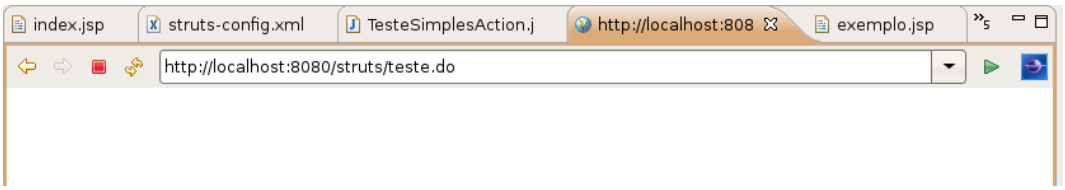
   Como o Struts não encontra um redirecionamento com tal chave, o método `findForward` retorna `null`, resultado: uma tela em branco.

2) Outro erro comum é esquecer de colocar a barra antes do nome do redirecionamento. Todo path de forward deve começar com uma barra. Se você colocar somente `exemplo.jsp` o erro diz claramente que faltou uma barra:



3) É comum errar o nome da classe de sua action, como por exemplo esquecer o .com e digitar `br.caelum.struts.action.TesteSimplesAction`. Nesse caso o Struts não consegue instanciar sua action:

4) A classe Action possui dois métodos execute: um com `HttpServletRequest` e `Response` e um com `ServletRequest` e `Response`. Como estamos usando o protocolo Http, precisamos usar o primeiro método. Caso usemos o segundo, recebemos uma página em branco.



5) Por último, o erro dos esquecidos. Se você não criar o arquivo JSP ou colocar um nome inválido o erro é o já conhecido 404:

18.7 - Pesquisando um banco de dados

Continuando com nossa aplicação criada no capítulo anterior, iremos montar agora um esquema de simulação de acesso a um banco de dados para listar todos os contatos através do MVC e usando Struts, JSP e JDBC.

Repare que já estamos usando três camadas e três bibliotecas diferentes!

18.8 - Criando a ação

Para criar a ação de listagem basta utilizarmos a idéia de criar um novo objeto do tipo DAO e chamar o método de lista:

```
// pesquisa no banco de dados a lista completa
List<Contato> lista = new ContatoDAO().getLista();
```

Mas, espere um pouco, esse é o exemplo que vimos no começo da apostila? Até aqui, sem novidades. A questão que fica é como enviar o conteúdo referenciado pela variável lista para a página JSP que será acessada em breve.

Precisamos de um escopo de variável que sobreviva ao método `execute` e continue durante o forward da requisição até o arquivo JSP. Repare que a frase anterior entrega a solução: o escopo da requisição.

Atrelaremos o valor referenciado pela variável lista para um nome qualquer ligada a requisição do cliente. Esse valor só ficará lá até o término da requisição, tempo suficiente para mostrá-lo no arquivo jsp.

Podemos adicioná-la como atributo no request, para que nossa página JSP possa receber tal objeto. Suponha que desejamos chamar nossa lista de “contatos":

```
request.setAttribute("contatos", lista);
```

E o redirecionamento é simples:

```
return mapping.findForward("lista");
```

Portanto, o código final de nossa ação é:

```
package br.com.caelum.struts.action;
// imports aqui
```

```
public class ListaContatosAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
                    HttpServletRequest request, HttpServletResponse response)
                    throws Exception {
        // pesquisa no banco de dados a lista completa
        List<Contato> lista = new ContatoDAO().getLista();
        request.setAttribute("contatos", lista);

        // ok.... para onde ir agora?
        return mapping.findForward("lista");

    }
}
```

18.9 - O arquivo WebContent/lista.jsp

Para criarmos o JSP de listagem, temos três opções. A primeira seria escrever o código através de scriplets, que já vimos no capítulo de JSP: não é uma boa solução.

A segunda opção é utilizar a biblioteca de tags de lógica do Struts, a `struts-logic`, que funciona e é uma alternativa.

A terceira, é utilizar JSTL. Qual a diferença entre a `struts-logic` e a JSTL core? Acontece que a biblioteca do Struts veio antes da JSTL: a JSTL é a tentativa de padronizar essas taglibs que apareceram pelo mundo inteiro. Sendo assim, todos, inclusive o grupo Apache, estão migrando para a JSTL. Por essa razão, seguiremos no curso apenas com a JSTL, que é a biblioteca padrão.

Como fizemos antes, primeiro devemos declarar a variável, que está sendo lida do `request`. Logo depois iteramos por todos os itens:

```
<!-- for -->
<c:forEach var="contato" items="${contatos}">
    ${contato.id} - ${contato.nome} <br/>
</c:forEach>
```

Portanto, o arquivo final, com cabeçalho e tudo o que faltava, fica sendo:

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

<!-- for -->
<c:forEach var="contato" items="${contatos}">
    ${contato.id} - ${contato.nome} <br/>
</c:forEach>

</html>
```

Nesse momento, você se pergunta: mas o JSP não declarou a variável `contatos`?! Exato, ele não declarou. A expression language buscará o valor de tal chave no `request` (e em outros lugares, que veremos adiante no curso) e utilizá-la para a iteração, ou seja ele não tem ligação direta com o DAO, ele sabe que vem uma variável contatos, mas não sabe de onde.

18.10 - ListaContatos no struts-config.xml

Por fim, vamos alterar o struts-config.xml para configurar nossa ação:

```
<action path="/listaContatos" type="br.com.caelum.struts.action.ListaContatosAction">
    <forward name="lista" path="/lista.jsp"/>
</action>
```

Portanto, para testarmos nossa aplicação, devemos reiniciar o Tomcat e utilizar o link `/listaContatos.do`.

Repare que agora não faz mais sentido acessar o JSP de listagem diretamente pois a variável não existe!

18.11 - Exercício: ListaContatosAction

Vamos criar sua listagem de contatos:

1) Crie sua classe de lógica `ListaContatosAction` no mesmo pacote `br.com.caelum.struts.action`:
   a) Lembre-se de estender a classe `Action`
   b) Implemente o método `execute`:

```
@Override

public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
        throws Exception {
    // pesquisa no banco de dados a lista completa
    List<Contato> lista = new ContatoDAO().getLista();
    request.setAttribute("contatos", lista);

    // ok.... para onde ir agora?
    return mapping.findForward("lista");
}
```

2) Configure o `struts-config.xml`

```
<action path="/listaContatos" type="br.com.caelum.struts.action.ListaContatosAction">
    <forward name="lista" path="/lista.jsp"/>
</action>
```

3) Crie seu JSP de resultado `WebContent/lista.jsp`

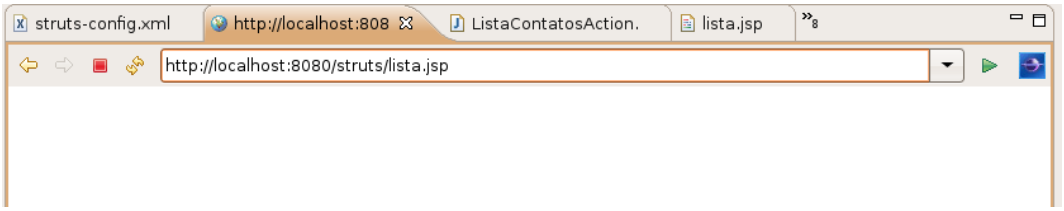
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>

    <!-- for -->
    <c:forEach var="contato" items="${contatos}">
```

```
            ${contato.id} - ${contato.nome} <br/>
        </c:forEach>

    </html>
```

4) Reinicie o Tomcat.

5) Teste a URL http://localhost:8080/struts/listaContatos.do

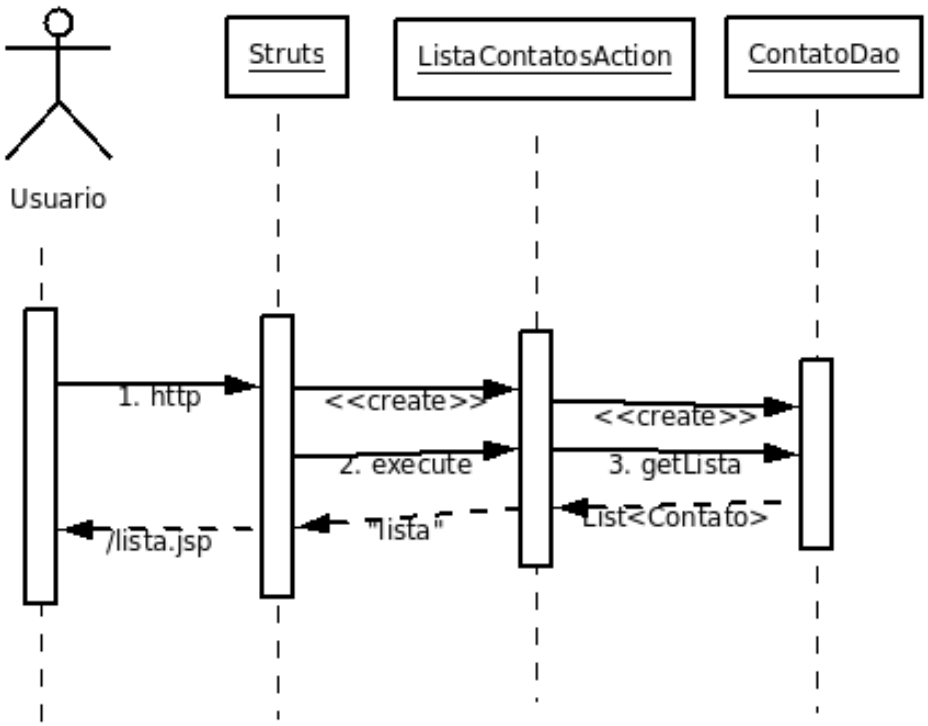
6) O que acontece se acessarmos diretamente o JSP? O que estamos fazendo de errado? Teste a URL http://localhost:8080/struts/lista.jsp



Neste momento, seu arquivo `struts-config.xml` possui duas actions:

```
<struts-config>
    <action-mappings>
        <action path="/teste" type="br.com.caelum.struts.action.TesteSimplesAction">
            <forward name="ok" path="/exemplo.jsp"/>
        </action>

        <action path="/listaContatos"
                type="br.com.caelum.struts.action.ListaContatosAction">
            <forward name="lista" path="/lista.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

O seguinte diagrama descreve o que acontece com o nosso sistema ao requisitar a listagem de contatos:



18.12 - Resultado condicional com o Struts

Como fazer para mostrar a mensagem “Nenhum contato fora encontrado"?

A primeira idéia é a de colocar um if dentro do seu JSP e resolver o problema, certo? Mas isso só trará problemas para o designer, que não sabe tanto de lógica quanto você e pode ser que o editor que ele usa não suporte tais tipos de lógicas...

Então, a melhor saída é verificar, ainda dentro de sua ação, se o banco de dados retornou uma coleção de tamanho zero. E, nesse caso, redirecionar para outra página.

18.13 - Exercícios: listagem vazia

1) Crie um JSP novo chamado `lista-vazia.jsp` na pasta WebContent.

```
<html>
    Você não possui nenhum contato.
</html>
```

2) Alterando somente sua lógica e adicionando um novo forward, quando a lista estiver vazia, a página `lista-vazia.jsp` seja mostrada:

```
 // pesquisa no banco de dados a lista completa

List<Contato> lista = new ContatoDAO().getLista();
request.setAttribute("contatos", lista);

// ok.... para onde ir agora?
if(lista.isEmpty()) {
```

```
        return mapping.findForward("vazia");
    } else {
        return mapping.findForward("lista");
    }
```

3) Altere seu `struts-config.xml` para adicionar um novo forward para a `lista-vazia`:

```
<action path="/listaContatos" type="br.com.caelum.struts.action.ListaContatosAction">
    <forward name="lista" path="/lista.jsp"/>

    <!-- adicionar a linha a seguir -->
    <forward name="vazia" path="/lista-vazia.jsp"/>
</action>
```

4) Teste a sua listagem com o banco de dados vazio. Para simular a lista vazia, você pode, por exemplo, chamar o método `lista.clear()` ou remover todos os seus contatos do banco.

18.14 - Resultado do struts-config.xml

Neste momento, seu arquivo `struts-config.xml` possui duas actions:

```
<struts-config>
    <action-mappings>
        <action path="/teste" type="br.com.caelum.struts.action.TesteSimplesAction">
            <forward name="ok" path="/exemplo.jsp"/>
        </action>

        <action path="/listaContatos"
                type="br.com.caelum.struts.action.ListaContatosAction">
            <forward name="lista" path="/lista.jsp"/>
            <forward name="vazia" path="/lista-vazia.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

18.15 - Novos contatos

Agora, já estamos prontos para criar a lógica de negócios e a camada de visualização para permitir adicionar novos clientes e, consequentemente, listá-los.

Como de costume, seguiremos os passos:

1) Criar a lógica de negócios;

2) Criar o JSP de visualização;

3) Criar o mapeamento da lógica para a visualização;

E depois os passos opcionais:

4) Criar a validação do formulário na lógica de negócios

5) Criar o controle de erro na visualização

18.16 - Formulário

Nunca é elegante trabalhar com o método `getParameter` do `request`, já que é muito melhor trabalhar com classes que nós mesmos escrevemos. Assim, vamos imaginar um cenário simples: desejamos adicionar o nome, email e descrição do cliente.

O Struts possui uma classe chamada `ActionForm` que ao ser estendida permite ler os parâmetros do `request` sem nos preocuparmos com o mesmo!

Sendo assim, no Struts, para cada formulário HTML que existe no nosso site, criamos uma classe em Java para representar os campos do mesmo.

No nosso caso, precisamos dos campos `nome`, `email` e `endereco`, mas opa, isso é um `Contato`! Resultado:

```
package  br.com.caelum.struts.form;

import org.apache.struts.action.*;

public class ContatoForm extends ActionForm {
    private Contato contato = new Contato();
    public Contato getContato() {
        return this.contato;
    }
}
```

Atenção: o formulário HTML deverá ter os campos com o mesmo nome que as variáveis membro do seu formulário!

Existe uma opção avançada de fazer o formulário através de xml, não deixa de ser bastante código e ainda com a desvantagem de não mostrar erros de compilação.

18.17 - Mapeando o formulário no arquivo struts-config.xml

Assim como a action, devemos configurar nosso form no arquivo `struts-config.xml`. Para isso, usamos a tag chamada `form-bean`.

Atributos de uma tag form-bean:

name: um nome qualquer que queremos dar a um formulário; **type:** a classe que representa esse formulário.

Atenção: tal tag vem antes das definições dos `action-mappings`! Todos os formulários devem ser definidos dentro de uma única tag `form-beans`.

```
<form-beans>
    <form-bean name="ContatoForm" type="br.com.caelum.struts.form.ContatoForm"/>
</form-beans>
```

18.18 - Exercício: ContatoForm

1) O primeiro passo é criar o formulário como classe:

   a) Crie a classe `ContatoForm` no pacote `br.com.caelum.struts.form`.

   b) Faça com que seu formulário estenda a classe `ActionForm` do Struts.

```
public class ContatoForm extends ActionForm{


}
```

   c) Coloque uma variável do tipo `Contato` no formulário, chame-a de `contato` e instancie um `Contato`:

```
public class ContatoForm extends ActionForm{

        private Contato contato = new Contato();
    }
```

   d) Vá no menu Source, *Generate Getters and Setters* e escolha o método `getContato`. (ou digite Ctrl+3 e escreva *ggas*).

```
public class ContatoForm extends ActionForm{


    private Contato contato = new Contato();

    public Contato getContato() {
        return contato;
    }
}
```

2) Agora, vamos mapear esse formulário no `struts-config.xml`. Lembre-se que a tag `form-beans` deve vir **antes** da tag `action-mappings`.

```
<form-beans>
    <form-bean name="ContatoForm" type="br.com.caelum.struts.form.ContatoForm"/>
</form-beans>
```

18.19 - Erro comum

O erro mais comum com o struts está no arquivo `struts-config.xml`. Ao configurar seu primeiro `form-bean`, o aluno deve prestar muita atenção que a tag `form-beans` deve vir antes da tag `action-mappings`.

18.20 - Lógica de Negócios

Podemos escrever um código bem simples que adiciona um novo contato (recebido através de um formulário) para o banco de dados:

Criamos um contato, recuperamos os valores do formulário e adicionamos este cliente ao banco de dados:

```
package br.com.caelum.struts.action;

// série de imports aqui

public class AdicionaContatoAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
                    HttpServletRequest request, HttpServletResponse response)
                throws Exception {
        // log
        System.out.println("Tentando criar um novo contato...");

        // formulário de cliente
        ContatoForm formulario = (ContatoForm) form;
        // acessa o bean
        Contato contato = formulario.getContato();

        // adiciona ao banco de dados
        ContatoDAO dao = new ContatoDAO();
        dao.adiciona(contato);

        // ok.... visualização
        return mapping.findForward("ok");
    }
}
```

18.21 - Exercício: AdicionaContatoAction

1) Vamos criar a classe `AdicionaContatoAction`:

   a) Crie a classe `AdicionaContatoAction` em `br.com.caelum.struts.action`

   b) Faça com que sua classe estenda `Action` do Struts.

   c) Digite execute, CTRL+ESPAÇO e dê enter: implemente o método `execute` lembrando de verificar os nomes de seus argumentos:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
                    HttpServletRequest request, HttpServletResponse response)
                throws Exception {
    // log
    System.out.println("Tentando criar um novo contato...");

    // formulário de cliente
    ContatoForm formulario = (ContatoForm) form;
    // acessa o bean
    Contato contato = formulario.getContato();

    // adiciona ao banco de dados
```

```
ContatoDAO dao = new ContatoDAO();
dao.adiciona(contato);

// ok.... visualização
return mapping.findForward("ok");
}
```

2) Vamos configurar sua ação:

a) Define sua ação `/novoContato` no arquivo `struts-config.xml` apontando para a classe `AdicionaContatoAction`.

```
<action path="/novoContato" name="ContatoForm"
        type="br.com.caelum.struts.action.AdicionaContatoAction">
</action>
```

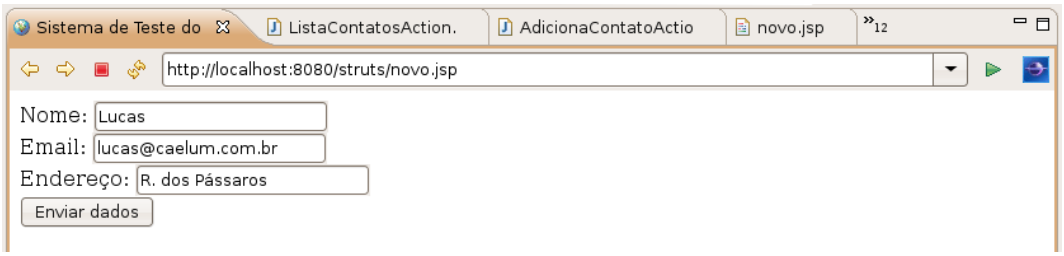
b) Em caso de sucesso (ok), redirecione para o `/listaContatos.do` (isto mesmo, estamos encadeando duas ações).

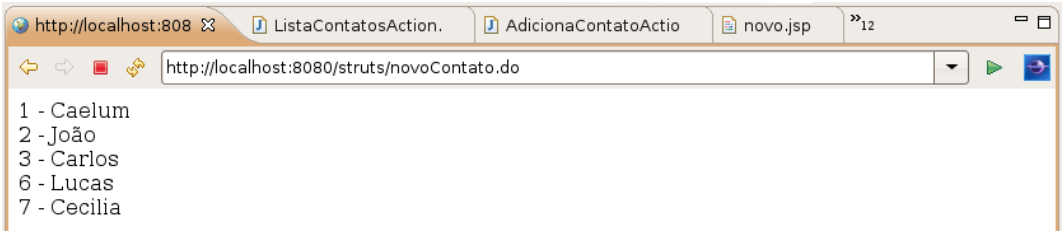
```
<action path="/novoContato" name="ContatoForm"
    type="br.com.caelum.struts.action.AdicionaContatoAction">
    <forward name="ok" path="/listaContatos.do"/>
</action>
```

3) Crie seu arquivo `novo.jsp` na pasta WebContent:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>

<html:html>
    <head>
        <title>Sistema de Teste do Struts</title>
    </head>

    <html:errors/>

    <html:form action="/novoContato" focus="contato.nome">
        Nome:
            <html:text  property="contato.nome"/>
            <br/>

        Email:
            <html:text  property="contato.email"/>
            <br/>

        Endereço:
            <html:text  property="contato.endereco"/>
            <br/>

            <html:submit>Enviar dados</html:submit>
        <br/>
    </html:form>
</html:html>
```

4) Teste a url http://localhost:8080/struts/novo.jsp





5) Agora, tente criar um contato com nome vazio, funciona?

18.22 - Erros comuns

A seguir veja os erros mais comuns no exercício anterior:

1) Esquecer de fazer seu `ContatoForm` estender `ActionForm`: como `ContatoForm` e `ActionForm` não possuem conexão, o compilador (no caso, o Eclipse), reclama do casting que está sendo feito dentro da sua classe `Action`, afinal nenhum `ActionForm` é um `ContatoForm`. Solução: estenda `ActionForm`.

2) Esquecer de colocar o atributo `name="ContatoForm`

   na sua tag action dentro do `struts-config.xml`. Como você não notificou o Struts que sua `Action` precisa de um `form`, ele passa `null` para seu método e, quando chega na linha que tenta chamar o método `getContato`, acontece um `NullPointerException`. Percebeu que o `NullPointerException` foi um erro do programador? Eles normalmente são descuidos do programador, portanto sempre preste atenção naquilo que você faz e configura! Solução: vá no seu arquivo `struts-config.xml` e coloque o atributo `name="ContatoForm`

   na sua tag action.

18.23 - Arquivo de mensagens

O struts possui um sistema bem simples de internacionalização.

Esse é o processo onde centralizamos todas as mensagens do sistema em um (ou mais) arquivos de configuração que são baseados na antiga idéia de chave-valor, um dicionário de mensagens. Por exemplo:

```
menu.nome = Menu principal
menu.arquivo = Arquivo
menu.editar = Editar
menu.sair = Sair
site.titulo = Sistema de teste do Struts
```

Para configurar o struts e usar um tipo de arquivo como esse, começamos indicando qual o arquivo de configuração que usaremos. O nome mais comum é `MessageResources.properties`. Esse arquivo deve ser criado no nosso diretório `src`.

Para o Struts ler tal arquivo basta configurá-lo no `struts-config.xml`, localizado no diretório `WEB-INF`:

```
<struts-config>
    <form-beans>    <!-- beans aqui -->    </form-beans>
    <action-mappings>
            <!-- actions -->
        </action-mappings>

    <!-- Arquivo de Mensagens -->
    <message-resources parameter="MessageResources" />
</struts-config>
```

Para utilizar tal arquivo é bem simples, basta no nosso JSP usar uma taglib do struts chamada `bean`:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
```

Dada uma chave (`menu.nome` por exemplo), podemos chamar a tag `message` que é capaz de mostrar a mensagem “Menu principal":

```
<bean:message key="site.titulo" />
```

Portanto, o exemplo a seguir mostra um menu completo usando essa taglib:

```
<html>
    <head>
        <title><bean:message key="site.titulo" /></title>
    </head>

    <body>
        <bean:message key="menu.nome" /><br/>
        <bean:message key="menu.arquivo" /><br/>
        <bean:message key="menu.editar" /><br/>
        <bean:message key="menu.sair" /><br/>
    </body>
</html>
```

18.24 - Exercícios: Mensagens

1) Crie um arquivo chamado `testa-mensagens.jsp`.

   a) Inclua a taglib `bean`:

      ```
      <%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
      ```

b) Inclua as mensagens:

```
<html>
    <head>
        <title><bean:message key="site.titulo" /></title>
    </head>

    <body>
        <bean:message key="menu.nome" /><br/>
        <bean:message key="menu.arquivo" /><br/>
        <bean:message key="menu.editar" /><br/>
        <bean:message key="menu.sair" /><br/>
    </body>
</html>
```

2) Abra o arquivo chamado `MessageResources.properties` no seu diretório `src` e adicione:

```
# comentario de um arquivo .properties

menu.nome = Nome do menu
menu.arquivo = Escolher Arquivo
menu.editar = Editar Arquivo
menu.sair = Sair da aplicação
site.titulo = Sistema de teste do Struts
```

3) Adicione a configuração a seguir no final arquivo `struts-config.xml` para utilizar tal arquivo de mensagens:

```
<struts-config>

    <form-beans>
        <!-- beans aqui -->
    </form-beans>

    <action-mappings>
            <!-- actions aqui-->
    </action-mappings>

    <!-- Arquivo de Mensagens -->
    <message-resources parameter="MessageResources" />

</struts-config>
```

4) Reinicie o Tomcat. Será sempre necessário fazer isso ao alterar seu arquivo `struts-config.xml`.

5) Teste a URL http://localhost:8080/struts/testa-mensagens.jsp

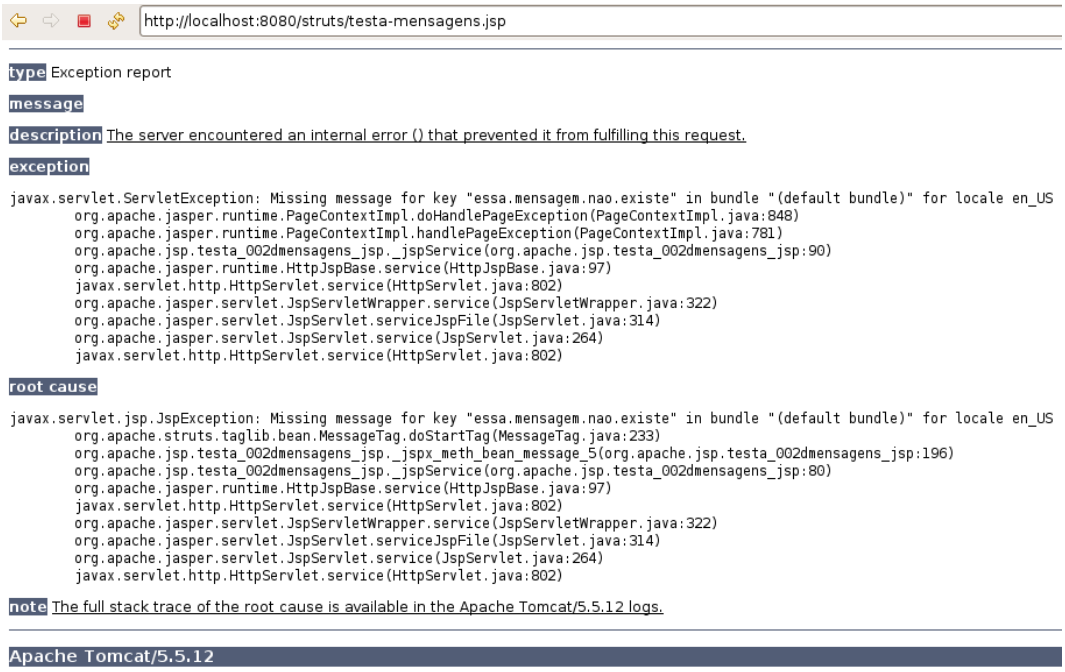
18.25 - Erros comuns

Infelizmente a maior parte dos erros possíveis no exercício anterior trazem a mesma tela de erro: código 500, incapaz de achar o valor de uma mensagem: *Cannot find message resources under key org.apache.struts.action.MESSAGE*.

1) Na sua página HTML, digitar o valor de uma mensagem de maneira errada. O Struts encontra seu arquivo mas não encontra a mensagem. Verifique o exercício 1.

2) Esquecer de alterar o arquivo `MessageResources.properties` e colocar a mensagem nele. O Struts encontra seu arquivo, mas não encontra a mensagem. Verifique o exercício 2.

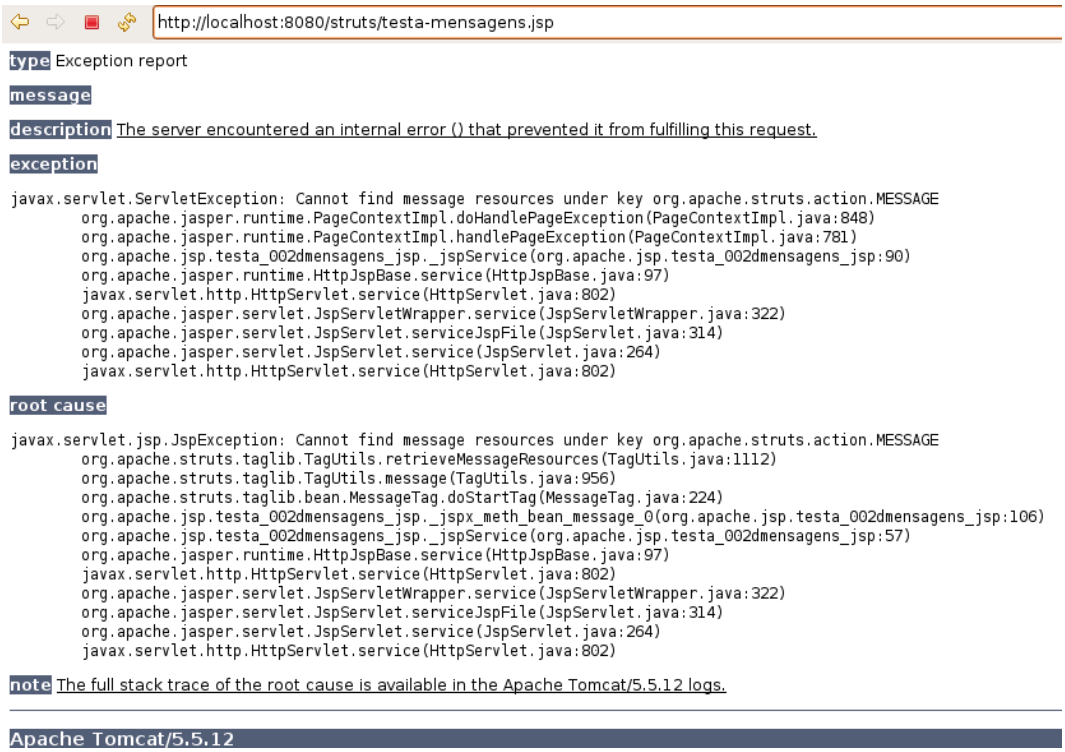
   Para os dois erros acima a mensagem é a mesma:



3) Esquecer de alterar o arquivo `struts-config.xml` para configurar o arquivo `MessageResources.properties`. Verifique o exercício 3.

4) Esquecer de reiniciar o servidor. Verifique o exercício 4.

   Tela dos erros 3 e 4:



18.26 - Validando os campos

Para facilitar nosso trabalho, podemos agora implementar o método de validação que vem junto com o Struts.

Escreveremos, através do formulário, um método que retorna uma lista de erros encontrados. Para tanto, vamos verificar se as strings de nome, email ou endereço foram preenchidas ou não.

O método de validação do formulário é o método `validate`. Caso haja algum erro de validação, devemos adicionar os erros ao objeto `ActionErrors`. Por exemplo:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {

    ActionErrors erros = new ActionErrors();

    if (contato.getNome() == null || contato.getNome().equals("")) {
        erros.add("nome", new ActionMessage("erro.campoNome"));
    }

    return erros;
}
```

Nesse caso, usaremos a palavra `erro.campoNome` como chave para a mensagem de erro! Isso mesmo, fica muito mais fácil controlar o que vai ser apresentado ao seu usuário como mensagem de erro pois vamos configurá-lo no arquivo `MessageResources.properties`.

Acrescentando as verificações dos outros campos, temos o código final do método `validate`:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {

    ActionErrors erros = new ActionErrors();

    // verifica o nome
    if (contato.getNome() == null || contato.getNome().equals("")) {
        erros.add("nome", new ActionMessage("erro.campoNome"));
    }

    // verifica o email
    if (contato.getEmail() == null || contato.getEmail().equals("")) {
        erros.add("email", new ActionMessage("erro.campoEmail"));
    }

    // verifica o endereco
    if (contato.getEndereco() == null || contato.getEndereco().equals("")) {
        erros.add("endereco", new ActionMessage("erro.campoEndereco"));
    }

    return erros;
}
```

Agora, basta alterar nossa configuração do `struts-config.xml` e adicionar o atributo chamado `input`.

```
<action path="/novoContato" name="ContatoForm" input="/novo.jsp"
        type="br.com.caelum.struts.action.AdicionaContatoAction">
    <forward name="ok" path="/listaContatos.do"/>
</action>
```

18.27 - Exercício: validação

1) Abra a sua classe `ContatoForm`.

   a) Crie o método `validate`:

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {


    ActionErrors erros = new ActionErrors();

    // verifica o nome
    if (contato.getNome() == null || contato.getNome().equals("")) {
        erros.add("nome", new ActionMessage("erro.campoNome"));
    }

    // verifica o email
    if (contato.getEmail() == null || contato.getEmail().equals("")) {
        erros.add("email", new ActionMessage("erro.campoEmail"));
    }

    // verifica o endereco
    if (contato.getEndereco() == null || contato.getEndereco().equals("")) {
        erros.add("endereco", new ActionMessage("erro.campoEndereco"));
    }
```

```
        return erros;
    }
```

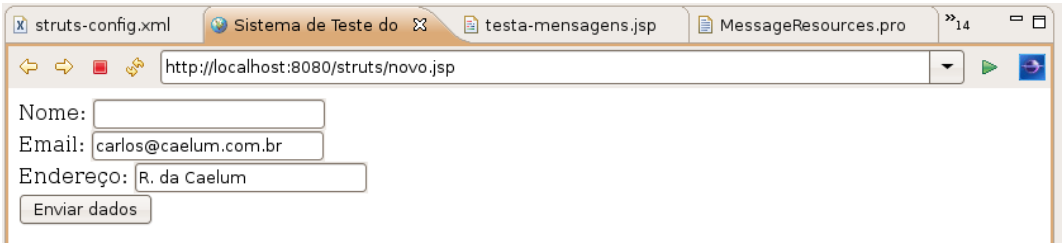
2) **Altere** o mapeamento da action AdicionaContatoAction no struts-config.xml, **não adicione**!

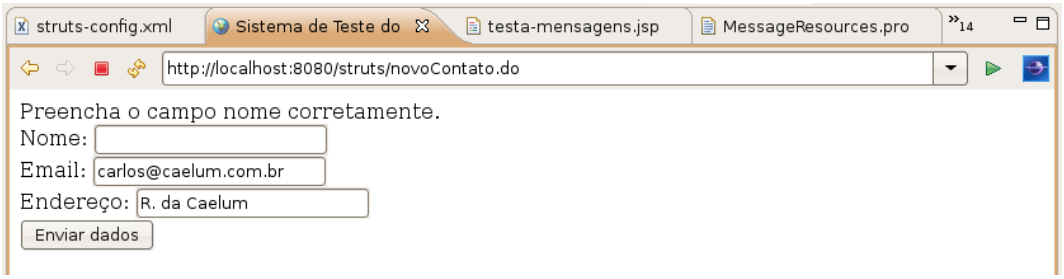
```
<action path="/novoContato" name="ContatoForm" input="/novo.jsp"
        type="br.com.caelum.struts.action.AdicionaContatoAction">
    <forward name="ok" path="/listaContatos.do"/>
</action>
```

3) Coloque as mensagens de erro no seu arquivo de `resources`.

```
 erro.campoNome = Preencha o campo nome corretamente.

erro.campoEmail = Preencha o campo email corretamente.
erro.campoEndereco = Preencha o campo endereco corretamente.
```

4) Tente criar um novo contato com nome vazio: http://localhost:8080/struts/novo.jsp





18.28 - Erros comuns

1) Esquecer de colocar o atributo `input="/novo.jsp`

   . O Struts fica sem saber para onde ir no caso da validação dar errado, então ele reclama que você não colocou o atributo `input`.

   Solução: coloque o atributo `input` na sua tag `action`.

2) Em vez de alterar o mapeamento do seu xml, copiar novamente o mapeamento inteiro, isto é, possuir duas actions com o path `/novoContato`. O Struts fica perdido e não funciona.

18.29 - Exercícios opcionais

1) Abra o seu arquivo `MessageResources.properties`.

   a) Adicione as seguintes linhas:

```
errors.header=<UL>

errors.footer=</UL>
errors.prefix=<LI>
errors.suffix=</LI>
```

   b) Efetue a inserção de um contato que gere diversas mensagens de erro.
   O item `header` aparece antes da lista de erros enquanto o item `footer` aparece após a lista. Já os itens `prefix` e `suffix` são prefixos e sufixos a serem adicionados a toda mensagem de erro.

2) Utilize a tag `html:errors` com o atributo `properties` para mostrar somente as mensagens de erro de determinado campo. Para mostrar as mensagens relativas ao campo nome utilize, por exemplo:

```
<html:errors property="nome"/>
```

18.30 - Limpando o formulário

Adicione um contato no banco de dados, acesse um outro site qualquer e volte ao seu `novo.jsp`. O que acontece? O formulário continua preenchido?

Isso ocorre por que o Struts recicla objetos do tipo `ActionForm` entre diferentes `requests`, fazendo com que seu objeto permaneça sujo. Por padrão, os nossos action forms estão no chamado escopo de sessão (session), facilitando, por exemplo, a criação de longos wizards.

Mas, na maioria dos casos, queremos que o formulário sirva apenas para o request atual. Para isso, usamos um atributo opcional na tag `action` quando utilizando um formulário: `scope`. Esse atributo é utilizado para deixar os dados do formulário no request ou na sessão.

Utilizamos o escopo de sessão (padrão) quando desejamos que os dados se mantenham atrelados aquele cliente por mais de um request, enquanto que o escopo de `request` atrela os dados somente até o término da requisição.

Portanto, você pode colocar no seu action:

```
scope="session"
```

ou:

```
scope="request"
```

É comum utilizar o escopo de sessão para manter os dados de um formulário através de diversas requisições.

18.31 - Exercícios: scope

1) **Altere** seu `struts-config.xml` para utilizar o escopo do formulário como request.

   Basta **acrescentar** o atributo `scope="request`

   na action do novo contato:

```
<action path="/novoContato" name="ContatoForm" input="/novo.jsp"
        type="br.com.caelum.struts.action.AdicionaContatoAction" scope="request">
    <forward name="ok" path="/listaContatos.do"/>
</action>
```

18.32 - Exercícios opcionais

1) Crie um formulário chamado `RemoveContatoForm` e mapeie-o no `struts-config.xml`.
2) Crie uma ação chamada `RemoveContatoAction`
   a) Ela recebe um formulário do tipo `RemoveContatoForm`;
   b) Ela remove do banco (usando o `ContatoDAO`) o contato com `id` igual ao do contato do formulário. Algo como:

```
Contato contato = ((RemoveContatoForm) form).getContato();

new ContatoDAO().remove(contato);
```

   c) Mapeie uma ação no `struts-config.xml` chamada `removeContato` para sua classe `RemoveContatoAction`;
   d) Redirecione para `/listaContatos.do` após remover um contato;
   e) Na sua lista, altere o código para incluir um link para remoção:

```
<c:forEach var="contato" items="${contatos}">
    ${contato.id} - ${contato.nome}
    (<a href="removeContato.do?contato.id=${contato.id}">remover</a>)<br/>
</c:forEach>
```

   f) Teste a sua listagem e depois remova algum contato;

18.33 - O mesmo formulário para duas ações

Nesse momento do aprendizado é bem comum que um aluno pense em reutilizar formulários para duas ações diferentes. Será que vale a pena? Será que as duas ações tem exatamente a mesma validação? Será que elas sempre vão ter a mesma validação?

Como ou: (1) duas ações não costumam ter a mesma validação, ou: (2) não podemos prever o futuro e saber se alterações no negócio trarão validações diferentes para essas duas ações é considerado boa prática criar formulários diferentes para ações diferentes, utilizando sempre o bom senso.

18.34 - Exercícios opcionais

1) Vamos internacionalizar nosso sistema:

a) Crie uma ação chamada `MudaIdiomaAction`;

b) Ela chama o método `setLocale` que altera o locale do cliente e retorna o forward de ok:

```
String idioma = request.getParameter("idioma");

Locale locale = new Locale(idioma);

System.out.println("Mudando o locale para " + locale);
setLocale(request, locale);

return mapping.findForward("ok");
```

c) Mapeie essa action para o path `/mudaIdioma`;

```
<action path="/mudaIdioma" type="br.com.caelum.struts.action.MudaIdiomaAction">
    <forward name="ok" path="/testa-mensagens.jsp"/>
</action>
```

d) Altere seu arquivo `testa-mensagens.jsp` para adicionar dois links para a ação de alterar a língua:

```
<a href="mudaIdioma.do?idioma=en">EN</a>
<a href="mudaIdioma.do?idioma=pt">PT</a><br/>
```

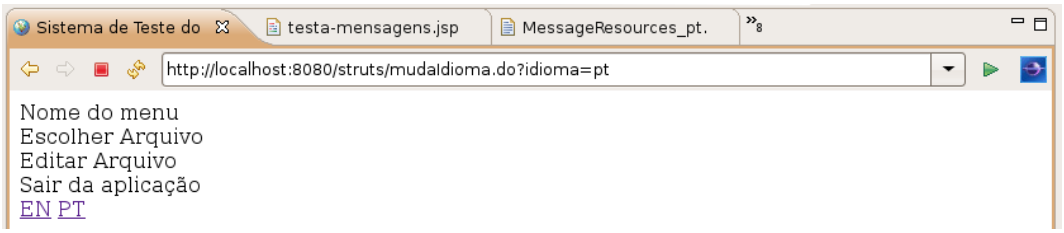
e) Crie o arquivo `MessageResources_en.properties` no seu diretório `src`;

```
site.titulo = Struts Test

pergunta.usuario = What is your username?
pergunta.senha = What is your password?
pergunta.enviar = Send data

# comentario de um arquivo .properties
menu.nome = Menu name
menu.arquivo = Choose file
menu.editar = Edit file
menu.sair = Quit
```

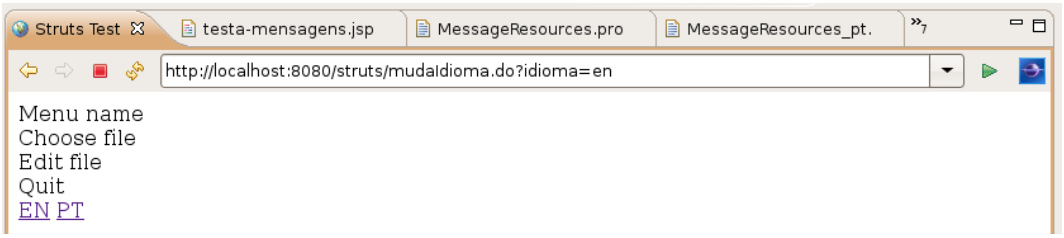
f) Reinicie o Tomcat;

g) Teste a URL http://localhost:8080/struts/testa-mensagens.jsp:
O site aparece por padrão na língua que o seu browser pediu, que é a língua configurada no seu sistema operacional.

h) Escolha o link para português:



i) Escolha o link para inglês:



2) Vamos mostrar agora os detalhes de um contato.

a) Crie a classe `MostraContatoForm` similar ao `ContatoForm`;

b) Crie uma ação chamada `MostraContatoAction`;

c) Ela chama o método `procura`:

```
MostraContatoForm formulario = (MostraContatoForm) form;

Contato contato = formulario.getContato();
Contato encontrado = new ContatoDAO().procura(contato.getId());
request.setAttribute("contato", encontrado);
```

d) Mapeie uma ação no `struts-config.xml` chamada `mostraContato` para sua classe `MostraContatoAction`

e) Redirecione para `/mostraContato.jsp` após mostrar um contato. O código dele mostra os dados do contato (sem formulário).

f) Na sua `lista.jsp`, altere o código para incluir um link para `mostraContato.do`:

```
<c:forEach var="contato" items="${contatos}">
    ${contato.id} - ${contato.nome}
    (<a href="removeContato.do?contato.id=${contato.id}">remover</a>)
    (<a href="mostraContato.do?contato.id=${contato.id}">mostrar</a>)<br/>
</c:forEach>
```

3) Vamos terminar a parte de alterar o contato.

a) Altere a pagina `mostraContato.jsp` para mostrar um formulario acessando `/alteraContato.do`. Nas tags `html:text` utilize o campo `value="${...}`
para colocar o valor inicial nos mesmos;

b) Crie um form chamado `AlteraContatoForm`;

c) Crie uma ação chamada `AlteraContatoAction`;

d) Ela chama o método `altera`:

```
AlteraContatoForm formulario = (AlteraContatoForm) form;

Contato contato = formulario.getContato();

new ContatoDAO().altera(contato);
```

e) Mapeie uma ação no `struts.config.xml` chamada `alteraContato` para sua classe `AlteraContatoAction`;

f) Redirecione para `/listaContatos.do` após alterar um contato;

18.35 - Para saber mais

1) O **Struts Tiles** ajuda você a componentizar os "pedaços" das suas páginas. Dando nome para os diversos componentes comuns as páginas, você pode incluí-los dinamicamente em qualquer JSP.

2) O **Struts Validator** pode ser configurado para que os `form beans` sejam verificados antes de suas ações serem executadas. Ele é, de longe, o plugin mais famoso e poderoso do Struts.

3) O projeto **Velocity Tools** faz a ponte entre o Velocity e o Struts, entre outros.

4) Você pode utilizar a tag `<html:error property="nome"/>`, por exemplo, para mostrar somente os erros relacionados ao campo `nome`.

AlwaysLinkToActions

Um dos patterns mais simples e famosos que o Struts construiu é o *Always Link To Actions*. Você sempre deve se referenciar as ações do Struts e nunca as suas páginas JSP diretamente. Se você já esconde suas páginas JSP no diretório WEB-INF, está se obrigando a utilizar tal procedimento. Qual a vantagem?
Se, em algum dia, sua página JSP precisa executar uma lógica antes de ser chamada ou se ela deve ser renomeada, basta alterar o arquivo `struts-config.xml`, caso contrário você deveria procurar todos os links em sua aplicação!

Forwards de redirecionamento no cliente

Podemos efetuar o redirecionamento no cliente em vez de fazê-lo no servidor. Utilizando tal recurso, o cliente fica sabendo do redirecionamento e, ao clicar em Refresh (Atualizar) ou pressionar F5 no seu navegador, ele efetuará a requisição do redirecionamento e não da página original.

```
<forward name="ok" redirect="true" path="/listaContatos.do" />
No exemplo acima, o redirecionamento após a adição de um contato ao banco será feito
para a listagem, portanto ao pressionar F5 o cliente pede a listagem novamente e não a
adição.
```

A ação padrão

Para marcar uma ação como a padrão, isto é, aquela que deve ser executada caso nenhuma das outras for a correta, basta adicionar um atributo chamado unknown. Somente uma ação pode ter tal atributo com valor `true`.

```
<action path="/seu path aqui" type="sua classe aqui" unknown="true" />
```

Ações só de forward

Às vezes, é interessante criar um apelido para uma página JSP. Para isso, uma das alternativas é criar uma ação que em vez de possuir um `type`, possui um atributo chamado `forward`:

```
<action path="/apelido" forward="/minha_pagina.jsp" />

No exemplo acima, comum no mercado, a URL que termina com /apelido.do será
redirecionada para a página JSP dentro do diretório WEB-INF/jsp.
```

Global Forwards

O Struts permite configurar no `struts-config.xml` uma lista de forwards globais que podem ser utilizados por todas as ações. Para isso, basta adicionar a tag `global-forwards` antes dos `action-mappings`.

```
<global-forwards>
    <forward name="exception" path="/error.jsp"/>
</global-forwards>
```

Se você quiser controlar os erros através desse forward, basta usar algo similar ao código a seguir:

```
 catch (Exception e) {

    return mapping.findForward("exception");
}
```

Índice Remissivo