Caelum

"Mata o tempo e matas a tua carreira"
Bryan Forbes -

Sobre a empresa

A Caelum atua no mercado desde 2002, desenvolvendo sistemas e prestando consultoria em diversas áreas, à luz sempre da plataforma Java. Foi fundada por profissionais que se encontraram no Brasil depois de uma experiência na Alemanha e Itália, desenvolvendo sistemas de grande porte com integração aos mais variados ERPs. Seus profissionais publicaram já diversos artigos nas revistas brasileiras de Java, assim como artigos em eventos acadêmicos, e são presença constante nos eventos da tecnologia.

Em 2004 a Caelum criou uma gama de cursos que rapidamente ganharam grande reconhecimento no mercado. Os cursos foram elaborados por ex-instrutores da Sun que queriam trazer mais dinamismo e aplicar as ferramentas e bibliotecas utilizadas no mercado, tais como Eclipse, Hibernate, Struts, e outras tecnologias open source que não são abordadas pela Sun. O material utilizado foi inicialmente desenvolvido enquanto eram ministrados os cursos de verão de java da Universidade de São Paulo em janeiro de 2004 pelos instrutores da Caelum.

Em 2006 a empresa foca seus projetos em três grandes áreas: sistemas de gerenciamento de conteúdo para portais, soluções de integração financeira e treinamento com intuito de formação.

Sobre a apostila

Esta é a apostila da Caelum que tem como intuito ensinar Java de uma maneira elegante, mostrando apenas o que é necessário no momento correto e 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, e que ele possa ser de grande valia para auto didatas e estudantes. Todos os comentários, críticas e sugestões serão muito bem vindos.

Atenção: Você pode verificar a data de última atualização da apostila no fim do índice. Nunca imprima a apostila que você receber de um amigo ou pegar por email, pois atualizamos constantemente esse material, quase que mensalmente. Vá até o nosso site e faça o download da última versão!

www.caelum.com.br

Índice

capítulo 1

Como aprender Java

"Homens sábios fazem provérbios, tolos os repetem"
Samuel Palmer -

Como o material está organizado e dicas de como estudar em casa.

1.1 - O que é realmente importante?

Muitos livros, ao passar os capítulos, mencionam todos os detalhes da linguagem juntamente com seus princípios básicos. Isso acaba criando muita confusão, em especial pois o estudante não consegue distinguir exatamente o que é importante aprender e reter naquele momento daquilo que será necessário mais tempo e principalmente experiência para dominar.

Se uma classe abstrata deve ou não ter ao menos um método abstrato, se o if só aceitar argumentos booleanos e todos os detalhes de classes internas realmente não devem ser preocupações para aquele que possui como objetivo primário aprender Java. Esse tipo de informação será adquirida com o tempo, e não é necessário até um segundo momento.

Neste curso separamos essas informações em quadros especiais, já que são informações extras. Ou então apenas citamos num exercício e deixamos para o leitor procurar informações se for de seu interesse.

Algumas informações não são mostradas e podem ser adquiridas em tutoriais ou guias de referência, são detalhes que para um programador experiente em Java pode ser importante, mas não para quem está começando.

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 aqueles que fazem os cursos intensivos.

O curso

Para aqueles que estão fazendo o curso Java e Orientação a Objetos, é recomendado estudar em casa aquilo que foi visto durante a aula, tentando resolver os exercícios que não foram feitos e os desafios que estão lá para envolver mais o leitor no mundo de Java.

Convenções de Código

Para mais informações sobre as convenções de código-fonte Java, acesse:

http://java.sun.com/docs/codeconv/

1.2 - Sobre os exercícios

Os exercícios do curso variam entre práticos até pesquisas na Internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos para incitar a curiosidade do aprendiz na tecnologia.

Existem também, em determinados capítulos, uma série de desafios. Eles focam mais no problema computacional que na linguagem, porém são uma excelente forma de treinar a sintaxe e principalmente familiarizar o aluno com a biblioteca padrão Java, além de proporcionar um ganho na velocidade de desenvolvimento.

1.3 - Tirando dúvidas

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do site do GUJ (http://www.guj.com.br/), onde sua dúvida será respondida prontamente.

Se você já participa de um grupo de usuários java ou alguma lista de discussão, pode tirar suas dúvidas nos dois lugares.

Fora isso, sinta-se a vontade de entrar em contato conosco para tirar todas as suas dúvidas durante o curso.

1.4 - Sobre os autores

Guilherme Silveira (guilherme.silveira@caelum.com.br) é programador e web developer certificado pela Sun, trabalhando com Java desde 2000 como especialista e instrutor. Programou e arquiteturou projetos na Alemanha durante 2 anos. Cofundador do GUJ, escreve para a revista Mundo Java, estuda Matemática Aplicada na USP e é instrutor e consultor na Caelum. Um dos comitters do Codehaus XStream.

Paulo Silveira (paulo.silveira@caelum.com.br) é programador e desenvolvedor certificado Java. Possui grande experiência em desenvolvimento web, trabalhando em projetos na Alemanha e em diversas consultorias no Brasil. Foi instrutor Java pela Sun, é cofundador do GUJ e formado em ciência da computação pela USP, onde realiza seu mestrado. É um dos editores técnicos da revista Mundo Java.

Sérgio Lopes (sergio.lopes@caelum.com.br) Bacharelando em Ciência da Computação na USP e desenvolvedor Java desde 2002. É programador certificado Java pela Sun, moderador do GUJ e colaborador da revista Mundo Java. Trabalha com Java para Web e dispositivos móveis, além de ministrar treinamentos na Caelum.

Inúmeras modificações e sugestões foram realizadas por outros consultores e instrutores da Caelum, em especial Alexandre da Silva, Fábio Kung e Thadeu Russo.

Diversos screenshots, remodelamentos e melhorias nos textos foram realizados por Guilherme Moreira e Jacqueline Rodrigues.

Agrecimentos a todas as pessoas que costumam enviar erros, bugs e sugestões para a equipe.

capítulo 2

O sistema

"Explica-se sobretudo o que não se entende.."
Jules Barbey D'Aurevilly -

Neste capitulo iremos:
- levantar os requisitos do sistema
- escolher as tecnologias utilizadas

2.1 - A necessidade do cliente e os testes para aprovação

Nossa empresa foi contratada por uma pequena gravadora de música que deseja entrar na era da música digital. Contratou nossos serviços para desenvolver uma loja virtual que venda downloads de músicas. Detalhe: urgência máxima no projeto.

Fomos chamados para desenvolver uma solução para seus problemas em tempo recorde: somente 20 horas.

Suas maiores preocupações é que o sistema seja capaz de suportar qualquer tipo de banco de dados e a interface de iteração com o cliente seja através da web, com muitos recursos visuais e ajax para atrair os compradores.

O cliente pediu que o sistema passe pelos seguintes testes para aprovar os gastos com o mesmo:

- permitir listar, cadastrar, e remover um usuário que acessa o sistema
- permitir listar, cadastrar, remover e alterar um cd
- permitir listar, cadastrar, remover e alterar uma música
- sistema de login
- permitir alguém selecionar musicas diversas para comprar
- receber dados de relativos à venda, como nome do cliente, email e cartão de credito
- permitir adicionar e remover musicas do carrinho

2.2 - Partes do projeto

Sendo assim, iremos dividir o projeto em algumas partes:

1. criar o ambiente de desenvolvimento
2. criar o banco de dados de usuário
3. refatorar o sistema para web e dao genérico
4. criar sistemas de gerenciamento de Cds e Musicas
5. criar o sistema de login

6. criar a loja virtual, com a interface básica com o usuário
7. inserir recursos de ajax na loja virtual
8. validar
9. dar uma boa turbinada na nossa loja

2.3 - Visual

Para facilitar nosso trabalho, todos os arquivos CSS (Cascade Style Sheet – Folhas de Estilo) já estão prontos. Isso quer dizer que não precisamos nos preocupar com o "visual" da página.

Em um sistema real, freqüentemente há alguém responsável pelo WebDesign que simplesmente nos passo o estilo visual já pronto.

Para usar os estilos já definidos (recomendado!!), inclua o arquivo css/style.css em todas as páginas que fizer. É só colocar a seguinte tag no <head>

```
<link rel="stylesheet" type="text/css" href="css/style.css"/>
```

2.4 - Tecnologias usadas

Ambiente de desenvolvimento

Eclipse 3.2.x com plugins:

- Sysdeo Tomcat
- Amateras HTMLEditor
- Hibernate Tools

Navegador Firefox com extensão Firebug.

Java

- Vraptor 2
- Hibernate 3
- Hibernate Annotations
- Hibernate Validator
- DisplayTag
- JSTL
- Tomcat 5.5

Ajax

JQuery com os seguintes plugins oficiais:
- Interface
- Thickbox
- AjaxForm

2.5 - Sobre o Vraptor

O VRaptor é um poderoso controlador MVC open source. É voltado para

desenvolvimento ágil e de alta produtividade. Implementa um controlador baseado no padrão Front Controller.

É desenvolvido pela Caelum mas disponível para uso geral. Além da própria Caelum, é usado em outros projetos (como o GUJ e o JForum) e por várias empresas.

2.6 - Sobre o Hibernate

O Hibernate é um framework ORM (Object Relational Mapping) que auxilia a camada de persistência de nossa aplicação. Seu objetivo é prover uma interface Orientada a Objetos para nossa programa Java de toda a lógica de persistência em bancos relacionais.

O Hibernate cuida dos SQLs e da tradução dos dados entre esses dois modelos, OO e relacional. Traz como um forte benefício a independência de Banco de Dados, além de ser extremamente simples e produtivo.

É desenvolvido pela Jboss e possui forte penetração no mercado. Utilizaremos, além do pacote principal do Hibernate, uma extensão chamada Hibernate Annotations que traz muita produtividade ao permitir que utilizemos anotações para configurar o Hibernate.

Além disso, usaremos o pacote Hibernate Validator que traz facilidades para a validação de nossos dados.

2.7 - Sobre o JQuery

O JQuery (www.jquery.com) é uma famosa biblioteca Javascript que implementa diversos recursos adicionais ao Javascript usual dos browsers. Possui vários efeitos visuais já implementados e prontos para uso.

Além disso, já possui muitas APIs para trabalhar com Ajax de forma simples e produtiva. O JQuery se integra de forma muito fácil ao VRaptor.

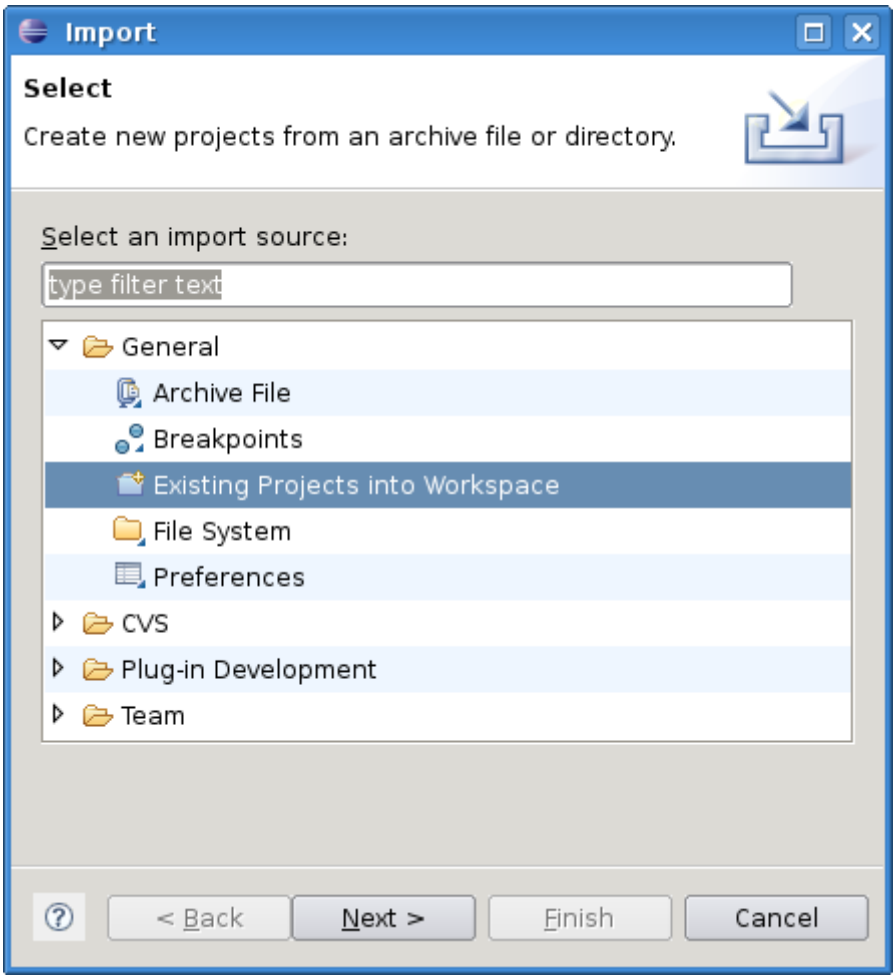
Utilizamos também três plugins oficiais em cima do JQuery para efeitos de Drag'n'Drop (Interface), janelas não intrusivas (ThickBox) e submit de forms via ajax (AjaxForm).

2.8 - Criando o projeto

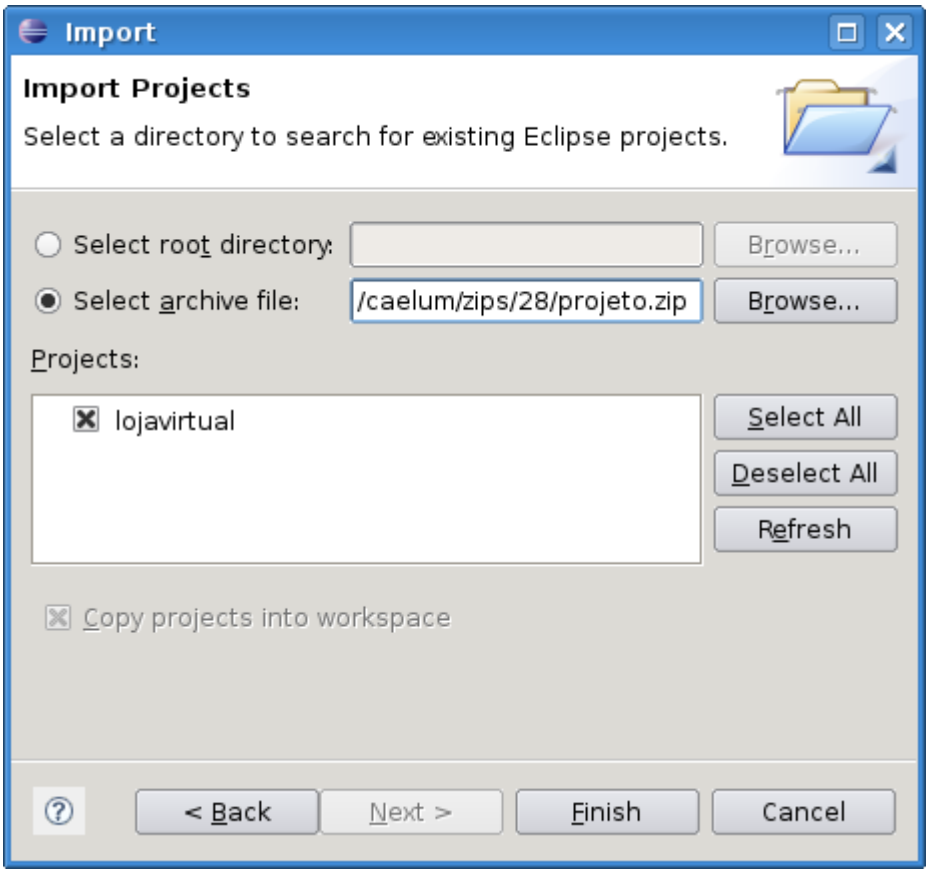
Para iniciar, vamos criar nosso projeto no Eclipse.

1) Abra o Eclipse

2) Vá em **File** -> **Import** e selecione **Existing projects into Workspace**



2) Selecione **Select archive file** e indique o caminho para o zip do projeto (**/caelum/zips/28/projeto.zip**) e selecione o projeto **lojavirtual** :



Escondendo os jars no Eclipse

A lista de jars no nosso projeto é muito grande, o que pode prejudicar a visualização dos arquivos do projeto. No Eclipse, você pode filtrar a visualização de recursos do seu projeto de modo a omiti-los ou exibi-los.

Para fazer o Eclipse não mostrar todos os jars, vá no Menu na View Package Explorer, e selecione Filters. Marque o checkbox Name filter patterns e coloque ***.jar**

log4j.properties

Muitos projetos atualmente usam o log4j do projeto apache como API de log. No nosso caso, tanto o Hibernate quanto o Vraptor utilizam o log4j.

O arquivo log4j.properties, dentro da pasta src, configura o log4j para uso em todo o nosso projeto. Este arquivo foi copiado do modelo do hibernate.

capítulo 3

Controle de usuários

"Os loucos abrem os caminhos que depois emprestam aos sensatos."
Carlo Dossi -

Neste capitulo iremos:
- configurar o hibernate
- começar o projeto pelo cadastro de usuários

3.1 - Primeiros passos

Já sabemos o que nosso sistema deve ter. Precisamos agora começar a codificá-lo, mas por onde? Que tal começarmos modelando entidades e seus Daos?

Nosso primeiro passo será criar o sistema de controle de Usuários do nosso futuro sistema. Usaremos uma entidade sem relacionamentos e operações básicas de adicionar, remover e listar.

Aproveite esse capítulo para entender o funcionamento básico das ferramentas envolvidas no projeto. Nos capítulos posteriores usaremos esse conhecimento na parte mais complexa da aplicação.

3.2 - Preparando o hibernate

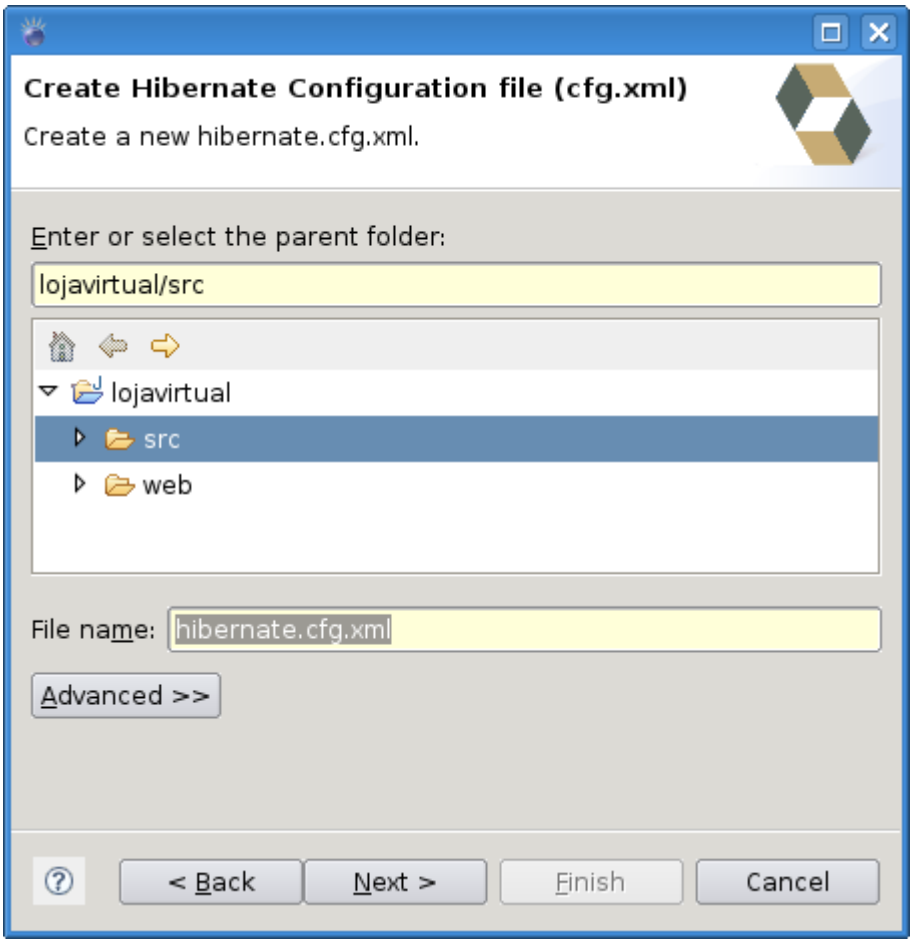
Como vamos usar o Hibernate como ferramenta ORM, precisamos primeiro configurá-lo. Os jars necessários já foram copiados quando criamos o projeto. Precisamos apenas configurar o banco de dados.

3.3 - Exercícios

1) Precisamos agora configurar o Hibernate para usar o MySQL através do **hibernate.cfg.xml**.

Vá em **File** -> **New** -> **Create Hibernate Configuration File**.

Selecione o seu diretório **src** e o nome de arquivo como **hibernate.cfg.xml** e clique em Next.



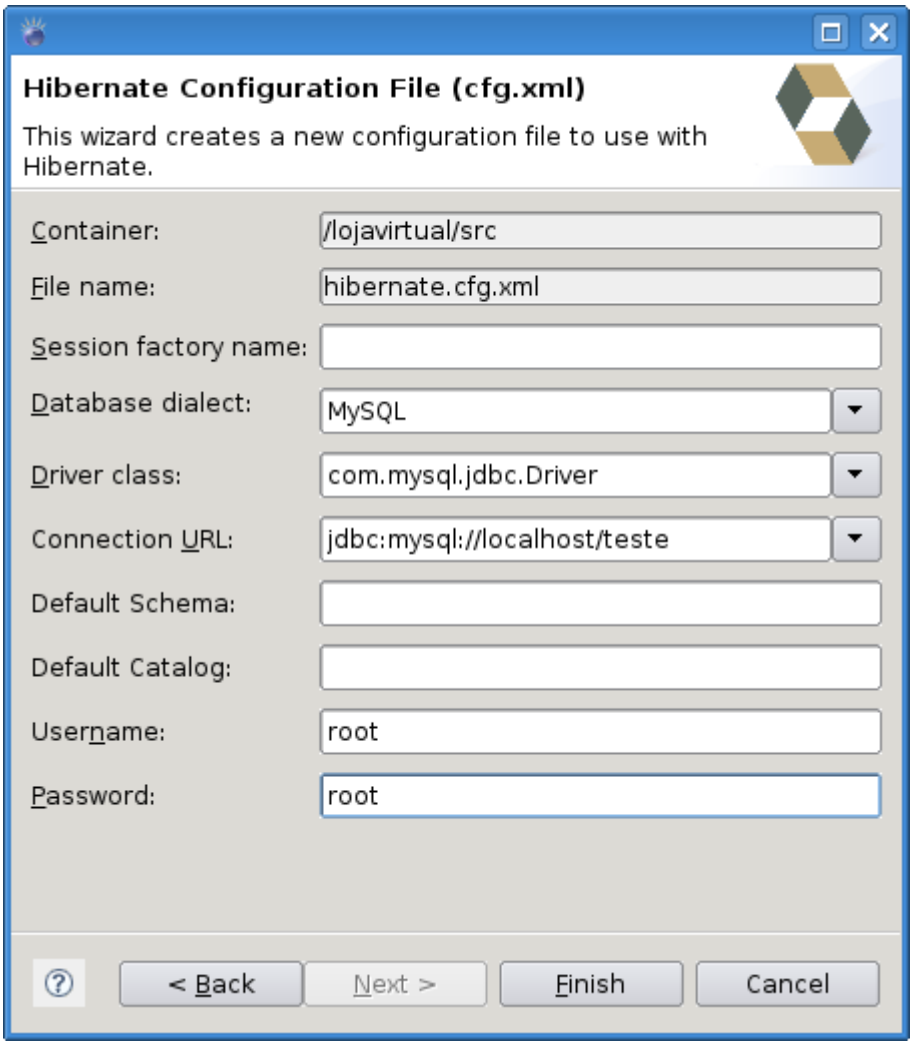
Na próxima tela, selecione as opções para configurar o MySQL:

- Database dialect: **MySQL**

- Driver class: **com.mysql.jdbc.Driver**

- Connection URL: **jdbc:mysql://localhost/teste**

- Username: **root**



(mude as configurações acima de acordo com o banco de dados que estiver utilizando)

2) Adicione duas 2 novas propriedades ao **hibernate.cfg.xml** para que o hibernate mostre o código sql gerado formatado.

No final, seu xml deve estar assim:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
            "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/teste
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
```

```
        </property>
        <property name="hibernate.connection.username">root</property>

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

3.4 - Usuário

Para nosso sistema de usuários, é fundamental a existência de alguma classe que modele o que é um Usuário. Criaremos um JavaBean simples para tal, e o usaremos como Entidade do Hibernate.

3.5 - Exercício

1) Crie a classe Usuario no pacote br.com.caelum.lojavirtual.modelo. Coloque atributos String para login e senha e um atributo Long (não long) para id:

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

public class Usuario {

      private Long id;

      private String login;

      private String senha;

}
```

Acrescente outros atributos se achar necessário.

2) Gere os getters e setters pelo Eclipse (menu Source, Generate getters and setters).

3) Adicione as anotações da Java Persistence API (JPA) para que o Hibernate saiba persistir nossa classe. Lembre-se de **sempre** importar do pacote **javax.persistence**.

a) Adicione o @Entity à classe:

```
@Entity
public class Usuario {
```

b) Adicione @Id e @GeneratedValue ao id:

```
@Id
@GeneratedValue
private Long id;
```

4) Configure nossa classe Usuario no hibernate.cfg.xml. Adicione a tag abaixo *dentro* da tag **session-factory** logo após as tags **property**:

```
<mapping class="br.com.caelum.lojavirtual.modelo.Usuario"/>
```

3.6 - Gerando o banco de dados

Vamos agora gerar o banco de dados para nossa classe Usuario usando o Hibernate.

1) Crie a classe GeraBanco no pacote **br.com.caelum.lojavirtual.util** que, em seu método main, cria a o banco usando o SchemaExport do Hibernate:

```
public class GeraBanco {
      public static void main(String[] args) {
            Configuration conf = new AnnotationConfiguration();
            conf.configure();
            SchemaExport se = new SchemaExport(conf);
            se.create(true, true);
      }
}
```

2) Rode essa classe e veja a saída.

3.7 - Adicionando

Sempre que quisermos usar o Hibernate, precisamos de de uma **Session**. No Hibernate, ao invés de trabalharmos diretamente com Connections JDBC, utilizamos sessões. As sessões do hibernate são bastante interessantes para a implementação de pool de conexões e outros recursos como cache.

Para obter uma sessão do hibernate, precisamos criar uma fábrica de sessões, uma SessionFactory. Desta forma:

```
Configuration conf = new AnnotationConfiguration();
conf.configure();
SessionFactory factory = conf.buildSessionFactory();
Session session = factory.openSession();
```

O Hibernate também facilita muito o controle de transações, que é necessário para operações de insert/delete/update. E para adicionar algo no banco, basta chamar o método **save** na session passando o objeto a ser persistido:

```
Usuario u = new Usuario();
u.setLogin("admin");
u.setSenha("admin");

Transaction t = session.beginTransaction();
session.save(u);
t.commit();
```

3.8 - Exercícios

1) Crie a classe TestaUsuario no pacote

br.com.caelum.lojavirtual.main para testar a adição de um usuario:

```
public class TestaUsuario {

	public static void main(String[] args) {
		// configura o hibernate
		Configuration conf = new AnnotationConfiguration();
		conf.configure();
		SessionFactory factory = conf.buildSessionFactory();
		Session session = factory.openSession();

		// cria um usuario
		Usuario u = new Usuario();
		u.setLogin("admin");
		u.setSenha("admin");

		// abre transacao e insere
		Transaction t = session.beginTransaction();
		session.save(u);
		t.commit();

		session.close();
	}
}
```

2) Rode a classe acima e veja o resultado no console.

Abra o mysql-query-browser e execute uma busca para ver se foi adicionado com sucesso:

SELECT * FROM Usuario;

3.9 - Dao

Como sabemos, ao utilizar acesso a banco de dados, uma prática bastante recomendada é a do uso do padrão DAO (Data Access Object). DAOs encapsulam todo acesso a dados referente às nossas entidades.

Vamos criar uma classe UsuarioDao que encapsula operações de adicionar, remover e editar Usuarios e também listar todos os Usuarios do sistema. Usaremos o Hibernate em nosso Dao, portanto precisamos da Session do Hibernate (que receberemos como argumento no construtor):

```
public class UsuarioDao {

	private Session session;

	UsuarioDao(Session session) {
		this.session = session;
	}

	public void adiciona(Usuario u) {
		this.session.save(u);
	}

	public void remove (Usuario u) {
		this.session.delete(u);
```

```
	}

	public void atualiza (Usuario u) {
		this.session.merge(u);
	}

	public List<Usuario> listaTudo() {
		return this.session.createCriteria(Usuario.class).list();
	}
	public Usuario procura(Long id) {
		return (Usuario) session.load(Usuario.class, id);
	}
}
```

3.10 - Exercício

1) Crie a classe UsuarioDao no pacote **br.com.caelum.lojavirtual.dao**

2) Altere sua classe TestaUsuario para usar o Dao para adicionar e listar.

a) Instancie o UsuarioDao passando a Session para ele:

```
UsuarioDao dao = new UsuarioDao(session);
```

f) Adicione um Usuario através do dao:

```
Usuario u = new Usuario();
u.setLogin("admin");
u.setSenha("admin");

dao.adiciona(u);
```

g) Liste todos os usuários do banco e percorra com um for imprimindo seus logins:

```
List<Usuario> lista = dao.listaTudo();
for (Usuario usuario : lista) {
	System.out.println(usuario.getLogin());
}
```

No final, sua classe deve estar assim:

```
public class TestaUsuario {

	public static void main(String[] args) {
		// configura o hibernate
		Configuration conf = new AnnotationConfiguration();
		conf.configure();
		SessionFactory factory = conf.buildSessionFactory();
		Session session = factory.openSession();

		// cria um usuario
		Usuario u = new Usuario();
		u.setLogin("admin");
		u.setSenha("admin");
```

```
            // cria dao
            UsuarioDao dao = new UsuarioDao(session);

            // abre transacao e insere
            Transaction t = session.beginTransaction();
            dao.adiciona(u);
            t.commit();

            // lista usuarios
            List<Usuario> lista = dao.listaTudo();
            for (Usuario usuario : lista) {
                  System.out.println(usuario.getLogin());
            }

            session.close();
      }
}
```

3) Rode o TestaUsuario e veja a saída.

capítulo 4

Melhorando o sistema e preparando pra Web

"A gata apressada muitas vezes pare gatinhos tísicos."
Teofilo Folengo -

Neste capitulo iremos:
- deixar nosso sistema mais elegante
- aprender o que é refatoração
- preparar o projeto para Web

4.1 - Refatoração

Uma prática bastante comum e difundida no meio da Orientação a Objetos é a chamada Refatoração (Refactoring). Refatorar um programa é melhorar seu código sem alterar sua funcionalidade. A idéia da refatoração não é corrigir bugs, por exemplo, mas melhorar a estrutura de seu código, deixá-lo mais OO.

Há diversos tipos de refatorações. Renomear uma variável para uma de nome mais claro é um exemplo simples. Quebrar um método grande em vários métodos menores é um outro exemplo um pouco mais complicado.

Várias IDEs de Java possui suporte à refatorações comuns. O Eclipse possui ótimas opções automáticas que utilizaremos nesse curso.

O livro mais famoso sobre refatorações foi escrito por Martin Fowler e chama-se *Refactoring – Improving the Design of existing code*. É um catálogo com dezenas de técnicas de refatoração e instruções de como e quando executá-las.

4.2 - Melhorando nosso sistema - HibernateUtil

No capítulo anterior escrevemos uma classe para testar operações com Usuario (adição e lista). A classe ficou um pouco grande e feia, principalmente por fazer várias coisas dentro de um mesmo método (o main).

Sempre que trabalhamos com o Hibernate, precisamos configurá-lo (AnnotationConfiguration). Seria muito ruim copiar esse código todo em vários lugares onde precisamos do hibernate.

Vamos usar o Eclipse para nos ajudar a refatorar o código e melhorar sua estrutura.

Passo 1

O primeiro passo é retirar de dentro do main toda a lógica necessária para configurar o Hibernate e obter uma sessão. Usamos um refactoring conhecido como **extract method**.

a) Selecione as linhas que configuram o hibernate e pegam a sessão:

```
public class TesteUsuarios {
    public static void main(String[] args) {
        // configura o hibernate
        Configuration conf = new AnnotationConfiguration();
        conf.configure();
        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();

        // cria dao
        UsuarioDao dao = new UsuarioDao(session);
```

b) Vá no menu **Refactor** e selecione **Extract Method**. Dê o nome de **getSession** e marque a visibilidade como **public**.

Repare no resultado final. É exatamente a mesma funcionalidade, mas com um código melhor.

Passo 2

Não é interessante a lógica de criação de uma sessão ficar na classe TesteUsuarios, afinal outras classes que usam hibernate precisam de funcionalidade igual.

Uma prática muito comum é encapsular a configuração do hibernate a a fabricação (padrão Factory) de sessões em uma classe separada. Um nome muito comum para essa classe é HibernateUtil (usada até pelo pessoal do hibernate).

a) Crie uma classe chamada **HibernateUtil** no pacote **br.com.caelum.lojavirtual.util**

b) Volte à classe TestaUsuarios e selecione o nome do método getSession:

```
public static Session getSession() {
    // configura o hibernate
    Configuration conf = new Annotation
```

c) Vá em **Refactor** e **Move**. Selecione a classe br.com.caelum.lojavirtual.util.HibernateUtil e clique em OK.

Note como o Eclipse faz o refactoring. Como ele atualiza sua classe TestaUsuarios para chamar o método na classe HibernateUtil.

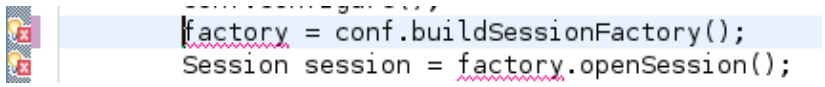
Passo 3

Se deixarmos a configuração do hibernate no getSession, toda vez que alguém precisar de uma sessão toda a configuração será refeita.

Uma prática muito comum é usar um bloco estático para configurar o hibernate, o que garante que ele será executado apenas uma única vez. E o getSession() que fabrica uma nova sessão do Hibernate a partir da SessionFactory.

Para isso, precisamos que *factory* seja um atributo estático e não uma variável local do método.

a) Dentro do getSession, remova a declaração da variável **factory**. O eclipse irá reclamar que ela não existe:



b) Aperte **Ctrl + 1** no erro (quick fix) e selecione **Create field factory**. Agora factory é um atributo estático da classe.

c) Agora crie um bloco estático na classe e mova o código de configuração do hibernate e criação da SessionFactory para dentro dele. Seu getSession deve ficar apenas com a chamada ao openSession e o return.

No final, sua classe HibernateUtil ficou assim:

```
public class HibernateUtil {

	private static SessionFactory factory;

	static {
		Configuration conf = new AnnotationConfiguration();
		conf.configure();
		factory = conf.buildSessionFactory();
	}

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

}
```

4.3 - Melhorando o sistema – DaoFactory

Outra boa prática no nosso sistema, seria criar uma fábrica de Daos. Imagine que teremos várias entidades no nosso sistema, com vários Daos diferentes. Todos esses Daos precisam ser instanciados, e precisam receber uma Session.

Fora isso, ainda precisamos controlar as transações com o hibernate. No nosso sistema com vários daos e várias operações no banco, precisamos definir quando e como abrir e comitar transações.

Vamos encapsular toda essa lógica em uma fábrica de Daos, que cuida da criação dos vários daos que teremos e ainda controla as transações com o Hibernate.

Passo 1

a) Crie a classe DaoFactory no pacote br.com.caelum.lojavirtual.dao

b) Como usaremos uma Session do hibernate para instanciar Daos e também controlar transações, escreva um construtor em DaoFactory que obtenha uma Session através do HibernateUtil e a salve em um atributo:

```
public class DaoFactory {

	private final Session session;

	public DaoFactory() {
		session = HibernateUtil.getSession();
	}
}
```

c) Crie um método de fábrica que devolve um novo UsuarioDao, chamado getUsuarioDao:

```
	public UsuarioDao getUsuarioDao() {
		return new UsuarioDao(this.session);
	}
```

d) Volte à classe TestaUsuarios e mude a forma como ela obtém um dao. Ao invés de pegar a Session e depois instanciar o UsuarioDao, faça ela usar o DaoFactory:

```
public class TesteUsuarios {
	public static void main(String[] args) {
		UsuarioDao dao = new DaoFactory().getUsuarioDao();

		// ... adiciona e lista aqui ...
	}
}
```

Repare como a classe TesteUsuarios ficou agora totalmente independente do Hibernate. Ela não precisa importar nada que seja do Hibernate, nem Session. Foi um ótimo encapsulamento.

Passo 2

Vamos adicionar controle de transações na nossa DaoFactory. Ela deve ter métodos para começar uma transação, fazer commit e rollback. Além disso, adicione um método que verifica se existe uma transação em aberto e outro para fechar tudo.

Seu DaoFactory fica assim:

```
public class DaoFactory {
```

```
	private final Session session;
	private Transaction transaction;

	public DaoFactory() {
		session = HibernateUtil.getSession();
	}

	public void beginTransaction() {
		this.transaction = this.session.beginTransaction();
	}

	public void commit() {
		this.transaction.commit();
		this.transaction = null;
	}

	public boolean hasTransaction() {
		return this.transaction != null;
	}

	public void rollback() {
		this.transaction.rollback();
		this.transaction = null;
	}

	public void close() {
		this.session.close();
	}

	public UsuarioDao getUsuarioDao() {
		return new UsuarioDao(this.session);
	}

}
```

4.4 - O Dao Genérico

Se fôssemos adicionar outra entidade no sistema, um Produto, por exemplo, como você escreveria o ProdutoDao? Provavelmente, copiaria todo o conteúdo do UsuarioDao, substituindo Usuario por Produto, certo? Isso porque, com o Hibernate, a lógica de acesso a dados é muito parecida quando tratamos de entidades diferentes.

Copiar e colar código nunca é legal. Modificações futuras têm que ser feitas em várias lugares, sempre com a chance de se esquecer algum. Seria legal centralizar tudo em um lugar só. Seria legal não precisar criar um dao específico para cada entidade.

Vamos fazer uma única classe Dao que usa Generics e, portanto, pode servir a entidades diferentes sem precisar copiar e colar código.

Passo 1

Use o **Refactor** chamado **Rename** para renomear a nossa classe UsuarioDao para **Dao**. Essa classe será depois generalizada para funcionar com qualquer entidade.

a) Abra a classe UsuarioDao e selecione o nome da classe no código.

b) Vá em **Refactor** -> **Rename** e mude o nome da classe para **Dao**

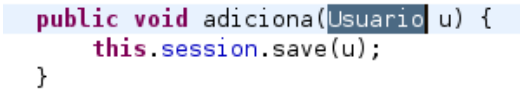
Passo 2

Adicionar generics à nossa classe:

a) Na assinatura da classe, adicione o tipo <T> genérico:

```
public class Dao<T> {
```

b) Agora precisamos mudar todos os métodos que usam Usuario para usarem T. Vá no método adiciona e selecione o parâmetro Usuario:



Aperte **Ctrl + 2**, solte, e aperte **R** (de rename). Note que todas as referências a Usuario na classe ficam com uma borda azul. Digite **T** e note que o Eclipse substitui em toda a classe.

c) Um erro aparece nos imports. Use o atalho do Eclipse de **Organize Imports** para corrigi-lo: **Ctrl + Shift + O**

d) Ainda há um erro em nosso sistema: no método listaTudo, em **T.class**. Isso acontece porque generics é resolvido em tempo de compilação e nenhuma informação é levada para o código executável. O Java não sabe quem é T.

A solução mais comum é mudar o construtor de Dao para receber um objeto Class como argumento, além da Session.

Mude seu Dao para receber essa classe no construtor e salvá-la em um atributo. Depois use esse atributo no método listaTudo.

Seu Dao fica assim no final:

```
public class Dao<T> {

	private final Session session;
	private final Class classe;

	Dao(Session session, Class classe) {
		this.session = session;
		this.classe = classe;
	}

	public void adiciona(T u) {
		this.session.save(u);
	}

	public List<T> listaTudo() {
		return this.session.createCriteria(this.classe).list();
```

```
    }

    // ... atualiza, procura e remove também ...
}
```

Passo 3

No instante que mudamos o construtor de Dao, a DaoFactory deixa de funcionar porque o método getUsuarioDao está com erro.

Corrija o método getUsuarioDao na DaoFactory:

```
public Dao<Usuario> getUsuarioDao() {
      return new Dao<Usuario>(this.session, Usuario.class);
}
```

4.5 - Testando os refactorings

Depois de todos esses passos de refatoração, nosso sistema deve continuar funcionando da mesma forma que antes.

Verifique isso rodando o TestaUsuarios e vendo se ele ainda funciona.

4.6 - Colocando na Web

O que devemos mudar em nosso projeto para ele funcionar na web? Várias coisas, mas usando um framework voltado à POJOs, como o Vraptor, muito menos que com outros frameworks (como o Struts, por exemplo).

Nossos passos serão:

- Instalar o tomcat e configurar o plugin sysdeo

- Configurar nosso projeto no Tomcat

- Mudar nossos exemplos de adição e listagem de usuários para web

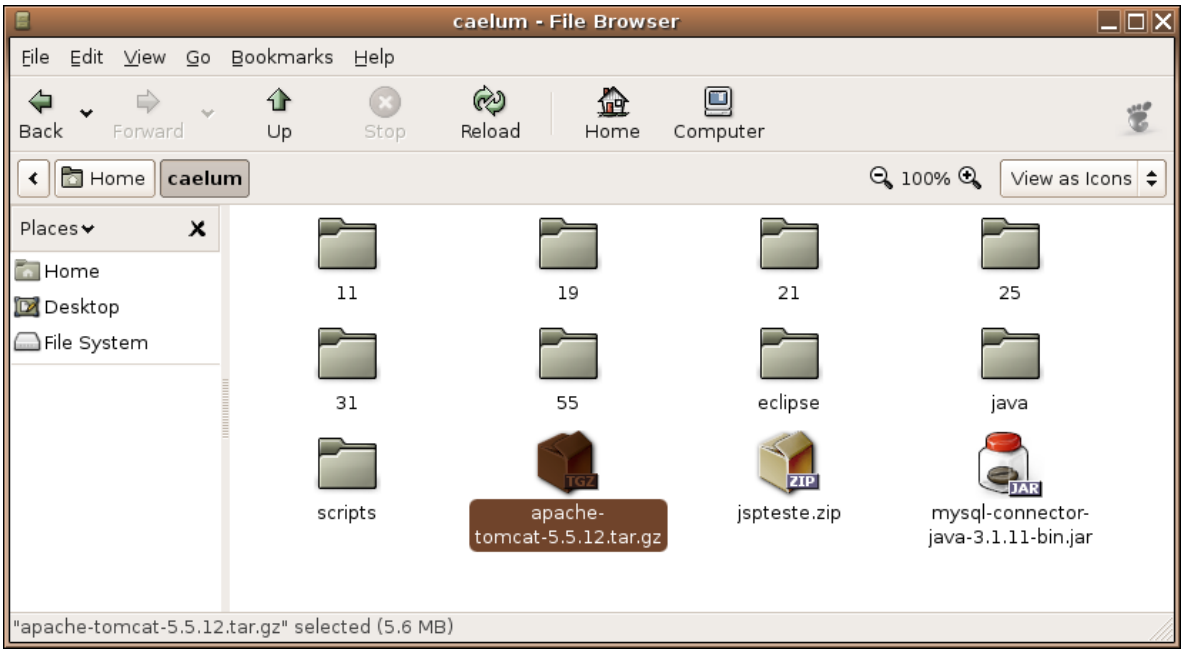
4.7 - Instalar o Tomcat

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

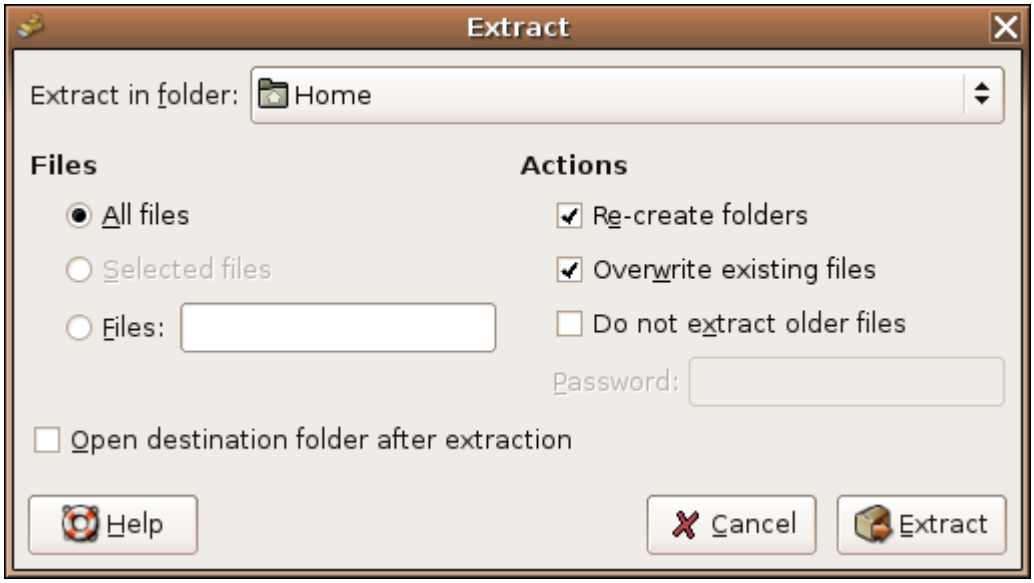
1) No Desktop, entre no atalho caelum.

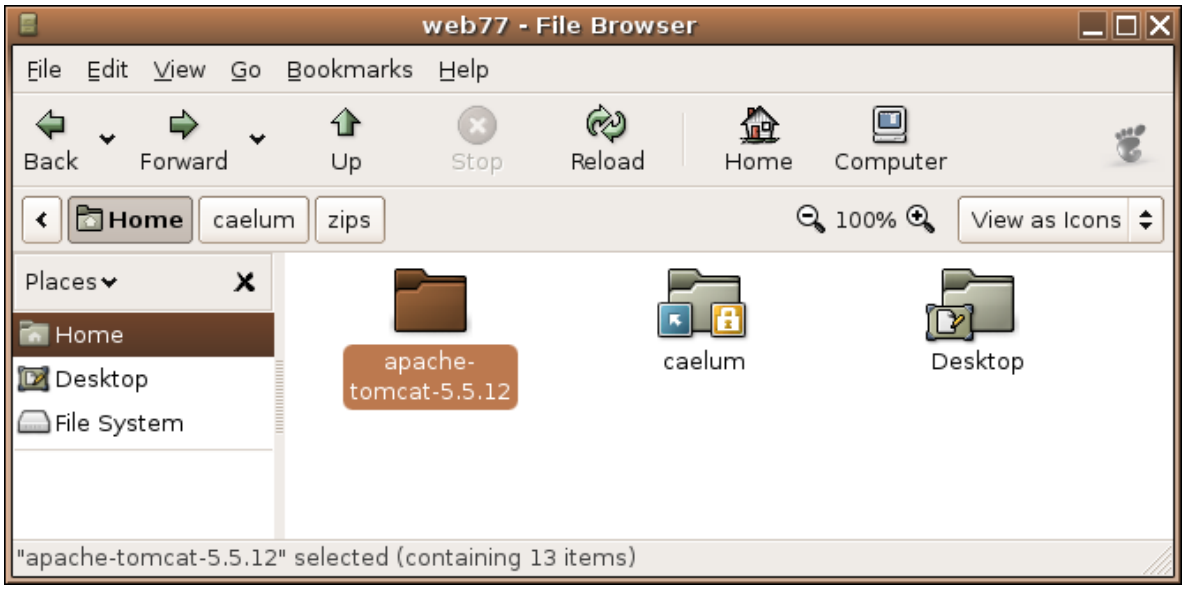
2) Selecione o arquivo do apache-tomcat.

3) Clique da direita escolha Extract to.



4) Escolha a sua pasta principal: **Home** (o seu nome de usuário) e selecione **Extract**.





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

Tomcat

Baixe o tomcat em http://tomcat.apache.org no link de download binaries.

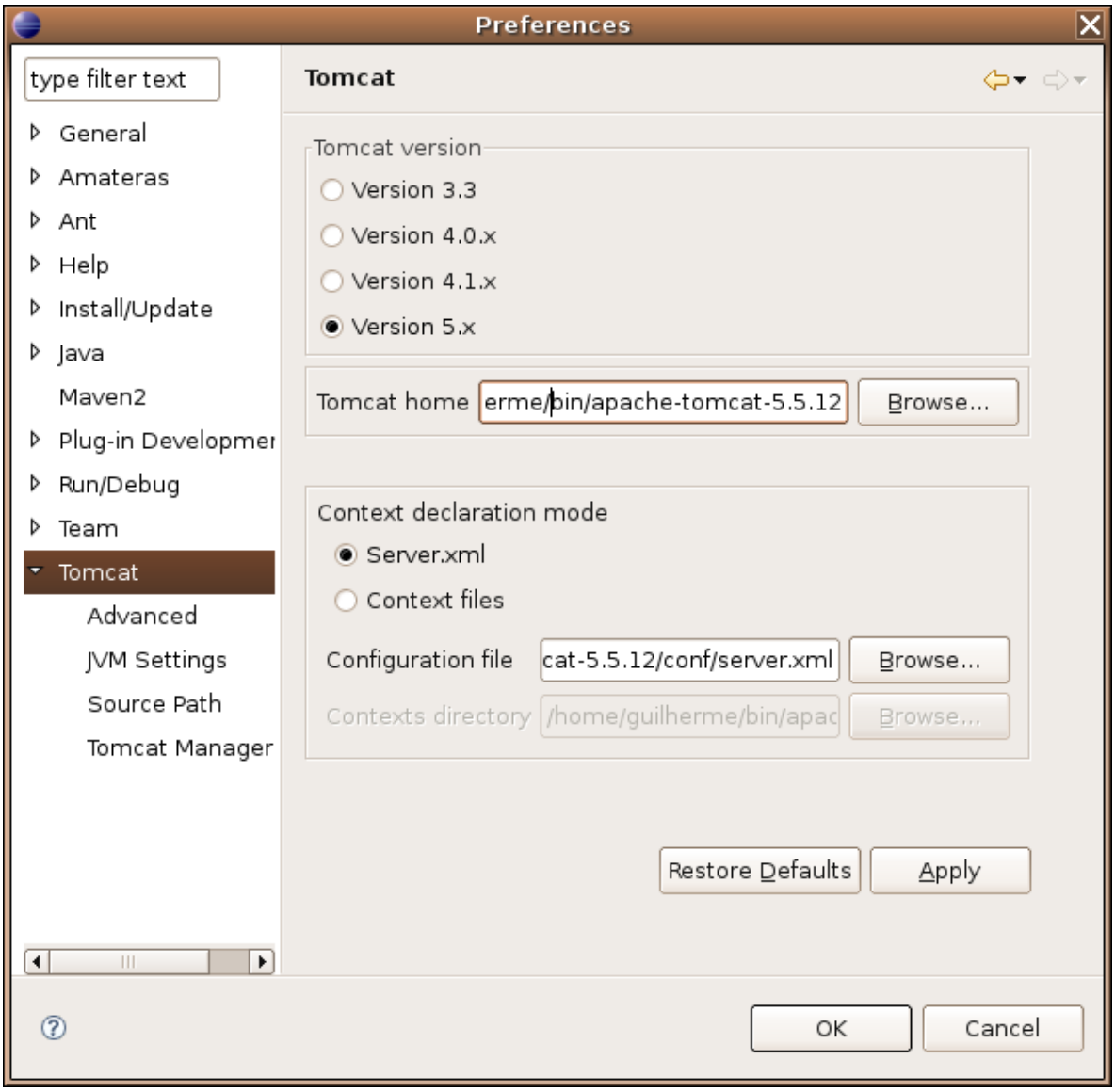
O Tomcat virou implementação padrão e referência de novas apis de servlets, isto é, quando uma nova especificação surge, o tomcat costuma ser o primeiro servlet contêiner a implementar a nova api.

4.8 - Configurando o plugin do tomcat no eclipse

Vá no menu **Window**, **Preferences**, **Tomcat**:

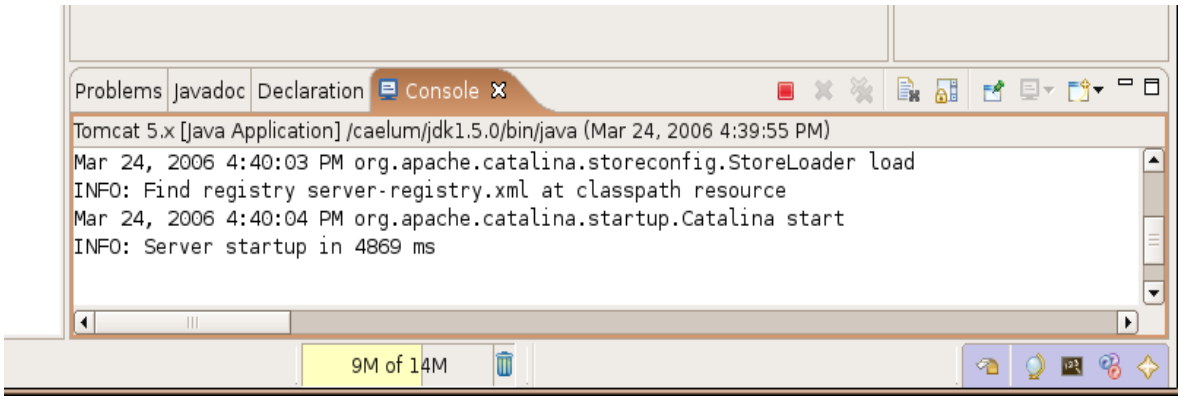
1-) Selecione a versão 5.x do tomcat.

2-) Selecione **tomcat home** e coloque o diretório onde instalou o tomcat.



3-) Aplique as alterações.

4-)Clique no botão do tomcat que está na barra de ferramentas. 

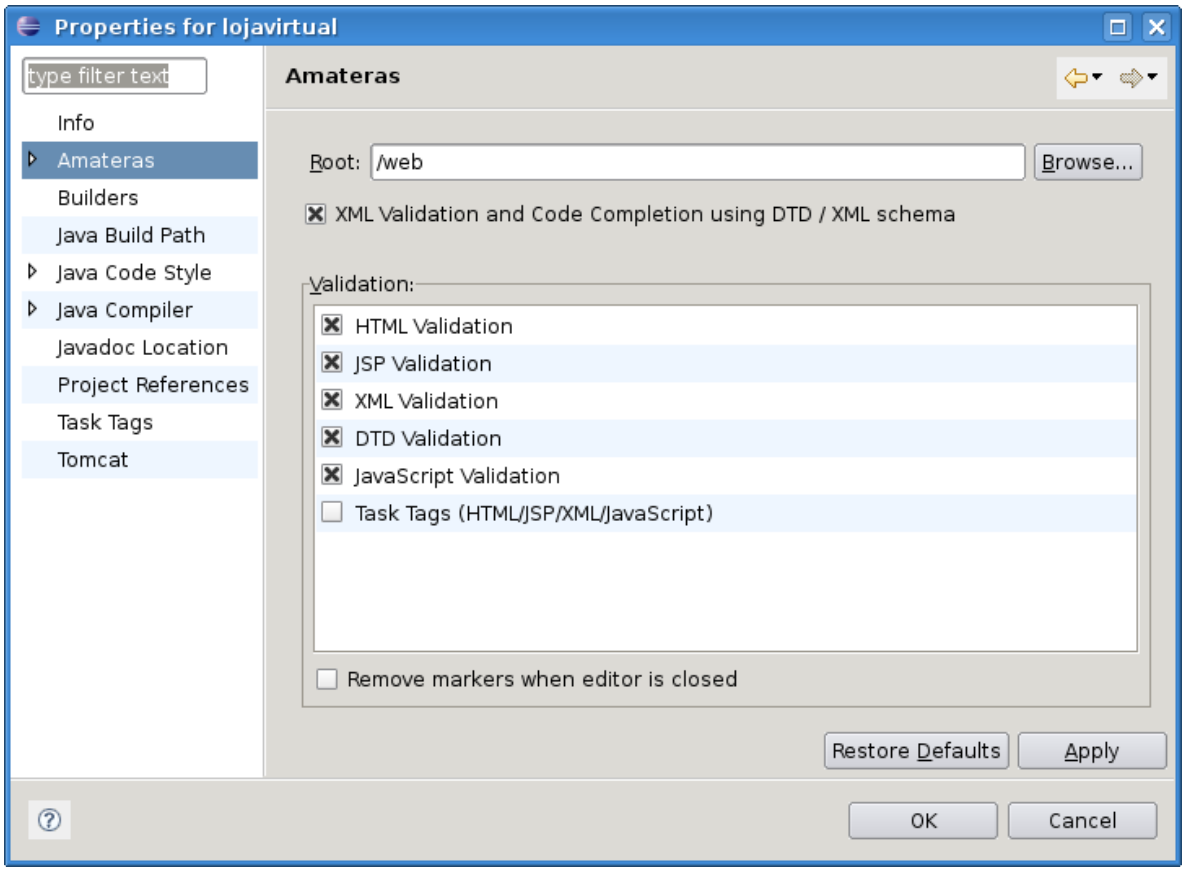


5-) Abra o seu browser e tente acessar: **http://localhost:8080**

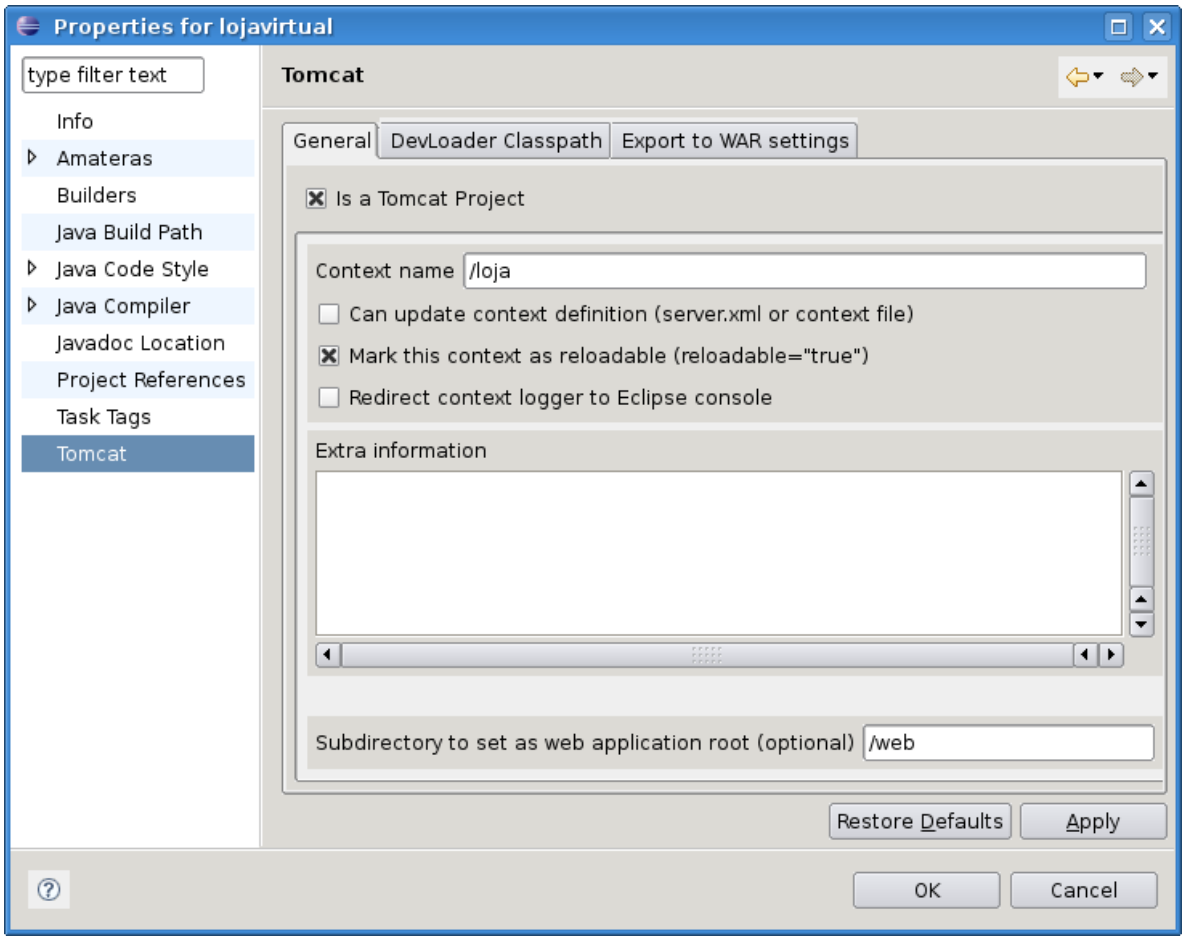
4.9 - Configurar nosso projeto no tomcat

1) Clique com o botão direito no projeto e vá em Propriedades

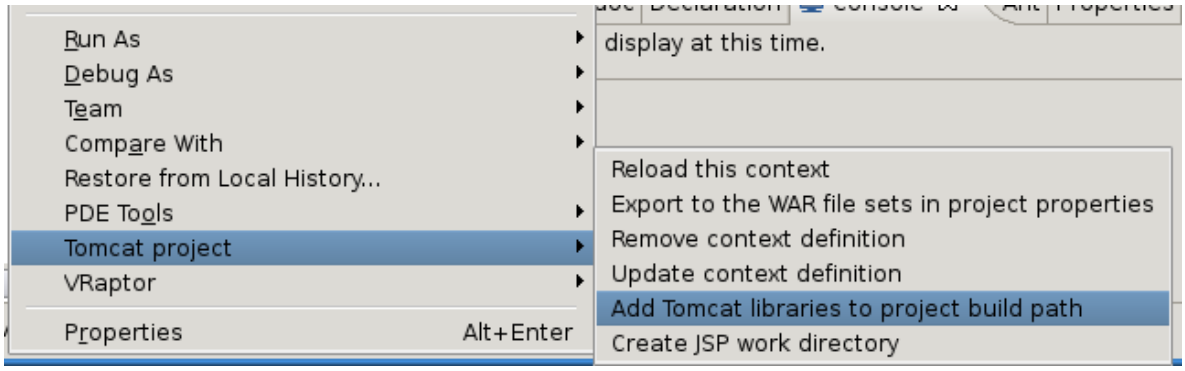
2) Selecione **Amateras**. No campo **ROOT** coloque **/web** (não esqueça a barra)



3) Agora selecione **Tomcat**. Marque a opção **is a Tomcat project**. Coloque o **Context name** como **/loja** e o **Subdiretory to set as root** como **/web** . Clique em OK



4) Clique da direita no projeto e vá em **Tomcat Project** -> **Add tomcat libraries to project buildpath**



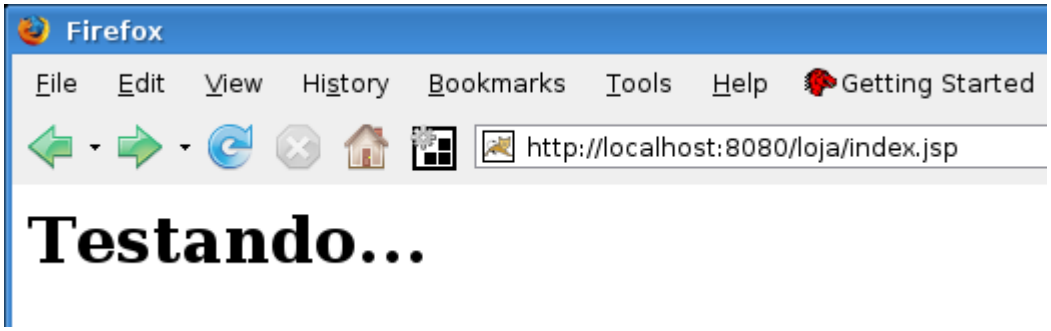
5) Clique da direita no projeto e vá em **Tomcat Project** -> **Update context definition**

6) Crie um arquivo **index.jsp** na pasta **web/** com algum conteúdo de teste:

```
<html><h1>Testando...</h1></html>
```

7) Inicie o Tomcat se ele já não estiver iniciado  e acesse no navegador:

http://localhost:8080/loja/index.jsp



capítulo 5

Cadastro de usuários com Vraptor

"Não basta mostrar a verdade, é preciso apresentá-la amavelmente.."
François Fénelon -

Neste capitulo iremos trabalhar com:
- componentes do vraptor
- interceptadores
- taglib displaytag para tabelas

5.1 - Lógica do vraptor

O vraptor trabalha com a idéia de POJOs (Plain Old Java Objects) como componentes para executar as lógicas de negócio.

A idéia é criar objetos muito simples, com código Java comum, que o framework possa entender.

Uma "action" necessária para adicionar um usuário no sistema não deve ser mais complexa que isso:

```
public class UsuarioLogic {

	public void adiciona (Usuario usuario) {
		// ... logica de adicionar no banco aqui ...
		System.out.println("Adiciona usuario " + usuario.getLogin());
	}
}
```

Observe o método adiciona, como ele é simples e totalmente desconectado da web.

Mas no ambiente web, existe um agravante: recebemos os dados como parâmetros no Request. Deveríamos recuperá-los, convertê-los no nosso objeto java e passar como argumento no método. Esse era o jeito comum antes dos frameworks MVC, mas com o vraptor todos esses passos são feitos automaticamente.

Quando o Vraptor percebe que nossa lógica recebe um Usuario ele procura no request por parâmetros como "usuario.xxx". Então ele cria um objeto Usuario e popula seus atributos com os métodos setXxx(). E esse objeto convertido e populado é passado para nosso método como argumento.

Nós, programadores, não precisamos nos preocupar com coisas de

baixo nível envolvendo HttpServletRequest. Nosso framework cuida dessas coisas pra gente, para que possamos focar na lógica de negócios.

A única configuração necessária é colocar @Component na nossa classe:

```
@Component
public class UsuarioLogic {
```

5.2 - Adicionar usuário

Precisamos de um formulário html para adicionar um usuário. Este formulário deve ter campos para colocar login e senha, e um botão de enviar.

O action do form deve enviar para nosso componente, para o método adiciona dele. Mas qual é a url que dispara esse meu método? No vraptor, não precisamos configurar todas as urls em xmls imensos, basta usar um **padrão**. No nosso caso, o padrão é **usuario.adiciona.logic**.

É **.logic** porque foi a extensão que demos no web.xml. É **adiciona** porque é o método que querermos. E é **usuario** porque é o nome do nosso componente.

Porque meu componente chama usuario?

Por padrão, o vraptor retira o sufixo Logic e coloca a primeira letra em minúscula na hora de dar o nome do componente. (muitos frameworks, como o Stripes, adotam postura semelhante)

Você pode se quiser colocar outro nome de componente dentro da anotação @Component:

```
@Component("user")
public class UsuarioLogic {
```

```
<form action="usuario.adiciona.logic">
      Login: <input type="text" name="usuario.login" /><br/>
      Senha: <input type="text" name="usuario.senha" /><br/>
      <input type="submit" />
</form>
```

Certo, então chamar **usuario.adiciona.logic** dispara nossa action. Mas e depois? O que é exibido depois da action ser executada?

Em frameworks como struts, teríamos que dizer qual é o próximo passo, qual o JSP com a mensagem de "Usuario adicionado!". No vraptor, há uma **convenção** para isso. Depois que a lógica **usuario.adiciona.logic** for executada, o fluxo segue para o jsp **usuario/adiciona.ok.jsp** ou seja, nomeDoComponente/nomeDaLogica.ok.jsp (ok indica que o metodo foi executado com sucesso).

Neste arquivo colocaremos uma mensagem de sucesso.

Já sabemos como nossa lógica vai ser chamada e qual jsp vai ser exibido, mas como chamamos o formulario para adicionar um usuário? Podemos fazer um jsp simples e chamá-lo diretamente, mas isso não é muito bom.

Um padrão muito conhecido desde a época do Struts é o **Always link to actions**. Isso quer dizer que nunca devemos acessar um jsp diretamente, mas sempre através de uma action, mesmo que hoje essa sua action não faça nada. No futuro, se algo precisar ser feito antes do form ser exibido, fica muito fácil de implementar.

Vamos fazer então com que, ao chamar **usuario.formulario.logic** o nosso formulário seja exibido. Precisamos de um método formulário na nossa UsuarioLogic que não faz nada:

```
	public void formulario() {
	}
```

E nosso jsp deve estar em **usuario/formulario.ok.jsp**

5.3 - Exercícios

1) Crie a **UsuarioLogic** no pacote **br.com.caelum.lojavirtual.logic**:

```
@Component
public class UsuarioLogic {

	public void adiciona (Usuario usuario) {
		// ... logica de adicionar no banco aqui ...
		System.out.println("Adiciona usuario " + usuario.getLogin());
	}
}
```

2) Crie uma pasta chamada **usuario** dentro da pasta **web**

3) Crie um arquivo **formulario.ok.jsp** dentro da pasta **web/usuario/** usando o menu: **File -> New** -> **Amateras, JSP File**

4) Implemente o **formulario.ok.jsp** da seguinte forma:

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<html>
<head>
	<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
	<title>Cadastro de usuarios</title>
	<link rel="stylesheet" type="text/css" href="css/style.css"/>
</head>
<body>

	<h1>Cadastro de usuarios</h1>

	<form action="usuario.adiciona.logic">

		Login: <input type="text" name="usuario.login" /><br/>
		Senha: <input type="text" name="usuario.senha" /><br/>
```

```
            <input type="submit" />

        </form>

</body>
</html>
```

Note os nomes dos inputs, como devem ser feitos para que o vraptor saiba como popular nosso bean.

Estilo

Não se esqueça de incluir o CSS de estilo nos JSPs que criar! Coloque sempre dentro da tag <head>:
<link rel="stylesheet" type="text/css" href="css/style.css"/>

4) Crie um método formulário vazio na classe UsuarioLogic:

```
public void formulario() {
}
```

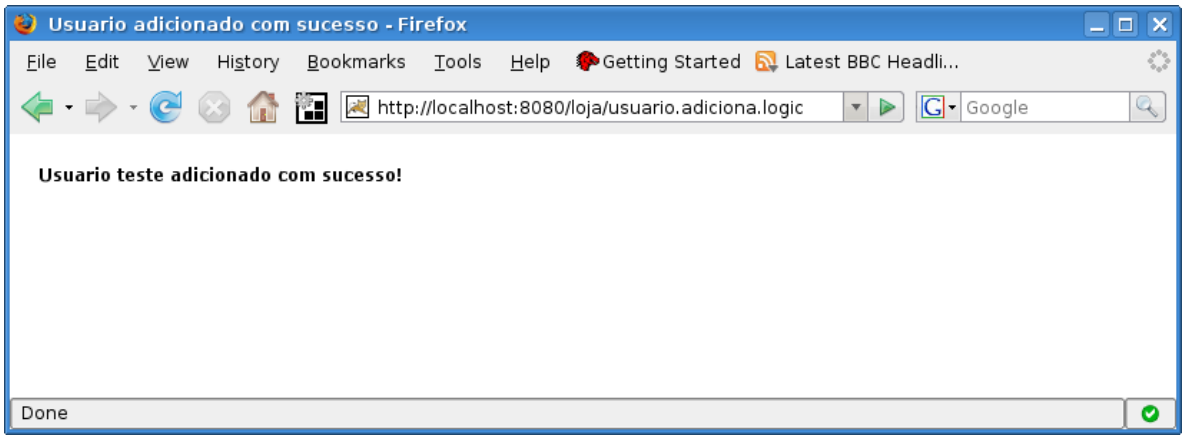
5) Crie um jsp (pelo menu do Amateras) na pasta **usuario** chamado **adiciona.ok.jsp** com o seguinte conteúdo:

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<html>
<head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>Usuario adicionado com sucesso</title>
        <link rel="stylesheet" type="text/css" href="css/style.css"/>
</head>
<body>

        <h2>Usuario ${param['usuario.login']} adicionado com sucesso!</h2>

</body>
</html>
```

6) Teste o sistema acessando:

http://localhost:8080/loja/usuario.formulario.logic

Coloque um novo login e senha e envie o form:



5.4 - Injetando o DaoFactory com interceptador

Nossa UsuarioLogic acima demonstra toda a funcionalidade básica do Vraptor, mas ainda não adiciona realmente um usuario no banco de dados. Para fazer essa adição, precisamos do Hibernate, ou melhor precisamos do nosso **DaoFactory**.

Mas como obter esse DaoFactory? Será que podemos simplesmente implementar o método adiciona assim:

```
public void adiciona (Usuario usuario) {
      DaoFactory factory = new DaoFactory();
      factory.beginTransaction();
      factory.getUsuarioDao().adiciona(usuario);
      factory.commit();
      factory.close();
}
```

Até poderíamos fazer isso, mas nos traria problemas mais a frente.

Quando trabalhamos com MVC na web, depois de nossa lógica (o método adiciona) o fluxo é redirecionado para um JSP de visualização. E devido ao comportamento Lazy do Hibernate, quando precisarmos exibir coisas no JSP que precisariam de buscas no banco, nossa session estará fechada. Isso porque colocamos o controle do ciclo de vida de nossa Session diretamente na lógica.

Na verdade, nossa Session (e nossa DaoFactory) deve viver mais que apenas no escopo da lógica. Ela deve estar aberta ainda na View, no JSP. A melhor forma de fazer isso é não criando a DaoFactory na lógica mas deixar alguém (quem?) criar e passar pra gente:

```
public class UsuarioLogic {

      private final DaoFactory daoFactory;

      public UsuarioLogic(DaoFactory daoFactory) {
            this.daoFactory = daoFactory;
      }
```

```
	public void adiciona (Usuario usuario) {
		this.daoFactory.beginTransaction();
		this.daoFactory.getUsuarioDao().adiciona(usuario);
		this.daoFactory.commit();
	}
}
```

A nossa lógica agora possui um construtor que recebe um DaoFactory como argumento. O DaoFactory é uma **dependência** da nossa classe. Não conseguimos usá-la sem passar um DaoFactory para ela!

Mas da onde vem esse DaoFactory? Quem cria instâncias da nossa classe deve se preocupar em passar o DaoFactory como argumento para nosso método. Mas se estamos escrevendo um componente do vraptor, será ele que criará nosso objeto!

Como o vraptor sabe o que passar como argumento para nosso construtor? Temos de dizer a ele. Vamos usar um **interceptador** do vraptor que cria um objeto DaoFactory e o entrega ao vraptor para que esse possa **injetar** na nossa classe.

Um interceptador é algo bem parecido com o Filter de servlet. É algo que será executado antes da sua lógica ser executada, e depois, no final de tudo. Assim como com filtros, podemos ter uma **cadeia de interceptadores** que fazem várias coisas diferentes. E também podemos bloquear a continuação da execução da cadeia.

Um exemplo bastante usado de interceptadores ou filtros é para sistemas de login, onde verificamos se o usuário está logado ou não antes da lógica ser executada. Se não estiver logado, não deixa a execução prosseguir. (nós vamos fazer algo parecido mais pra frente)

Neste instante, criaremos um interceptador que será executado antes e depois da nossa lógica para registrarmos um DaoFactory. Com o DaoFactory registrado, o vraptor encontra ele e usa para **injeção** na sua lógica de negócios.

Para isso iremos construir uma classe chamada DaoInterceptor no pacote logic. Esse interceptador deve implementar a interface Interceptor:

```
public class DaoInterceptor implements Interceptor {
	public void intercept(LogicFlow flow) throws LogicException, ViewException {
	}
}
```

Aproveitaremos nosso Interceptor para, além de criar a DaoFactory, também gerenciar nossas transações.

Dentro do método intercept iremos primeiro chamar a execução, após a mesma verificamos se sobrou alguma transaction aberta e, nesse caso, executamos um rollback. Finalizamos fechando a sessão.

```
	public void intercept(LogicFlow flow) throws LogicException, ViewException {
```

```
            // executa a logica
            flow.execute();

            // se sobrou transacao sem comitar, faz rollback
            if (factory.hasTransaction()) {
                  factory.rollback();
            }

            factory.close();
      }
```

Perceba que só fechamos a factory (e a sessão) no final de tudo, depois da chamada ao execute(). Um interceptor nos garante que o código depois da chamada ao execute() só será chamada depois que nossa lógica e nosso JSP forem chamados.

Ou seja, conseguimos com que nossa Session esteja viva tanto na lógica quanto no JSP. E ainda conseguimos controlar as transações de forma mais robusta.

Mas agora falta a variável factory! Como iremos "outjetá-la"? Criamos a mesma como variável membro e criamos um getter para ela:

```
      private DaoFactory factory = new DaoFactory();

      public DaoFactory getFactory() {
            return factory;
      }
```

Para deixarmos nossa **DaoFactory** disponível para os componentes, usamos a anotação @Out. Precisamos passar uma chave para a anotação que é exatamente o nome completo da classe DaoFactory:

```
      @Out(key="br.com.caelum.lojavirtual.dao.DaoFactory")
      public DaoFactory getFactory() {
```

5.5 - Como configurar o uso dos interceptors

Primeiro vamos configurar o uso de tais interceptadores em nossas lógicas.

Para isso, abra cada uma das suas quatro lógicas e adicione a anotação @InterceptedBy em suas classes, por exemplo:

```
      @InterceptedBy(DaoInterceptor.class)
      public class UsuarioLogic {
```

5.6 - Exercícios

1) Crie o DaoInterceptor no pacote br.com.caelum.lojavirtual.loja:

```
public class DaoInterceptor implements Interceptor {

      private final DaoFactory factory = new DaoFactory();
```

```
	public void intercept(LogicFlow flow) throws LogicException, ViewException {

		// executa a lógica
		flow.execute();

		// se sobrou transação sem comitar, faz rollback
		if (factory.hasTransaction()) {
			factory.rollback();
		}

		factory.close();
	}

	@Out(key="br.com.caelum.lojavirtual.dao.DaoFactory")
	public DaoFactory getFactory() {
		return factory;
	}
}
```

2) Altere sua classe UsuarioLogic para ser interceptada pelo DaoInterceptor:

a) Adicione a anotação @InterceptedBy no topo de sua classe

```
@Component
@InterceptedBy(DaoInterceptor.class)
public class UsuarioLogic {
```

b) Adicione um atributo DaoFactory na sua classe:

```
	private final DaoFactory daoFactory;
```

c) Crie um construtor que recebe DaoFactory e inicializa o atributo

```
	public UsuarioLogic(DaoFactory daoFactory) {
		this.daoFactory = daoFactory;
	}
```

d) Implemente o método **adiciona** de forma a adicionar usando a DaoFactory

```
	public void adiciona (Usuario usuario) {
		this.daoFactory.beginTransaction();
		this.daoFactory.getUsuarioDao().adiciona(usuario);
		this.daoFactory.commit();
	}
```

No fim, sua UsuarioLogic deve estar assim:

```
@Component
@InterceptedBy(DaoInterceptor.class)
public class UsuarioLogic {

	private final DaoFactory daoFactory;

	public UsuarioLogic(DaoFactory daoFactory) {
```

```
            this.daoFactory = daoFactory;
      }

      public void adiciona (Usuario usuario) {
            this.daoFactory.beginTransaction();
            this.daoFactory.getUsuarioDao().adiciona(usuario);
            this.daoFactory.commit();
      }
}
```

3) Teste novamente seu sistema. Mas desta vez olhe no banco de dados para ver o usuário inserido.

mysql -u root teste
select * from Usuario;

5.7 - Listando com displaytag

Para listar os usuário, precisamos de uma lógica nova no nosso UsuarioLogic. Além disso, precisamos salvar a lista a ser enviada para o JSP em um atributo e criar um getter pra ele.

```
      private List<Usuario> usuarios;

      public void lista () {
            usuarios = this.daoFactory.getUsuarioDao().listaTudo();
      }

      public List<Usuario> getUsuarios() {
            return usuarios;
      }
```

O JSP para exibir a lista será o **web/usuario/lista.ok.jsp**. Usando a displaytag para a listagem, nosso jsp fica assim:

```
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>

<h1>Usuários</h1>

<display:table id="usuario"" name="${usuarios}" requestURI="usuario.lista.logic">
  <display:column property="id" sortable="true"/>
  <display:column property="login" />
</display:table>
```

5.8 - Exercício

1) Crie a lógica de listagem:

a) Abra a classe UsuarioLogic

b) Crie um atributo usuarios:

```
      private List<Usuario> usuarios;
```

c) Crie um getter para ele:

```
public List<Usuario> getUsuarios() {
	return usuarios;
}
```

d) Crie um método lista que chama o listaTudo do DAO:

```
public void lista () {
	usuarios = this.daoFactory.getUsuarioDao().listaTudo();
}
```

2) Crie um jsp chamado lista.ok.jsp dentro da pasta usuario através do amateras.

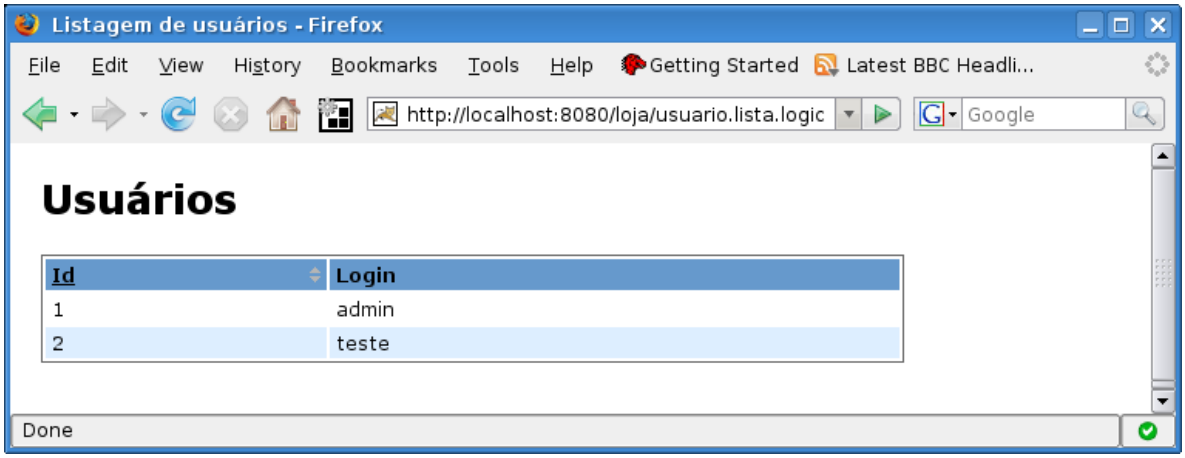
a) Adicione o cabeçalho da displaytag no começo do arquivo:

```
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>
```

b) Implemente dentro do <body> a listagem com a displaytag:

```
<h1>Usuários</h1>

<display:table id="usuario" name="${usuarios}" requestURI="usuario.lista.logic">
  <display:column property="id" sortable="true"/>
  <display:column property="login" />
</display:table>
```

3) Teste a url http://localhost:8080/loja/usuario.lista.logic



5.9 - Redirecionando depois do adiciona

Um recurso interessante do vraptor é a possibilidade de mudar a convenção padrão de que ao executar usuario.adiciona.logic a view usuario/adiciona.ok.jsp é executada.

Podemos criar um arquivo views.properties no nosso diretório src e, lá, mudar a localização padrão da view a ser executada. Para fazer a listagem ser executada depois da lógica de adição, adicionamos:

```
usuario.adiciona.ok = usuario.lista.logic
```

5.10 - Exercícios

1) Crie um arquivo views.properties na pasta src com o seguinte conteúdo:

```
usuario.adiciona.ok = usuario.lista.logic
```

2) Teste a adição de um novo usuario pela url http://localhost:8080/loja/usuario.formulario.logic

5.11 - Removendo usuários

Para fazer a remoção, precisamos de um método remove na nossa UsuarioLogic que recebe um Usuario a ser removido:

```
public void remove (Usuario usuario) {
      this.daoFactory.beginTransaction();
      this.daoFactory.getUsuarioDao().remove(usuario);
      this.daoFactory.commit();
}
```

Depois da remoção, redirecionamos para a lista. Então adicione no views.properties:

```
usuario.remove.ok = usuario.lista.logic
```

Na nossa listagem, adicionamos uma coluna na table que chama usuario.remove.logic passando o id do usuário. Veja como fazer uma coluna com um link usando a displaytag:

```
<display:column>
    <a href="usuario.remove.logic?usuario.id=${usuario.id}">remover</a>
</display:column>
```

Isso irá criar um link "remover" em cada linha que chama usuario.remove.logic passando um parâmetro usuario.id.

5.12 - Exercícios opcionais

1) Adicione um método remove na sua UsuarioLogic:

```
public void remove (Usuario usuario) {
      this.daoFactory.beginTransaction();
      this.daoFactory.getUsuarioDao().remove(usuario);
      this.daoFactory.commit();
}
```

2) Adicione uma nova coluna na displaytag do **lista.ok.jsp**:

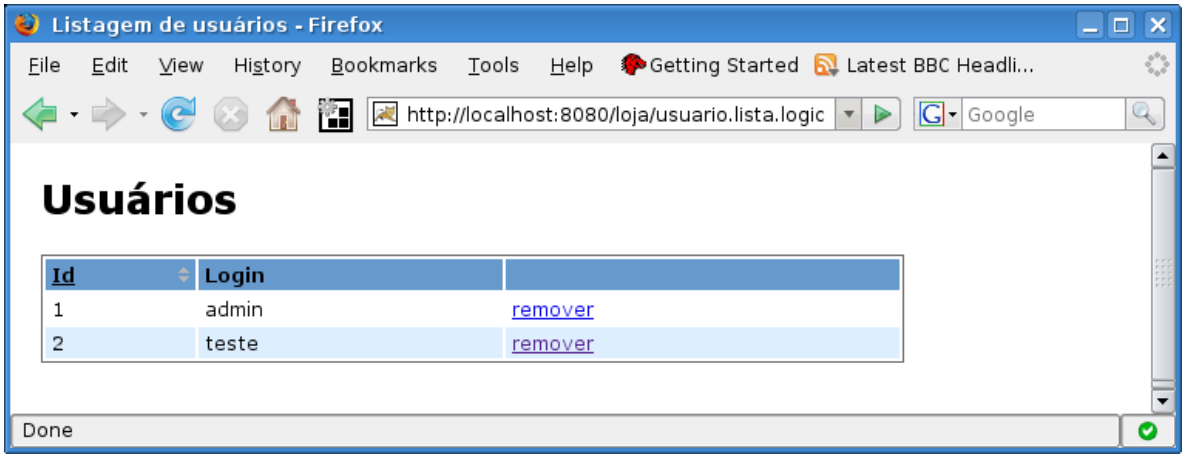
```
<display:column>
    <a href="usuario.remove.logic?usuario.id=${usuario.id}">remover</a>
```

```
</display:column>
```

3) Adicione a seguinte linha ao **views.properties**:

```
usuario.remove.ok = usuario.lista.logic
```

4) Teste a listagem de usuários e clicar em "remover".



capítulo 6

Cds e músicas

"Is man merely a mistake of God's? Or God merely a mistake of man's?"
Friedrich Nietzsche -

Neste capítulo iremos:
- modelar Cd e Musica
- criar lógicas para adicionar, remover, editar e listar Cds e Musicas

6.1 - Menu

Antes de prosseguirmos criando entidades e lógicas, vamos criar um html bem simples para servir de menu no nosso site em **web/menu.jsp**

Esse html deverá ser incluído em todas as páginas que queremos o menu, logo depois da tag body.

A idéia é que, com o tempo, você adicione mais links neste menu para facilitar a navegação no site.

6.2 - Exercício

1) Crie um arquivo **menu.jsp** na pasta **web** com o seguinte conteúdo:

```
<ul class="menu">
      <li><a href="usuario.lista.logic">Lista de Usuarios</a></li>
      <li><a href="usuario.formulario.logic">Novo Usuario</a></li>
</ul>
```

2) Abra os arquivos usuario/formulario.ok.jsp e usuario/lista.ok.jsp e adicione logo depois da abertura tag body o include do menu.jsp:

```
<body>
<%@ include file="../menu.jsp" %>
```

6.3 - Entidades

Crie duas entidades: Cd e Musica. Elas terão um relacionamento ManyToOne, de modo que um cd tem várias músicas e cada música, um cd.

6.4 - Exercício

1) Crie a classe **Cd** no pacote **br.com.caelum.lojavirtual.modelo**:

```
@Entity
```

```
public class Cd {

	@Id
	@GeneratedValue
	private Long id;

	private String titulo;

	private String artista;

	private String genero;

	// gere os getters e setters
}
```

(adicione outros atributos que achar necessário...)

2) Crie a classe **Musica** no pacote **br.com.caelum.lojavirtual.modelo**:

```
@Entity
public class Musica {

	@Id
	@GeneratedValue
	private Long id;

	private String titulo;

	private Double preco;

	@ManyToOne
	private Cd cd;

	// gere os getters e setters
}
```

(adicione outros atributos que achar necessário...)

3) Adicione Cd e Musica como entidades ao **hibernate.cfg.xml**:

```
<mapping class="br.com.caelum.lojavirtual.modelo.Cd"/>
<mapping class="br.com.caelum.lojavirtual.modelo.Musica"/>
```

4) Rode o **GeraBanco** novamente.

5) Adicione métodos **getCdDao()** e **getMusicaDao()** à classe DaoFactory:

```
public Dao<Cd> getCdDao() {
	return new Dao<Cd>(this.session, Cd.class);
}

public Dao<Musica> getMusicaDao() {
	return new Dao<Musica>(this.session, Musica.class);
}
```

6.5 - CdLogic

Crie a lógica para remover, listar e editar cds.

Adição, remoção e listagem são análogos ao que já vimos na UsuarioLogic. A novidade é a edição.

Para editar alguma coisa do banco, precisamos antes carregar os dados e exibir no formulário. O usuário então faz as alterações necessárias, envia, e nós atualizamos o banco.

Carregaremos os dados para edição na lógica **edita** que recebe um Cd (na verdade, só precisa do id) e busca no banco todos os dados do Cd:

```
public void editar(Cd cd) {
      this.cd = this.daoFactory.getCdDao().procura(cd.getId());
}
```

Esses dados devem então ser exibidos no formulário. Precisaríamos de um segundo form além daquele de adição em branco: um para edição que exiba os dados carregados. Mas esses dois forms seriam muito parecidos, então vamos reaproveitar e usar apenas um.

Mas usando apenas um form para adicionar novos Cds e também atualizar Cds existentes traz uma complicação: quando o usuário clicar em Salvar, vamos disparar uma mesma lógica nos dois casos!

Faremos a lógica **armazena** que serve tanto para adicionar quanto para atualizar:

```
public void armazena (Cd cd) {
      this.daoFactory.beginTransaction();
      this.daoFactory.getCdDao().atualiza(cd);
      this.daoFactory.commit();
}
```

6.6 - Exercício

1) Crie a classe **CdLogic** no pacote **br.com.caelum.lojavirtual.logic**

```
@Component("cd")
@InterceptedBy(DaoInterceptor.class)
public class CdLogic {

      private final DaoFactory daoFactory;

      private Cd cd;
      private List<Cd> cds;

      public CdLogic(DaoFactory daoFactory) {
            this.daoFactory = daoFactory;
      }

      // formulario para adicao
      public void formulario() {
```

```
        }

        // formulario para edicao
        public void editar(Cd cd) {
              // carrega os dados no banco para edicao
              this.cd = this.daoFactory.getCdDao().procura(cd.getId());
        }

        // Adiciona um cd novo ou atualiza se for edicao
        public void armazena (Cd cd) {
              this.daoFactory.beginTransaction();
              this.daoFactory.getCdDao().atualiza(cd);
              this.daoFactory.commit();
        }

        // remove um cd
        public void remove(Cd cd) {
              this.daoFactory.beginTransaction();
              this.daoFactory.getCdDao().remove(cd);
              this.daoFactory.commit();
        }

        // lista todos os cds
        public void lista() {
              this.cds = this.daoFactory.getCdDao().listaTudo();
        }

        // getter pro cd
        public Cd getCd() {
              return cd;
        }

        // getter pra lista de cds
        public List<Cd> getCds() {
              return cds;
        }
    }
```

6.7 - JSPs para cd

Precisamos criar os jsps de formulário e listagem de cds.

O jsp de formulário deve ter campos de texto para cada um dos atributos do CD (Titulo, Artista, etc). Vamos ter um campo hidden para "cd.id" assim nosso formulário servirá também para edição de cds.

```
<input type="hidden" name="cd.id" value="${cd.id}"/>
```

Além disso, todos os campos devem ter o campo value declarado. Na hora de adicionar, o objeto cd estará vazio, então nada será mostrado. Mas na hora de editar, o objeto Cd estará populado e seus dados serão exibidos no form. Por exemplo:

```
<input type="text" name="cd.titulo" value="${cd.titulo}"/>
```

Na listagem, liste todos os cds usando displaytag. Coloque algumas colunas com atributos de cd:

```
  <display:column property="id" />
  <display:column property="titulo" sortable="true" />
  <display:column property="artista" sortable="true" />
  <display:column property="genero" />
```

E mais duas colunas que invocam as ações de remover e editar um cd. Ambos os links devem passar "cd.id", para que a lógica saiba qual cd remover ou editar.

6.8 - Exercício

1) Crie uma pasta **cd** dentro de **web**

2) Crie o arquivo **web/cd/formulario.ok.jsp** através de File -> New -> JSP file.

Dentro do body, coloque:

```
<%@ include file="../menu.jsp" %>

<h1>Cadastro de cd</h1>

<form action="cd.armazena.logic" method="post">
      <input type="hidden" name="cd.id" value="${cd.id}"/>

      Titulo:
      <input type="text" name="cd.titulo" value="${cd.titulo}"/>

      Artista:
      <input type="text" name="cd.artista" value="${cd.artista}"/>

      Genero:
      <input type="text" name="cd.genero" value="${cd.genero}"/>

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

3) Crie o arquivo **web/cd/lista.ok.jsp** através de File -> New -> JSP file.

Adicione o cabeçalho da displaytag no topo do arquivo:

```
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>
```

Dentro do body, liste os cds através da displaytag:

```
<%@ include file="../menu.jsp" %>

<h1>CDs</h1>

<display:table name="${cds}" requestURI="cd.lista.logic">
  <display:column property="id" />
  <display:column property="titulo" sortable="true" />
  <display:column property="artista" sortable="true" />
```

```
  <display:column property="genero" />

  <display:column>
      <a href="cd.editar.logic?cd.id=${cd.id}">editar</a>
  </display:column>

  <display:column>
      <a href="cd.remove.logic?cd.id=${cd.id}">remover</a>
  </display:column>

</display:table>
```

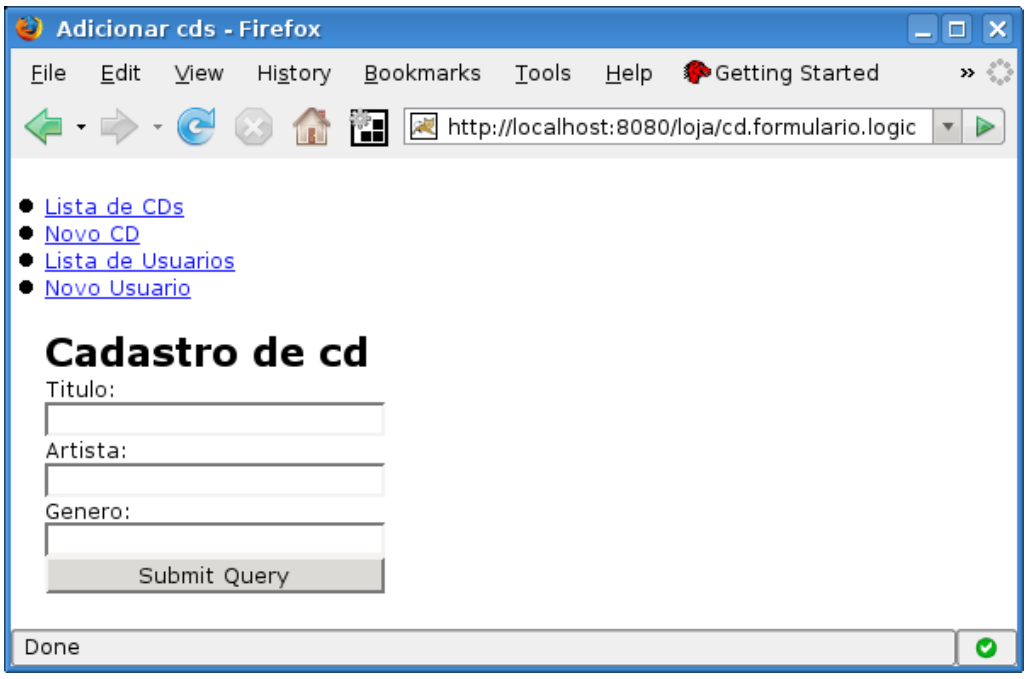
Repare que estamos mostrando os 4 atributos de cd e mais dois links, para remover e editar o cd.

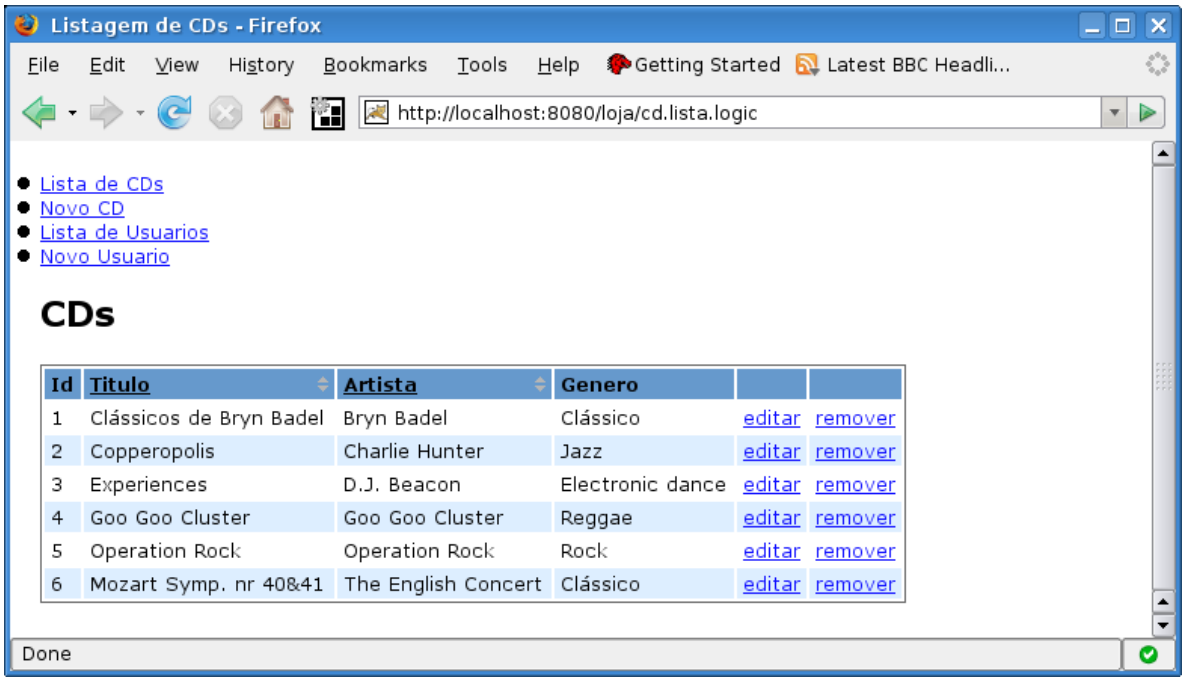
4) Depois que acontecer a adição ou a remoção, queremos que a listagem seja exibida. E queremos que o formulário seja exibido na edição. Adicione ao views.properties:

```
cd.armazena.ok = cd.lista.logic
cd.remove.ok = cd.lista.logic
cd.editar.ok = cd/formulario.ok.jsp
```

5) Reinicie o Tomcat e teste a adição de novos cds acessando:

http://localhost:8080/loja/cd.formulario.logic





6) Adicione mais duas entradas no arquivo menu.jsp:

```
<li><a href="cd.lista.logic">Lista de CDs</a></li>
<li><a href="cd.formulario.logic">Novo CD</a></li>
```

6.9 - MusicaLogic

A classe de lógica para música tem funcionalidades idênticas à CdLogic já feita.

6.10 - Exercício

1) Crie a classe **MusicaLogic** no pacote **br.com.caelum.lojavirtual.logic**

```
@Component("musica")
@InterceptedBy(DaoInterceptor.class)
public class MusicaLogic {

	private final DaoFactory daoFactory;
	private Musica musica;
	private List<Musica> musicas;

	public MusicaLogic(DaoFactory daoFactory) {
		this.daoFactory = daoFactory;
	}

	// formulario para adicao
	public void formulario() {
	}
```

```
	// formulario para edicao
	public void editar(Musica musica) {
		// carrega os dados no banco para edicao
		this.musica = this.daoFactory.getMusicaDao().procura(musica.getId());
	}

	// Adiciona uma musica nova ou atualiza se for edicao
	public void armazena (Musica musica) {
		this.daoFactory.beginTransaction();
		this.daoFactory.getMusicaDao().atualiza(musica);
		this.daoFactory.commit();
	}

	// remove uma musica
	public void remove(Musica musica) {
		this.daoFactory.beginTransaction();
		this.daoFactory.getMusicaDao().remove(musica);
		this.daoFactory.commit();
	}

	// lista todas as musicas
	public void lista() {
		this.musicas = this.daoFactory.getMusicaDao().listaTudo();
	}

	// getter pra musica
	public Musica getMusica() {
		return musica;
	}

	// getter pra lista de musicas
	public List<Musica> getMusicas() {
		return musicas;
	}
}
```

6.11 - JSPs para Música

Criaremos dois JSPs, assim como para os Cds, um formulário e uma listagem.

O formulário terá todos os campos da música para cadastro, análogo ao form de cd. A grande diferença é em como tratar o relacionamento.

Como temos um relacionamento ManyToOne entre Musica e Cd, quando formos cadastrar uma Musica precisamos indicar a qual Cd ela pertence. Teremos, então, uma opção no formulário de Musica que permita a escolha do Cd através de um select do html.

A complexidade neste form está em criar o select dinamicamente, a partir da lista de todos os Cds do banco. Para cada Cd do banco, criamos um <option> dentro do <select>.

De alguma forma, precisamos que a lista de Cds do sistema esteja disponível para o JSP. A melhor forma é acrescentar algum getter na nossa MusicaLogic que devolve uma lista de Cds, desta forma:

```
// metodo usado para popular o select
public List<Cd> getListaCds() {
      return this.daoFactory.getCdDao().listaTudo();
}
```

No JSP, o select fica:

```
<select name="musica.cd.id">
  <c:forEach var="cd" items="${listaCds}">
      <option value="${cd.id}">${cd.titulo}</option>
  </c:forEach>
</select>
```

E, para suportar edição, precisamos verificar qual opção estava selecionada anteriormente (se estava). O JSP final fica:

```
<select name="musica.cd.id">
  <c:forEach var="cd" items="${listaCds}">

      <option value="${cd.id}"
            <c:if test="${musica.cd.id == cd.id}">selected="true"</c:if>
      >
            ${cd.titulo}
      </option>
  </c:forEach>
</select>
```

Já a listagem será semelhante à dos Cds.

6.12 - Exercício

1) Crie um método getListaCds na sua classe MusicaLogic:

```
// metodo usado para popular o select
public List<Cd> getListaCds() {
      return this.daoFactory.getCdDao().listaTudo();
}
```

2) Crie uma pasta **musica** dentro de **web**

3) Crie o arquivo **web/musica/formulario.ok.jsp** através de File -> New -> JSP file.

Adicione o cabeçalho da jstl no topo do arquivo:

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

Dentro do body, coloque:

```
<%@ include file="../menu.jsp" %>

<h1>Cadastro de musica</h1>

<form action="musica.armazena.logic" method="post">
      <input type="hidden" name="musica.id" value="${musica.id}"/>
```

```
      CD:
      <select name="musica.cd.id">
        <c:forEach var="cd" items="${listaCds}">

            <option value="${cd.id}"
                  <c:if test="${musica.cd.id == cd.id}">selected="true"</c:if>
            >
                  ${cd.titulo}
            </option>
        </c:forEach>
      </select>

      Titulo
      <input type="text" name="musica.titulo" value="${musica.titulo}"/>

      Preco
      <input type="text" name="musica.preco" value="${musica.preco}"/>

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

4) Crie o arquivo **web/musica/lista.ok.jsp** através de File -> New -> JSP file.

Adicione o cabeçalho da displaytag no topo do arquivo:

```
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>
```

Dentro do body, liste os cds através da displaytag:

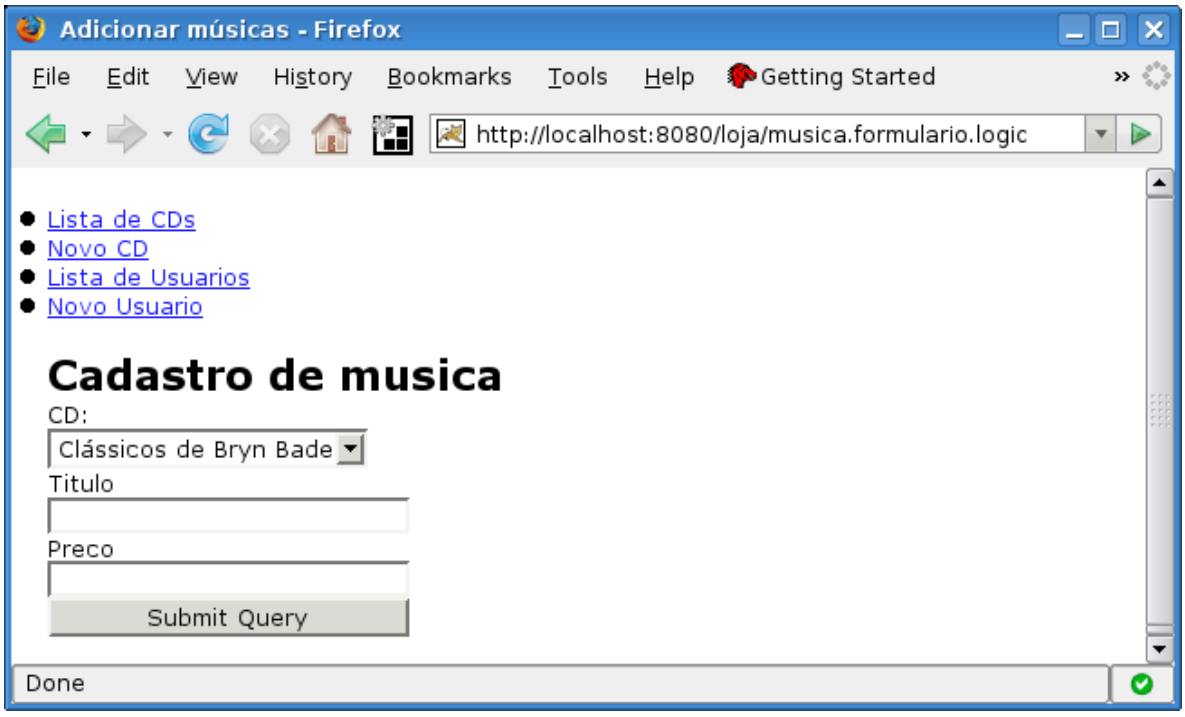
```
<%@ include file="../menu.jsp" %>

<h1>Musicas</h1>

<display:table id="musica" name="${musicas}" requestURI="musica.lista.logic">
  <display:column property="id" />
  <display:column property="cd.titulo" sortable="true" title="CD" />
  <display:column property="titulo" sortable="true" />
  <display:column property="preco" sortable="true" />

  <display:column>
      <a href="musica.editar.logic?musica.id=${musica.id}">editar</a>
  </display:column>

  <display:column>
      <a href="musica.remove.logic?musica.id=${musica.id}">remover</a>
  </display:column>

</display:table>
```

5) Adicione ao views.properties:

```
musica.armazena.ok = musica.lista.logic
musica.remove.ok = musica.lista.logic
musica.editar.ok = musica/formulario.ok.jsp
```

6) Reinicie o Tomcat e teste a adição de novas músicas acessando:

http://localhost:8080/loja/musica.formulario.logic



7) Adicione mais duas entradas no arquivo menu.jsp:

```
<li><a href="musica.lista.logic">Lista de Musicas</a></li>
<li><a href="musica.formulario.logic">Nova Musica</a></li>
```

6.13 - Desafio - Exercicios opcionais

1) Crie uma **musica.listaPorCd.logic** que recebe um **cd.id** e lista apenas as músicas de um determinado cd. Adicione um link na tabela de **web/cd/lista.ok.jsp** para essa lógica.

capítulo 7

Autenticação e autorização

"Hell is other people."
Jean-Paul Sartre -

Neste capitulo iremos:
- criar um sistema de login
- utilizar um interceptador para verificar a permissao do usuario

7.1 - Login

Agora que já temos um sistema de cadastro de usuários e algumas páginas da área de administração de nosso site prontas, vamos criar um sistema de login.

O primeiro passo é criar uma página de entrada para o usuário digitar login e senha. Quando o form for enviado, devemos então verificar em uma lógica se o usuário está correto ou não.

Se o usuário estiver correto, salvamos o usuário na sessão e redirecionamos para alguma página de entrada.

7.2 - Exercício - Formulário de login

1) Crie uma pasta **admin** dentro de **web**

2) Crie o arquivo **web/admin/login.ok.jsp** através de File -> New -> JSP file.

Adicione o cabeçalho da jstl no topo do arquivo:

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

E dentro da tag body, faça:

```
<h1>Loja Virtual - Login </h1>

<form action="admin.efetuaLogin.logic" method="post">

      Login:
      <input type="text" name="usuario.login">

      Senha:
      <input type="password" name="usuario.senha">

      <input type="submit">
```

```
</form>
```

7.3 - UsuarioDao

Para verificarmos se o usuário pode logar ou não no nosso sistema, precisamos buscar no banco de dados o Usuario passado. Precisamos então de algum método no nosso Dao que verifique se determinado Usuario existe ou não no banco de dados.

Mas será que este método é interessante para todas as entidades do sistema? Provavelmente não. Por isso, ao invés de adicionarmos o método em Dao<T>, criaremos uma classe UsuarioDao.

Uma prática muito comum quando se usa o Dao genérico é criar subclasses de Dao com Daos mais específicos, com métodos específicos de certa entidade.

Podemos criar então um UsuarioDao que **estende** Dao<Usuario> e acrescenta mais um método: existeUnico, que recebe um Usuario com login e senha e devolve um Usuario do banco de dados ou null se não existir.

```
public class UsuarioDao extends Dao<Usuario> {

	UsuarioDao(Session session) {
		super(session, Usuario.class);
	}

	public Usuario existeUnico (Usuario u) {
		// logica aqui...
	}
}
```

Precisaremos fazer apenas alguns ajustes nas nossas classes. Primeiro, a classe Dao precisará expor sua session para as classes filhas. Criaremos um getSession **protected** na classe Dao<T>:

```
	protected Session getSession() {
		return session;
	}
```

Além disso, na classe DaoFactory, o método getUsuarioDao precisa ser **modificado** para devolver um dao específico (e não o genérico):

```
	public UsuarioDao getUsuarioDao() {
		return new UsuarioDao(this.session);
	}
```

Mas, voltando ao nosso método existeUnico, como implementá-lo? Com HQL, é muito simples fazer esse tipo de verificação:

```
	public Usuario existeUnico (Usuario u) {
		String hql = "select u from Usuario as u where u.login = :login and
u.senha = :senha ";
		Query query = getSession().createQuery(hql);
		query.setParameter("login", u.getLogin());
```

```
            query.setParameter("senha", u.getSenha());
            return (Usuario) query.uniqueResult();
      }
```

7.4 - Exercício

1) Acrescente um getSession **protected** na classe Dao<T>.
Dica: no eclipse, digite *get* e Ctrl+Espaço, ele sugere a criação do getter.

```
      protected Session getSession() {
            return session;
      }
```

2) Crie uma classe chamada **UsuarioDao** no pacote **br.com.caelum.lojavirtual.dao**.
Esta classe deve estender **Dao<Usuario>** e implementar um método **existeUnico**, como segue:

```
public class UsuarioDao extends Dao<Usuario> {

      UsuarioDao(Session session) {
            super(session, Usuario.class);
      }

      public Usuario existeUnico (Usuario u) {
            String hql = "select u from Usuario as u where u.login = :login and
u.senha = :senha ";
            Query query = getSession().createQuery(hql);
            query.setParameter("login", u.getLogin());
            query.setParameter("senha", u.getSenha());
            return (Usuario) query.uniqueResult();
      }
}
```

3) **Modifique** o método getUsuarioDao na classe DaoFactory para devolver uma instância do dao específico:

```
      public UsuarioDao getUsuarioDao() {
            return new UsuarioDao(this.session);
      }
```

7.5 - Lógica

Nosso método que verifica se o usuário pode logar ou não tem que verificar isso usando o existeUnico do Dao. Se o retorno do método for um Usuario, quer dizer que ele pode logar, se for null, não pode.

Mas quando escrevemos um sistema de login, precisamos salvar na sessão o usuário logado. Mas como salvar um Usuario na sessão em uma lógica do Vraptor se não temos acesso a HttpServletRequest ou HttpSession? Na verdade, nem precisamos disso.

Com uma anotação, conseguimos dizer para o vraptor que determinado dado de nossa lógica deve ser salvo no escopo da sessão. Criamos um

atributo e um getter com essa anotação:

```
private Usuario usuario;

@Out(scope=ScopeType.SESSION)
public Usuario getUsuario() {
	return usuario;
}
```

Assim, nosso método efetuaLogin pode ser algo parecido com isso:

```
public void efetuaLogin(Usuario usuario) {
	UsuarioDao dao = this.daoFactory.getUsuarioDao();
	this.usuario = dao.existeUnico(usuario);

	if (this.usuario != null) {
		// ... pode logar ...
	} else {
		// ... nao pode logar ...
	}
}
```

Request, Response, etc

O vraptor encapsula totalmente a ideia de HttpServletRequest/Response para que voce não precisa tratar coisas do framework de servlets. Mas podem existir casos onde voce precisa acessar request/response diretamente (não é nosso caso aqui), e o vraptor permite isso.

Veja mais em: http://vraptor.org/injection.html

7.6 - Retorno condicional com Vraptor

Ainda falta um detalhe no nosso sistema de login: quando o usuario logar com sucesso, queremo redirecioná-lo para um lugar, e se ele não puder logar, para outro lugar. Ou seja, queremos que o redirecionamento seja **condicional**.

Até agora, havíamos visto apenas lógicas sem retorno, void. Quando era assim, vimos que a página a ser exibida era sempre **componente/logica.*ok*.jsp**. O sufixo "ok" indica uma lógica que foi executada com sucesso.

Podemos, porém, controlar a execução da logica devolvendo uma String, que será usada como sufixo no jsp no lugar de "ok". Por exemplo, uma logica que devolva uma String "sucesso" será redirecionada para **componente/logica.*sucesso*.jsp**.

Nós podemos usar este recurso no método efetuaLogin acima:

```
public String efetuaLogin(Usuario usuario) {
	UsuarioDao dao = this.daoFactory.getUsuarioDao();
	this.usuario = dao.existeUnico(usuario);

	if (this.usuario != null) {
```

```
            return "ok";
        } else {
            return "invalid";
        }
    }
```

Quando login for bem-sucedido, ele procurará por **admin/efetuaLogin.ok.jsp** e quando o login estiver inválido, ele vai para **admin/efetuaLogin.invalid.jsp**

Podemos inclusive sobrescrever essas páginas no views.properties. Por exemplo, para redirecionar o usuário para o form em caso de erro de login, fazemos:

```
admin.efetuaLogin.invalid = admin.login.logic
```

7.7 - Logout

Ok, nosso usuário consegue se logar no sistema. Mas como faríamos uma lógica para logout? Com o vraptor, isso é muito simples.

```
public void logout() {
    this.usuario = null;
}
```

7.8 - Exercício

1) Crie a classe **AdminLogic** no pacote **br.com.caelum.lojavirtual.logic**:

```
@Component("admin")
@InterceptedBy(DaoInterceptor.class)
public class AdminLogic {

    private final DaoFactory daoFactory;

    private Usuario usuario;

    public AdminLogic(DaoFactory daoFactory) {
        this.daoFactory = daoFactory;
    }

    public void login() {
    }

    public String efetuaLogin(Usuario usuario) {
        UsuarioDao dao = this.daoFactory.getUsuarioDao();
        this.usuario = dao.existeUnico(usuario);

        if (this.usuario != null) {
            return "ok";
        } else {
            return "invalid";
        }
    }

    public void logout() {
```

```
            this.usuario = null;
      }

      @Out(scope=ScopeType.SESSION)
      public Usuario getUsuario() {
            return usuario;
      }

}
```

2) Configure o sistema para:
- mostrar o form novamente em caso de erro de login
- mostrar a lista de cds assim que o usuario logar
- mostrar o form de login depois do logout

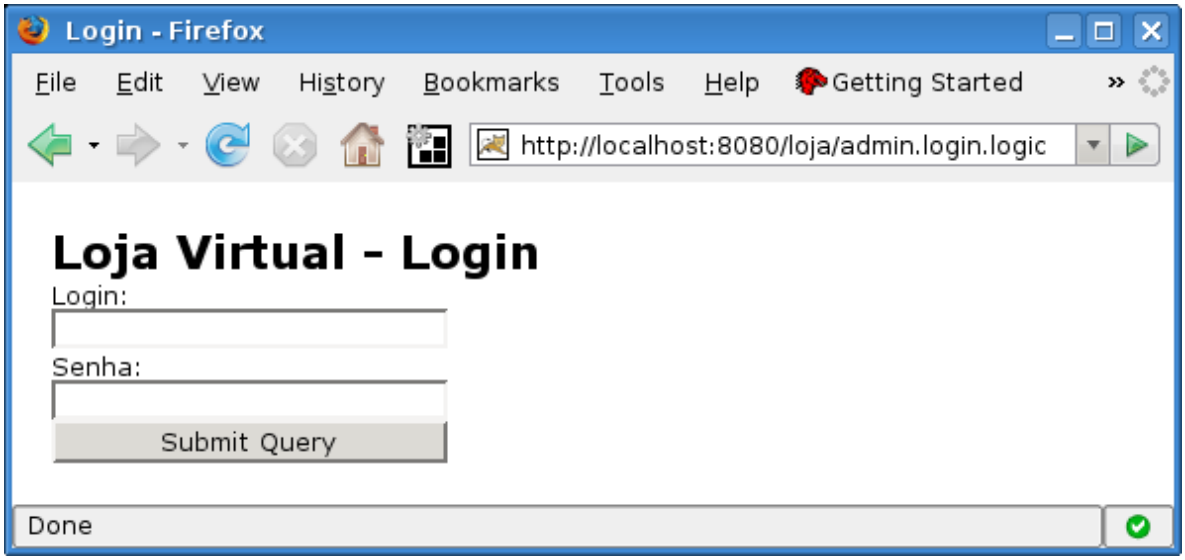
Adicione ao views.properties:

```
admin.efetuaLogin.invalid = admin.login.logic
admin.efetuaLogin.ok = cd.lista.logic
admin.logout.ok = admin.login.logic
```

3) Adicione um link de logout ao menu.jsp:

```
<li><a href="admin.logout.logic">Logout</a></li>
```

4) Teste o sistema de login



7.9 - Interceptador para autorização

Agora vamos para a parte de autorização.

Para toda requisição que requer que o usuário esteja logado vamos definir um interceptador que roda antes de tal execução e verifica se ele realmente está logado ou não.

Esse conceito de rodar algo antes ou depois de uma cadeia de lógica de negócios de forma a fornecer suporte a algo novo lembra programação

orientada a aspectos mas não é!

Criaremos um AutorizadorInterceptor que implementa Interceptor e no seu método precisamos fazer a verificação desejada:

```
public class AutorizadorInterceptor implements Interceptor {
	public void intercept(LogicFlow flow) throws LogicException, ViewException {

	}
}
```

Começamos injetando a variável usuario do escopo de sessão, mas não desejamos fazer isso obrigatoriamente caso contrário o Vraptor joga uma exception que não nos interessa. Portanto marcaremos a tag In como opcional:

```
	@In(scope=ScopeType.SESSION, required=false)
	private Usuario usuario;
```

E agora implementamos a verificação. Se a variavel estiver null, redirecionamos para o form de login e não executamos a logica. Caso não seja null, entao o usuario está logado, e prosseguimos a execucao da logica:

```
	if (this.usuario == null) {
		try {
			flow.getLogicRequest().getResponse()
				.sendRedirect("admin.login.logic");
		} catch (IOException e) {
			throw new LogicException(e);
		}
	} else {
		flow.execute();
	}
```

Repare que estamos usando o sendRedirect de HttpServletResponse, que pode ser acessado através da chamada a getLogicRequest e get Response.

7.10 - Exercício

1) Crie a classe AutorizadorInterceptor no pacote br.com.caelum.lojavirtual.logic

```
public class AutorizadorInterceptor implements Interceptor {

	@In(scope=ScopeType.SESSION, required=false)
	private Usuario usuario;

	public void intercept(LogicFlow flow) throws LogicException, ViewException {

		if (this.usuario == null) {
			try {
				flow.getLogicRequest().getResponse()
					.sendRedirect("admin.login.logic");
			} catch (IOException e) {
				throw new LogicException(e);
			}
```

```
            } else {
                  flow.execute();
            }

      }
}
```

7.11 - Autorizando lógicas

Nosso módulo Autorizador é um Interceptor do vraptor. Isso quer dizer que, para proteger certas logicas de serem executadas, basta pedir que o Componente se interceptado usando @InterceptedBy.

Como nossos componentes já tinham um Interceptador, o DaoInterceptor, vamos modificar a chamada ao @InterceptedBy para incluir o AutorizadorInterceptor também.

Por exemplo:

```
@Component("cd")
@InterceptedBy({AutorizadorInterceptor.class,DaoInterceptor.class})
public class CdLogic {
```

7.12 - Exercício

1) Altere suas classes **CdLogic, MusicaLogic** e **UsuarioLogic** para serem interceptadas pelo AutorizadorInterceptor:

```
@InterceptedBy({AutorizadorInterceptor.class,DaoInterceptor.class})
```

Cuidado: a AdminLogin não pode ser interceptada pelo Autorizador, isso não faz sentido (o usuário precisa estar logado para poder logar??)

2) Teste seu sistema de login e autorização.

3) Adicione um link de logout ao menu.jsp:

```
<li><a href="admin.logout.logic">Logout</a></li>
```

7.13 - Exercícios opcionais

1) Uma forma simples de mostrar uma mensagem de erro quando o usuário digitar login ou senha inválidos é verificar no login.ok.jsp se o parâmetro usuario.login foi enviado. Isso indica que o form foi enviado mas não aceito, ou seja, os dados estavam errados. Algo assim:

```
<c:if test="${not empty param['usuario.login']}">
      Login/senha errados
</c:if>
```

Uma outra forma poderia ser um atributo "loginInvalido" na AdminLogic que fosse outjetado para o JSP.

capítulo 8

Validação com Hibernate Validator

"Que o teu corpo não seja a primeira cova do teu esqueleto."
Jean Giradoux -

Aqui iremos:
- utilizar o Hibernate Validator como framework de validação
- integrar o Hibernate Validator com o Vraptor
- implementar validação no nosso modelo
- tratar erros na camada de visualização

8.1 - O Hibernate validator

É um framework muito poderoso desenvolvido pelo pessoal do Hibernate para auxiliar na validação de dados.

A integração com o vraptor permite que usemos as mesmas validações usadas pelo hibernate no banco para validar nossas lógicas de negocio.

Para integrar com o vraptor, precisamos registrar um plugin opcional através do vraptor.xml:

```
<vraptor>
   <plugin>org.vraptor.plugin.hibernate.HibernateValidatorPlugin</plugin>
</vraptor>
```

8.2 - Anotações nos beans

As validações são feitas através de anotações do Hibernate validator nos atributos dos beans. Algumas anotações possíveis:

@NotNull - não permite null
@NotEmpty - não permite null nem em branco
@Length – define comprimento máximo e mínimo
@Email – verifica se é um email valido

E muitas outras.

Todas as anotações podem receber um campo message com a mensagem a ser exibida.

Por exemplo, para validar que o título do CD não pode ser deixado em branco, use:

```
@NotEmpty(message="O título não pode estar vazio!")
private String titulo;
```

8.3 - Validando no vraptor

Para acionarmos a validação com hibernate no vraptor, adicionamos a anotação @Validate na lógica a ser validada. Ela recebe um argumento parameters com os parâmetros que desejamos validar.

Por exemplo, na CdLogic, podemos validar a lógica armazena:

```
@Validate(params={"cd"})
public void armazena (Cd cd) {
```

Mas o que acontece se ocorrer erro de validação? Simples: o vraptor não executa sua lógica e tenta encontrar a view **componente.logica.invalid**

Para fazer o formulário ser exibido novamente quando ocorrer erro de validação, podemos adicionar no views.properties:

```
cd.armazena.invalid = cd/formulario.ok.jsp
```

E, no formulário, podemos listar os erros que aconteceram com:

```
<ul id="erros">
      <c:forEach var="error" items="${errors.iterator}">
          <li>${error.key}</li>
    </c:forEach>
</ul>
```

8.4 - Exercícios

Vamos adicionar validação nos campos título de Cd e Musica.

1) Abra as classes Cd e Musica e anote seu atributo titulo com:

```
@NotEmpty(message="O titulo não pode estar vazio!")
private String titulo;
```

2) No componente CdLogic anote o método armazena com:

```
@Validate(params={"cd"})
public void armazena (Cd cd) {
```

e no componente MusicaLogic anote o método armazena com:

```
@Validate(params={"musica"})
public void armazena (Musica musica) {
```

3) Crie um arquivo **vraptor.xml** dentro da pasta src com a configuração do plugin do HibernateValidator:

```
<vraptor>
   <plugin>org.vraptor.plugin.hibernate.HibernateValidatorPlugin</plugin>
</vraptor>
```

4) Adicione no views.properties:

```
cd.armazena.invalid = cd/formulario.ok.jsp
musica.armazena.invalid = musica/formulario.ok.jsp
```

5) Abra a pagina **web/cd/formulario.ok.jsp** e adicione as mensagens de erro:

```
<ul id="erros">
      <c:forEach var="error" items="${errors.iterator}">
          <li>${error.key}</li>
    </c:forEach>
</ul>
```

Faça a mesma coisa para **web/musica/formulario.ok.jsp**

6) Teste seu sistema.

8.5 - Exercícios opcionais

1) Faça a validação na Venda do cliente. Use @NotEmpty e @Email no campo de email.

2) Ao invés de deixar as mensagens direto na anotação, podemos colocar apenas uma chave e deixar as mensagens em um arquivo tipo **message.properties**.

Aí, para carregar a mensagem baseado na chave, basta usar a JSTL fmt.

8.6 - Para saber mais: validações personalizadas

O Hibernate Validator possui uma forma muito simples mas robusta de fazer validações personalizadas. Ele permite que você crie sua própria anotação de validação e uma classe que implemente um método de validação.

Para ver como funciona esse mecanismo, veja a documentação eu vem junto com o hibernate-validator.zip (dentro da pasta doc).

8.7 - Para saber mais: validação sem Hibernate Validator

O vraptor suporta, além do hibernate validator, a validação através de métodos normais que nos escrevemos.

Veja mais sobre isso aqui:

http://vraptor.org/validation.html

capítulo 9

A loja virtual

"Odiar as pessoas é como atear fogo na casa a fim de se livrar de um rato."
H. E. Fosdick -

Aqui iremos:
- criar a interface básica para o cliente comprar musicas

9.1 - Idéia geral

A entrada de nossa loja virtual será uma lista com vários Cds e suas respectivas músicas. O usuário escolhe o que desejar comprar e adicione ao carrinho.

Mas nosso processo de compra não será tão tradicional. Usaremos recursos interessantes de Ajax e Efeitos visuais para criar uma interface drag-and-drop (arrastar e soltar) de compra. O usuário escolhe a música e a arrasta para o carrinho de compras.

No final, o usuário clica em finalizar, e uma janela com um formulário de encerramento de pedido é mostrada. O usuário preenche os dados, clica em finalizar e uma Venda é salva no banco de dados.

9.2 - Pensando na página - @OneToMany

Na nossa página, vamos listar todos os Cds do sistema e, junto com cada um, vamos listar suas músicas. Ou seja, teremos dois fors encadeados percorrendo as músicas do sistema (ou c:forEach da jstl).

Para facilitar (e muito!) o nosso trabalho, quando estivermos percorrendo os Cds e quisermos pegar as músicas desse cd, faremos:

```
Set<Musica> musicas = cd.getMusicas();
```

Mas não temos um método getMusicas na classe Cd. Normalmente, quando fazemos um relacionamento @ManyToOne no hibernate, não mapeamos o outro lado para simplificar o código. Porém, no nosso caso, será bastante interessante ter um **relacionamento bidirecional**, onde posso acessar as musicas de um cd e o cd de um musica.

No Hibernate, para mapearmos o outro lado de um relacionamento ManyToOne, usamos a anotação @OneToMany. Só não podemos esquecer de dizer para o hibernate que esse é o mesmo relacionamento que já foi mapeado do outro lado com ManyToOne. Fazemos isso com a opção

mappedBy.

Na classe Cd, então, teremos um atributo musicas mapeado da seguinte forma:

```
@OneToMany(mappedBy="cd")
private List<Musica> musicas;
```

Cuidado com relacionamerto bidirecional

Daqui a pouco voce ira entender porque o relacionamento bidirecional facilitara nosso sistema. Mas, como uma regra geral, evite fazer relacionamentos bidirecionais quando não tiver uma boa justificativa.

Relacionamentos bidirecionais entre objetos trazem um maior acoplamento e uma maior complexidade, que na maioria das vezes é desnecessária. Manter o estado do sistema consistente com relacionamentos bidirecionais é uma árdua tarefa!

9.3 - Exercício

1) Crie um atributo **musicas** na sua classe **Cd**, da seguinte forma:

```
@OneToMany(mappedBy="cd")
private List<Musica> musicas;
```

2) Gere get e set para esse atributo

9.4 - Página inicial

Para gerenciar nossa loja virtual, criaremos um componente LojaLogic. Ele terá uma lógica **inicio** que apenas exibirá a página de entrada de nosso site.

E, como nossa página irá listar os Cds, precisamos ter uma forma de obter esses Cds. Por isso criamos um método getCds que devolve todos os Cds do sistema.

Vamos aproveitar para criar o JSP e já configurar a JSTL, os JavaScripts para Ajax e o CSS de estilo.

9.5 - Exercício

1) Crie uma classe LojaLogic no pacote br.com.caelum.lojavirtual.logic

```
@Component("loja")
@InterceptedBy({DaoInterceptor.class})
public class LojaLogic {

	private DaoFactory daoFactory;

	public LojaLogic(DaoFactory daoFactory) {
		this.daoFactory = daoFactory;
	}
```

```
	public void inicio() {
	}

	public List<Cd> getCds() {
		return this.daoFactory.getCdDao().listaTudo();
	}
}
```

2) Crie uma pasta **loja** dentro de **web**

3) Crie o arquivo **web/loja/inicio.ok.jsp** através de File -> New -> JSP file.

a) No topo do arquivo, adicione a JSTL:

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

b) **Dentro** do <head>, vamos carregar o JavaScript que será usado mais pra frente para Ajax:

```
	<script src="javascripts/efeitos.js"></script>
```

c) **Dentro** do <head>, vamos carregar o CSS:

```
	<link rel="stylesheet" type="text/css" href="css/style.css"/>
```

d) Coloque a class=loja no <body>

```
	<body class="loja">
```

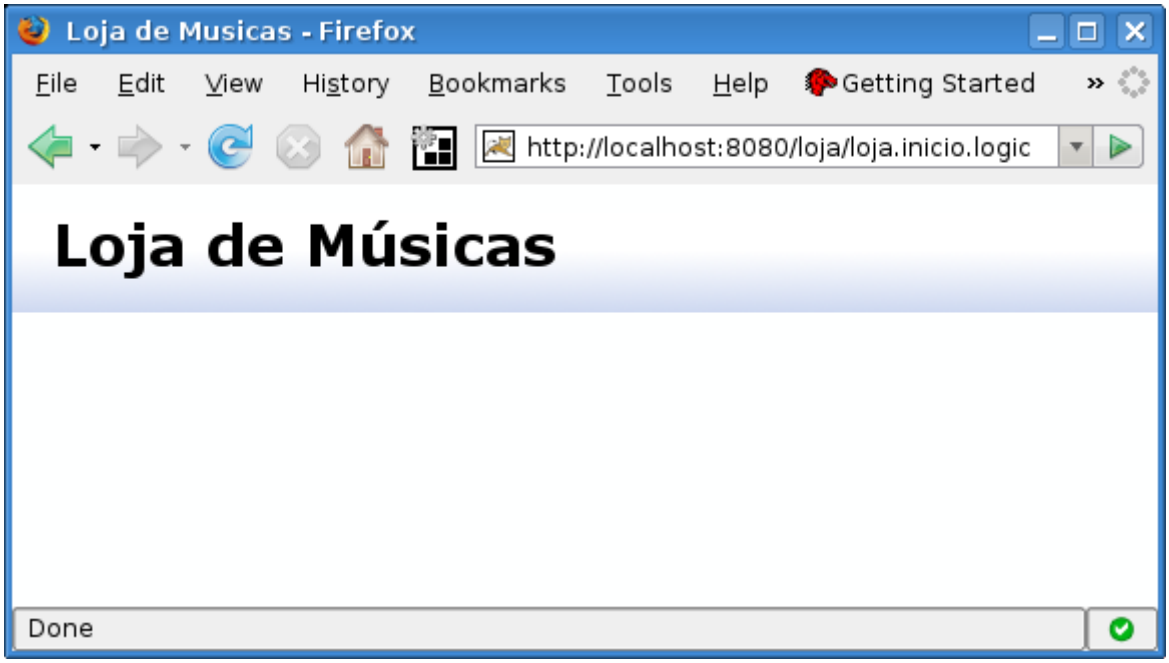
e) Coloque um título dentro do <body> para testarmos:

```
	<h1>Loja de Musicas</h1>
```

O JSP, no final, deve estar parecido com isso:

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
<head>

	<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
	<title>Loja de Musicas</title>

	<link rel="stylesheet" type="text/css" href="css/style.css"/>
	<script src="javascripts/efeitos.js"></script>

</head>
<body class="loja">

	<h1>Loja de Musicas</h1>

</body>
</html>
```

4) Reinicie o Tomcat e acesse http://localhost:8080/loja/loja.inicio.logic



9.6 - Listar Cds e Musicas

Vamos listar todos os Cds e suas respectivas músicas. Como esta página será a usada com Ajax, precisamos adicionar um pouco de html a mais, e alguma estrutura nas tags.

As classes e divs usadas agora serão de *extrema importância* depois para o Ajax, por isso, *não mude* esses nomes.

Vamos percorrer todos os cds com um forEach. Para cada cd, criamos um cujo id é "cd-num", onde num é o id desse cd no banco. Dentro desse div, temos 4 coisas: imagem da capa do cd (cadastrada no banco), título do cd, artista e as musicas.

Para as musicas, pegamos todas as musicas de um cd com ${cd.musicas} (que chama aquele nosso método de relacionamento bidirecional). Usando um segundo forEach, listamos as musicas com s.

Veja o código no exercício abaixo. Lembre de manter os ids e classes do html. E lembre que, como estamos usando um CSS já pronto, a página já terá um visual interessante definido.

9.7 - Exercício

1) Adicione os forEach para listar cds e musicas, dentro da tag <body> do seu arquivo inicio.ok.jsp

```
<div id="cds">
<c:forEach var="cd" items="${cds}">

      <div id="cd-${cd.id}" class="cd">
            <span class="titulo">${cd.titulo}</span>
            <span class="artista">${cd.artista}</span>
            <hr/>
            <ol>
                  <c:forEach var="musica" items="${cd.musicas}">
                        <li class="musica" id="musica-${musica.id}">
                              ${musica.titulo} - ${musica.preco}
                        </li>
                  </c:forEach>
            </ol>
      </div>

</c:forEach>
</div>
```

2) Acesse a página para ver o resultado

OrderBy

Você pode definir a ordem em que as músicas serão obtidas definindo a anotação OrderBy no relacionamento OneToMany. Para ordenar por titulo, por

exemplo:

```
@OneToMany(mappedBy="cd")
@OrderBy("titulo")
private List<Musica> musicas;
```

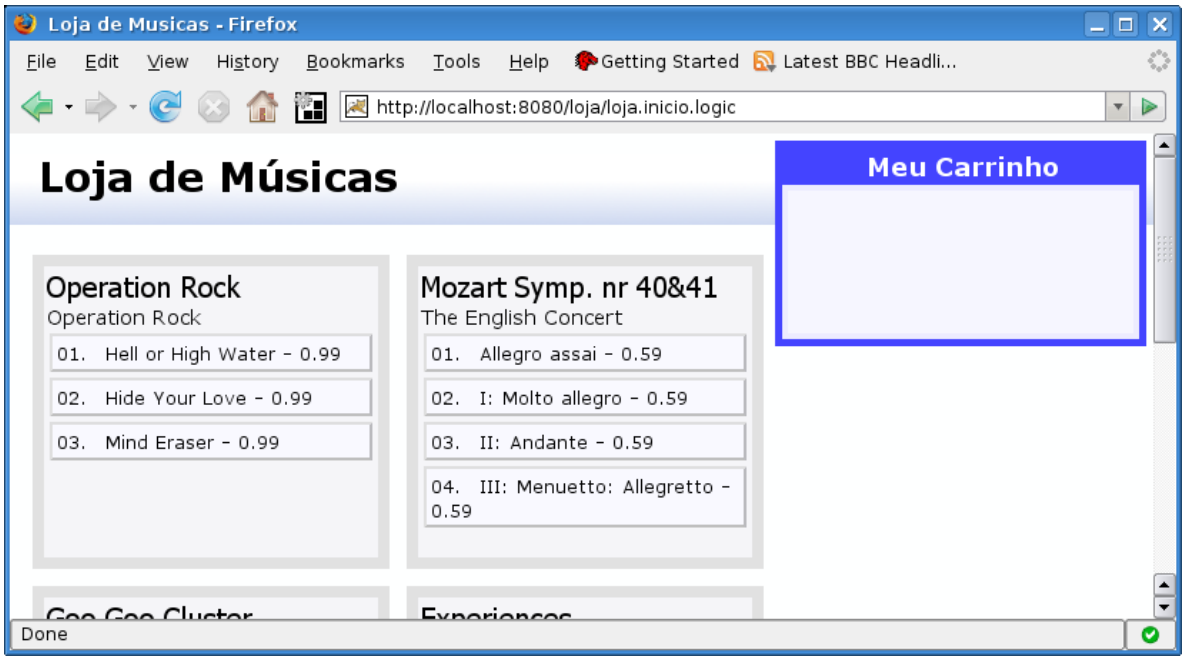
9.8 - Exercicio - Futuro carrinho de compras

Vamos definir mais uma seção no nosso JSP onde será o carrinho de compras. Ainda não vamos escrever a funcionalidade com Ajax, mas já teremos a área resevada na nossa pagina

1) Depois dos <forEach>s e de fechar o div de id=cds, logo antes de fechar </body>, adicione a área do carrinho de compras:

```
<div id="carrinho">
      <h2>Meu Carrinho</h2>
      <div id="compras">

      </div>
</div>
```

2) Acesse o site e veja como ele está ficando:



capítulo 10

Ajax e efeitos visuais

"As idéias simples só estão ao alcance de espíritos complexos."
Rémy de Gourmont -

Este capitulo mostra:
- o que é AJAX
- como usar o JQuery
- utilizar efeitos de drag-n-drop
- criar um carrinho de compras na sessão
- atualizar a pagina com ajax

10.1 - AJAX- Asynchronous JavaScript and XML

É o nome da tecnologia que nos permite atualizar uma página sem o seu refresh. Na verdade o mecanismo é muito simples. De acordo com alguma ação um javascript envia uma requisição ao servidor como se fosse em background. Na resposta dessa requisição vem um XML que o javascript processa e modifica a página segundo essa resposta.

É o efeito que tanto ocorre no gmail e google maps.

Há várias ferramentas para se trabalhar com Ajax em Java. DWR e Google Web Toolkit são exemplos famosos de ferramentas que te auxiliam na criação de sistemas que usam AJAX.

Nós utilizaremos o Vraptor no servidor para nos ajudar com as requisições Ajax.

No cliente, usaremos a biblioteca **JQuery** que é totalmente escrita em JavaScript, portanto independente de linguagem do servidor.

Há vários frameworks javascript disponíveis no mercado além do jquery. Apenas para citar os mais famosos: Prototype/Script.aculo.us, Yahoo User Interface (YUI), Dojo. Usaremos o Jquery devido a sua extrema simplicidade de uso, muito boa para quem não domina javascript mas quer usar recursos de Ajax.

10.2 - Um pouco de jquery

A biblioteca jquery é baseada em um conceito de encadeamento (**chaining**). As chamdas de seus métodos são encadeadas uma após a outra, o que cria código muito simples de serem lidos.

O ponto de partida do jquery é a funcão $ que seleciona elementos

DOM a partir de seletores CSS. Para selecionar um nó com id "teste" for exemplo, fazemos:

$('#teste')

ou para selecionar os elementos que possuem a classe musica:

$('.musica')

Seletores

O jquery suporta Xpath e CSS 3, com seletores avançadíssimos (além dos clássicos id e class). Veja mais aqui: http://docs.jquery.com/Selectors

Depois de selecionar o(s) elemento(s) desejado(s), podemos chamar métodos para as mais variadas coisas. O HelloWorld do jquery mostra como exibir e esconder um div especifico:

$('#meuDiv').show()
$('#meuDiv').hide()

Documentação

http://www.visualjquery.com/
http://docs.jquery.com/

10.3 - Draggables and Droppables

Antes de ver as requisições ajax propriamente ditas, vamos configurar os efeitos visuais de arrastar e soltar das musicas para o carrinho.

Usando jquery (e ao plugin Interface dele) precisamos declarar quais elementos serão usados para arrastar (Draggables) e onde poderemos solta-los (Droppables).

No nosso caso, queremos fazer as musicas (todos os elementos que têm a classe "musica") como Draggables e o carrinho como Droppables. Com jquery é simples assim:

```
$('.musica').Draggable();

$('#carrinho').Droppable({ accept:  'musica' });
```

O primeiro código seleciona todos os elementos com classe musica e os faz Draggable. O segundo, transforma o div carrinho em Droppable, onde podemos soltar elementos de classe 'musica' (o argumento accept indica a classe de elementos que ele aceita).

Podemos ainda declarar mais parâmetros, nos dois casos:

```
$('.musica').Draggable({
      ghosting:   true,
      opacity:    0.7,
      zIndex:     10,
```

```
            revert:     true
      });
```

Explicando:
- ghosting: cria um segundo elemento para representar o movimento
- opacity: opacidade do elemento fantasma
- zIndex: indica a camada onde o elemento está
- revert: faz com que o elemento volte depois de solto

```
      $('#carrinho').Droppable({
            accept:     'musica',
            onDrop:     atualizaCarrinho
      });
```

Explicando:
- accept: classe dos elementos que são aceitos
- onDrop: funcao a ser chamada quando algo for solto (drop) no elemento

Para colocarmos isso na página, precisamos ainda definir que esses comando só serão disparados quando o documento carregar (na verdade, quando a árvore DOM estiver pronta). Isso evita possíveis erros que aconteceriam se o navegador ainda não tivesse rendereizado os componentes requisitados.

Basta colocar essas 2 chamadas em uma function e passá-la para a function principal $(). Com functions "anonimas" (closures), podemos fazer:

```
$(function(){
      $('.musica').Draggable({
            ghosting:   true,
            opacity:    0.7,
            zIndex:     10,
            revert:     true
      });

      $('#carrinho').Droppable({
            accept:     'musica',
            onDrop:     atualizaCarrinho
      });
});
```

E só falta declarar a função atualizaCarrinho que será chamada quando o produto for solto em cima do carrinho. Inicialmente, só para testes:

```
function atualizaCarrinho(drag) {
      alert(drag.id);
}
```

Isso irá mostrar um alert com o id da musica que foi solta no carrinho (o parâmetro drag contem o elemento que foi solto).

10.4 - Exercício

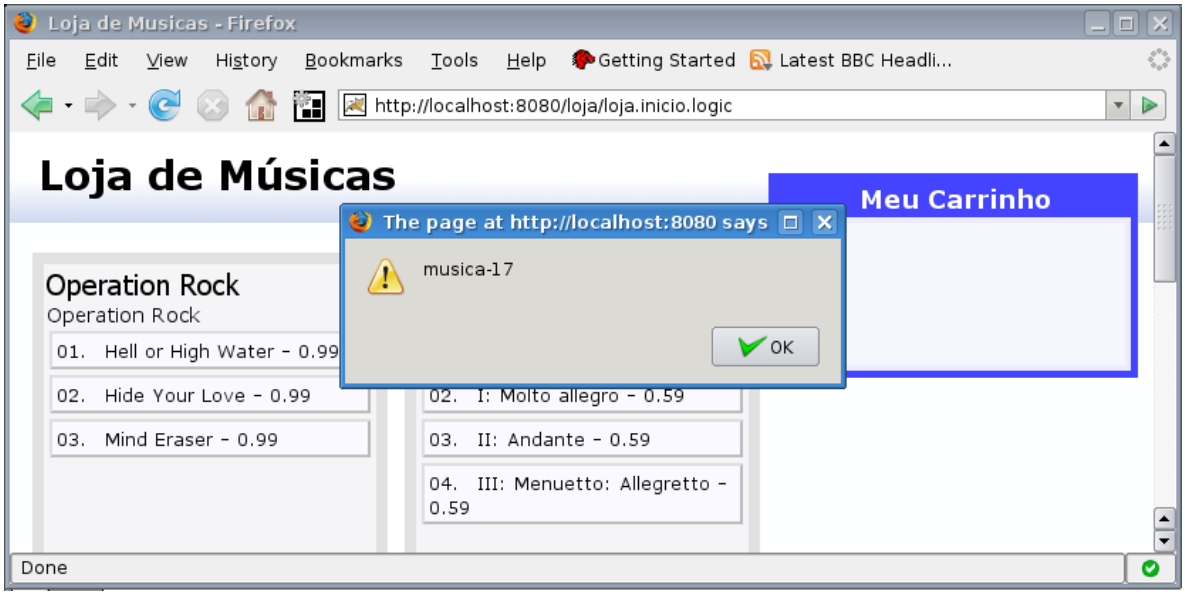
1) Crie um arquivo **loja.js** dentro da pasta **web/javascripts**. Coloque o conteúdo:

```
function atualizaCarrinho(drag) {
	// ...testando...
	alert(drag.id);
}

$(function(){
	$('.musica').Draggable({
		ghosting:	true,
		opacity:	0.7,
		zIndex:		10,
		revert:		true
	});

	$('#carrinho').Droppable({
		accept:		'musica',
		onDrop:		 atualizaCarrinho
	});
});
```

2) Dentro da tag <head> de inicio.ok.jsp, logo antes de fechar </head>, adicione uma tag <script> para importar nosso loja.js:

```
<script src="javascripts/loja.js"></script>
```

3) Acesse sua pagina novamente e teste o arrastar e soltar.



10.5 - Gerenciamento do carrinho na Sessão

Para fazer o nosso sistema funcionar de verdade, precisamos preparar as lógicas no servidor para gerenciar nosso carrinho de compras. A abordagem será manter em sessão o Carrinho de compras do usuário e, quando ele clicar em finalizar, recuperar as Musicas no Carrinho e salvar uma Venda.

Por ora, vamos criar uma classe Carrinho para representar nosso carrinho de compras. Note que, como o Carrinho será uma entidade gerenciada diretamente em memória (sessão), ela não é uma entidade do hibernate (sem @Entity).

```
public class Carrinho {

	private List<Musica> musicas = new ArrayList<Musica>();

	private double total = 0.0;

	public void adicionaMusica(Musica m){
		this.musicas.add(m);
		this.total += m.getPreco();
	}

	public List<Musica> getMusicas() {
		return musicas;
	}

	public double getTotal() {
		return total;
	}

}
```

Para gerenciar o ciclo de vida do carrinho na sessão, vamos ter um componente CarrinhoLogic. Ele terá um Carrinho em escopo de sessão e algumas lógicas para manipula-lo:

```
@Component("carrinho")
@InterceptedBy({DaoInterceptor.class})
public class CarrinhoLogic {

	private final DaoFactory daoFactory;

	@In(scope=ScopeType.SESSION, required=false)
	@Out(scope=ScopeType.SESSION)
	private Carrinho carrinho = new Carrinho();

	public CarrinhoLogic(DaoFactory daoFactory) {
		this.daoFactory = daoFactory;
	}

	public void adiciona (Musica m) {
		Musica musica =
this.daoFactory.getMusicaDao().procura(m.getId());
		this.carrinho.adicionaMusica(musica);
	}

	public void lista() {
	}
}
```

Ou seja, as duas operações basicas sao: adicionar uma nova musica a partir de um id e listar as coisas do carrinho.

Note que usamos 2 anotacoes no atributo carrinho. @In indica que queremos ler o carrinho da sessão (e não da erro se não existir). @Out indica que queremos tambem que o carrinho seja salvo na sessao.

10.6 - Exercícios

1) Crie a classe Carrinho no pacote br.com.caelum.lojavirtual.modelo

```
public class Carrinho {

	private List<Musica> musicas = new ArrayList<Musica>();

	private double total = 0.0;

	public void adicionaMusica(Musica m){
		this.musicas.add(m);
		this.total += m.getPreco();
	}

	public List<Musica> getMusicas() {
		return musicas;
	}

	public double getTotal() {
		return total;
	}

}
```

2) Gere os getters e setters pro Carrinho

3) Crie a classe CarrinhoLogic no pacote br.com.caelum.lojavirtual.logic

```
@Component("carrinho")
@InterceptedBy({DaoInterceptor.class})
public class CarrinhoLogic {

	private final DaoFactory daoFactory;

	@In(scope=ScopeType.SESSION, required=false)
	@Out(scope=ScopeType.SESSION)
	private Carrinho carrinho = new Carrinho();

	public CarrinhoLogic(DaoFactory daoFactory) {
		this.daoFactory = daoFactory;
	}

	public void adiciona (Musica m) {
		Musica musica =
this.daoFactory.getMusicaDao().procura(m.getId());
		this.carrinho.adicionaMusica(musica);
	}

	public void lista() {
	}
}
```

10.7 - Chamando as lógicas com Ajax

Mas quem ira chamar as lógicas no CarrinhoLogic? Ajax!

A adição de uma Musica ao Carrinho será feita no instante em que o cliente soltar uma musica em cima do do carrinho. Iremos acionar então uma requisição ao servidor via Ajax que disparara a lógica e receber como resposta o carrinho atualizado.

A idéia agora não é mais criar uma pagina completa para listagem de musicas no carrinho, por exemplo. A saída da listagem do carrinho não é um html completo, mas apenas um pedaço, aquele correspondente a lista de musicas apenas.

Atraves de Ajax iremos receber esse pedaço de pagina html como resultado e iremos atualizar a parte correspondente na pagina via javascript. Não é mais necessário dar um refresh na tela toda e carregar tudo; apenas a parte da lista é carregada.

Assim, o jsp que lista as musicas que estão no carrinho, é bem curto:

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<ul>
<c:forEach var="musica" items="${carrinho.musicas}">
      <li class="compra">
            <b>${musica.titulo} ($${musica.preco}) </b> <br/>
            CD ${musica.cd.titulo} - ${musica.cd.artista}
      </li>
</c:forEach>
</ul>

<div id="total">
      Total: $${carrinho.total}
</div>
```

Faremos a lógica do adiciona também devolver a listagem de musicas, colocando no views.properties:

```
carrinho.adiciona.ok = carrinho.lista.logic
```

Agora só falta fazermos a chamada ao adiciona via ajax. Ou seja, a função atualizaCarrinho deverá chamar **carrinho.adiciona.logic** passando como argumento a **musica.id** e atualizando o conteúdo do carrinho com a nova listagem recebida.

Com o jquery isso é muito simples:

```
function atualizaCarrinho(drag) {
      id = drag.id.substr(7);
      $('#compras').load('carrinho.adiciona.logic', {'musica.id': id});
}
```

Na primeira linha, pegamos o id do elemento no DOM e extraímos o id

do banco de dados. Na segunda linha, dizemos ao jquery que queremos fazer uma requisição ajax para a url carrinho.adiciona.logic e com o o parâmetro 'musica.id'. O método load faz o request e pega o resultado e carrega dentro do elemento desejado (no nosso caso, #compras).

Pode parecer que o estado da nossa aplicação está salvo no cliente e não no servidor, mas isso não é verdade: tudo está salvo na session. Por isso, vamos fazer nossa pagina "a prova de refreshs". Mesmo que o usuário atualize a pagina, ou saia do site e volte depois, queremos que seu carrinho seja recuperado da sessão.

Ou seja, o estado inicial da pagina já deve ser direto o carrinho no estado atual. A forma mais simples de fazer isso é usando o <c:import> para importar carrinho.lista.logic logo quando a pagina for acessada:

```
<div id="compras">
      <c:import url="carrinho.lista.logic" />
</div>
```

10.8 - Exercicio

1) Crie um diretório **carrinho** na pasta **web**

2) Crie um arquivo **web/carrinho/lista.ok.jsp** com o seguinte conteúdo:

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<ul>
<c:forEach var="musica" items="${carrinho.musicas}">
      <li class="compra">
            <b>${musica.titulo} ($${musica.preco}) </b> <br/>
            CD ${musica.cd.titulo} - ${musica.cd.artista}
      </li>
</c:forEach>
</ul>

<div id="total">
      Total: $${carrinho.total}
</div>
```

3) Abra o views.properties e adicione:

```
carrinho.adiciona.ok = carrinho.lista.logic
```

4) No arquivo **loja.js** altere a implementação da função **atualizaCarrinho** para:

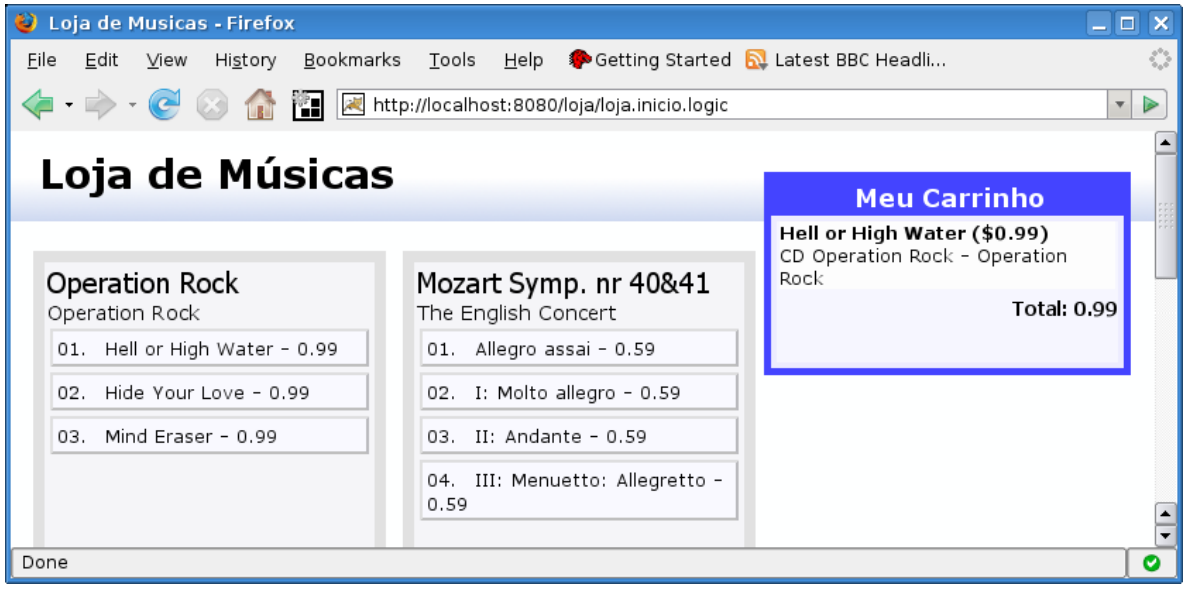
```
function atualizaCarrinho(drag) {
      id = drag.id.substr(7);
      $('#compras').load('carrinho.adiciona.logic', {'musica.id': id});
}
```

5) Localize o div de id "compras" no arquivo inicio.ok.jsp. Dentro dele,

coloque o import da listagem do carrinho:

```
<div id="compras">
      <c:import url="carrinho.lista.logic" />
</div>
```

6) Teste seu sistema. Coloque musicas no carrinho e veja o resultado.



Visite outro site e volte à loja para ver como o carrinho está salvo na sessão.

Partial html

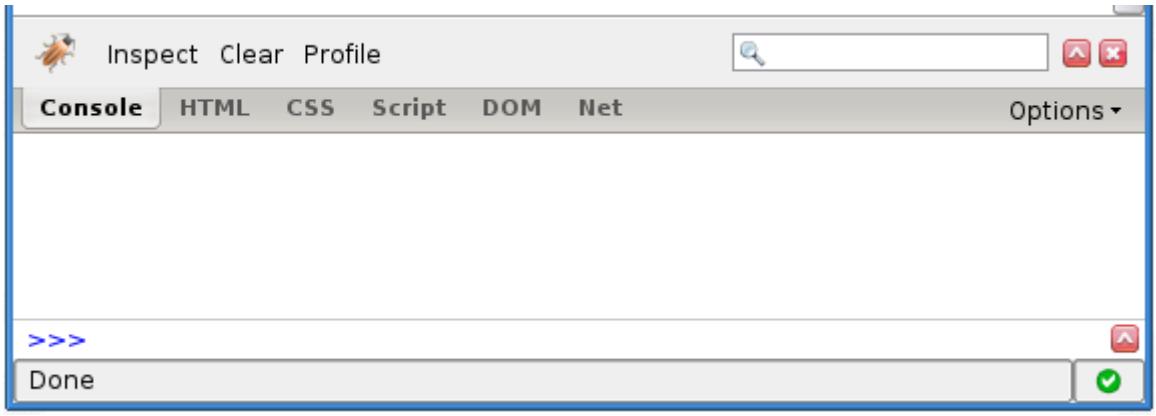
As paginas como as que fizemos aqui com pedaços de html são conhecidas como "partial html". São muito úteis quando unidas ao Ajax.

10.9 - Usando o Firebug para ver o Ajax acontecendo

O Firebug é uma extensão para o navegador Firefox para facilitar o desenvolvimento de páginas web. Possui debugger de javascript, navegação da árvore DOM, console de javascript, logger de requisições ajax e não-ajax e muitas outras coisas.

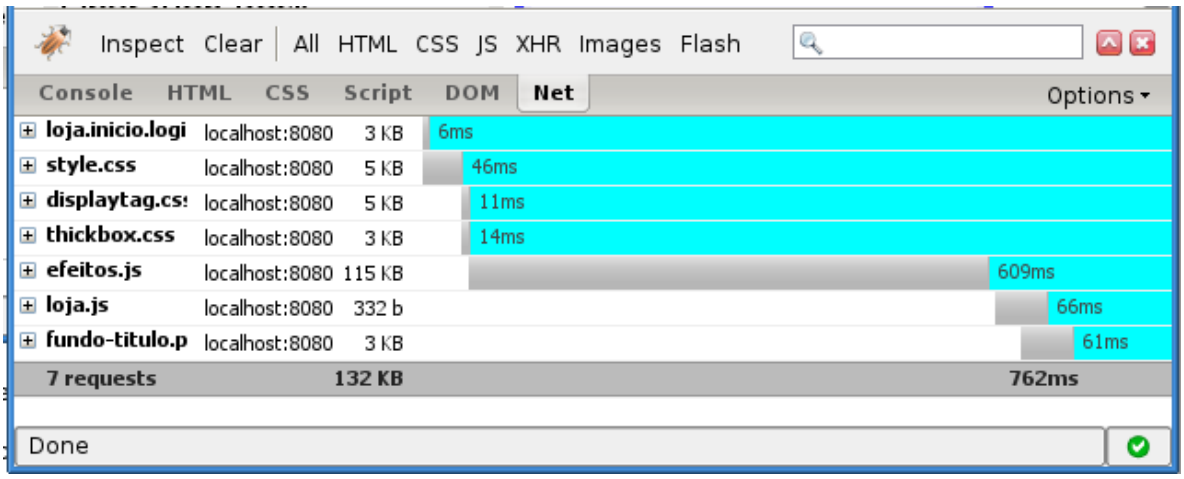
Vamos usar o Firebug agora para ver as requisições ajax feitas no nosso site.

Feche o navegador e abra novamente (para começar uma nova sessão). Clique no ícone verde no canto direito inferior do Firefox para abrir o Firebug.



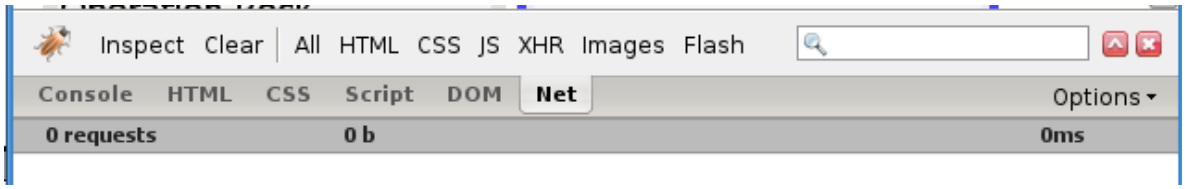
Selecione a aba **Net** para observarmos o tráfego de rede feito em nossa página.

Acesse http://localhost:8080/loja/loja.inicio.logic e observe todo o tráfego gerado (imagens, css, javascript etc).

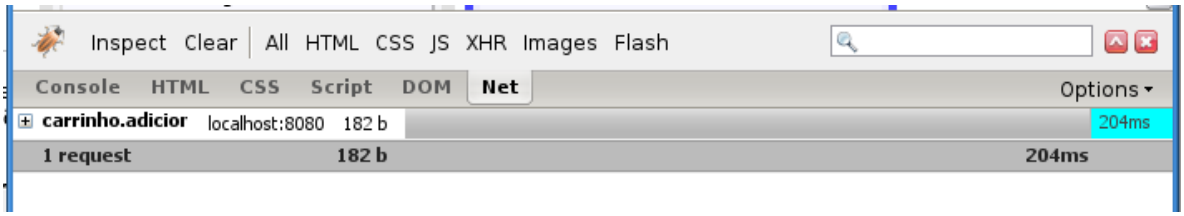


O Firebug te mostra detalhadamente todo o tráfego gerado. Cada URL que foi requisitada, quanto tempo levou a requisição, a quantidade de bytes trafegados e outras opções.

Dentro dessa aba **Net** selecione no topo a opção **XHR**. Isso limitará o logger às requisições do Ajax (chamdas de XmlHttpRequest, XHR). Repare que inicialmente está vazia:



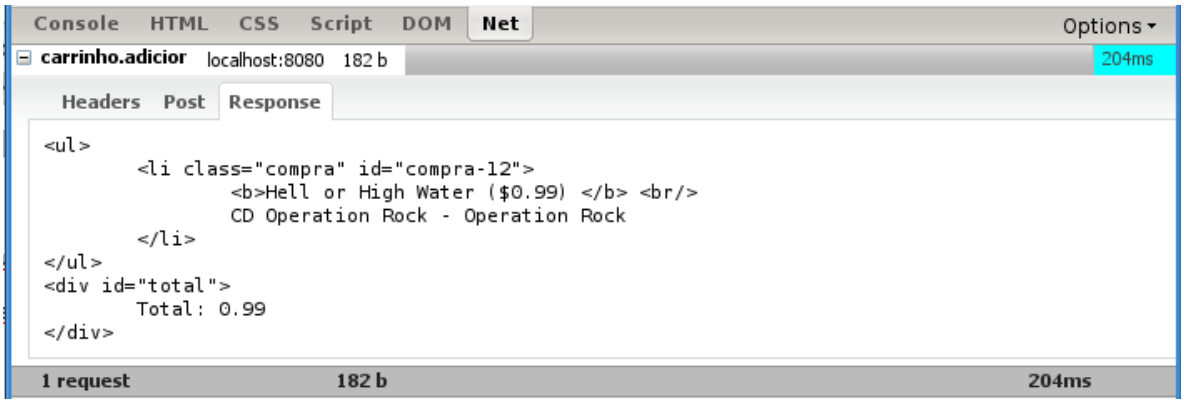
Agora teste nosso site arrastando uma música e soltando dentro do carrinho. Observe o log do Firebug: é a nossa requisição ajax:



O Firebug está indicando que foi feita uma requisição dentro da página (AJAX!) para a url carrinho.adiciona.logic. Se você clicar no **+** ao lado do request, verá detalhes sobre o request: os headers http, o que foi enviado (post), e o que foi recebido (response).

Repare no Post: estamos enviando para o servidor qual é o id da música que queremos adicionar no carrinho:

E repare na resposta do servidor: um **pedaço de html**. É justamente a parte da página que queremos atualizar, e não a página inteira. Isso é o Ajax!



Explore outras funcionalidades do Firebug. E acostume-se a usá-lo durante seu desenvolvimento web. Ele é particularmente útil para ver os erros de Ajax que ficariam escondidos (já que não atualizamos a página toda).

10.10 - Finalizar compra

Depois que o cliente enche o carrinho de compras, ele precisa finaliza a compra, dando seu nome, email, cartão de credito etc.

Usaremos um formulário onde o cliente preenche seus dados, envia e recebe as músicas para download.

Quando o cliente efetuar uma compra, nós pegamos na sessão as

musicas que ele escolheu e os dados preenchidos no formulário e salvamos no banco de dados. Usaremos um bean Venda que contem todos esses dados e uma relacionamento @ManyToMany com Musica.

```
@Entity
public class Venda {

	@Id
	@GeneratedValue
	private Long id;

	@ManyToMany
	private List<Musica> musicas;

	private String nome;

	private String email;

	private String cartao;

}
```

E, na LojaLogic, criaremos uma lógica finalizaCompra. Essa lógica recebe os dados da Venda digitados pelo cliente e associa as musicas do carrinho à essa venda. No final, o carrinho é esvaziado e a venda outjetada para o jsp.

Como precisamos acessar o carrinho, colocaremos um atributo Carrinho na LojaLogic:

```
	@In(scope = ScopeType.SESSION, required = false)
	@Out(scope = ScopeType.SESSION)
	private Carrinho carrinho;
```

Esse atributo precisa ser lido da sessão e também escrito na sessão (quando limparmos ele).

E o atributo Venda e seu getter para outjetar no JSP:

```
	private Venda venda;
```

Por fim, o método finalizaCompra:

```
	public void finalizaCompra(Venda venda) {
		venda.setMusicas(this.carrinho.getMusicas());

		this.daoFactory.beginTransaction();
		this.daoFactory.getVendaDao().adiciona(venda);
		this.daoFactory.commit();

		this.carrinho = null;
		this.venda = venda;
	}
```

10.11 - Exercicios

1) Crie a classe Venda no pacote br.com.caelum.lojavirtual.modelo

```
@Entity
public class Venda {

	@Id
	@GeneratedValue
	private Long id;

	@ManyToMany
	private List<Musica> musicas;

	private String nome;

	private String email;

	private String cartao;

}
```

2) Configure a Venda no hibernate.cfg.xml

```
  <mapping class="br.com.caelum.lojavirtual.modelo.Venda"/>
```

3) Rode o GeraBanco

4) Adicione um método getVendaDao na DaoFactory:

```
	public Dao<Venda> getVendaDao() {
		return new Dao<Venda>(this.session, Venda.class);
	}
```

5) Na classe LojaLogic, implemente a lógica de finalizar venda:

a) Crie dois atributos a mais:

```
	@In(scope = ScopeType.SESSION, required = false)
	@Out(scope = ScopeType.SESSION)
	private Carrinho carrinho;

	private Venda venda;
```

b) Gere getter para Venda

c) Implemente o finalizaCompra:

```
	public void finalizaCompra(Venda venda) {
		venda.setMusicas(this.carrinho.getMusicas());

		this.daoFactory.beginTransaction();
		this.daoFactory.getVendaDao().adiciona(venda);
		this.daoFactory.commit();

		this.carrinho = null;
		this.venda = venda;
	}
```

d) Adicione uma lógica formulario vazia:

```
	public void formulario() {
	}
```

No final, sua LojaLogic deve estar assim:

```
@Component("loja")
@InterceptedBy( { DaoInterceptor.class })
public class LojaLogic {

	private final DaoFactory daoFactory;

	@In(scope = ScopeType.SESSION, required = false)
	@Out(scope = ScopeType.SESSION)
	private Carrinho carrinho;

	private Venda venda;

	public LojaLogic(DaoFactory daoFactory) {
		this.daoFactory = daoFactory;
	}

	public void inicio() {
	}

	public void formulario() {
	}

	public void finalizaCompra(Venda venda) {
		venda.setMusicas(this.carrinho.getMusicas());

		this.daoFactory.beginTransaction();
		this.daoFactory.getVendaDao().adiciona(venda);
		this.daoFactory.commit();

		this.carrinho = null;
		this.venda = venda;
	}

	public Venda getVenda() {
		return venda;
	}

	public List<Cd> getCds() {
		return this.daoFactory.getCdDao().listaTudo();
	}
}
```

10.12 - O formulário de finalização da compra

Vamos criar um formulário onde o usuário possa finalizar sua compra. Algo assim:

```
	<form action="loja.finalizaCompra.logic" method="POST">
		Seu nome:
		<input type="text" name="venda.nome">
```

```
            E-mail:
            <input type="text" name="venda.email">

            Cartao de credito:
            <input type="text" name="venda.cartao">

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

Por fim, na pagina do resultado final, a **finalizaCompra.ok.jsp**, listaremos todas as musicas que o usuário comprou (depois podemos colocar um link para o mp3, por exemplo):

```
<ul>
<c:forEach var="musica" items="${venda.musicas}">
      <li class="compra">
            <b>${musica.titulo}</b> <br/>
            CD ${musica.cd.titulo} - ${musica.cd.artista}
      </li>
</c:forEach>
</ul>
```

10.13 - Exercícios

1) Crie o arquivo **web/loja/formulario.ok.jsp** pelo menu **File, New, Other, Amateras JSP File**

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<html>
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
      <title>Finalizar a compra</title>
      <link rel="stylesheet" type="text/css" href="css/style.css"/>
</head>
<body>

      <h2>Preencha seus dados para finaliza a compra no site</h2>

      <form action="loja.finalizaCompra.logic" method="POST">
            Seu nome:
            <input type="text" name="venda.nome">

            E-mail:
            <input type="text" name="venda.email">

            Cartao de credito:
            <input type="text" name="venda.cartao">

            <input type="submit" value="Comprar">
      </form>

</body>
</html>
```

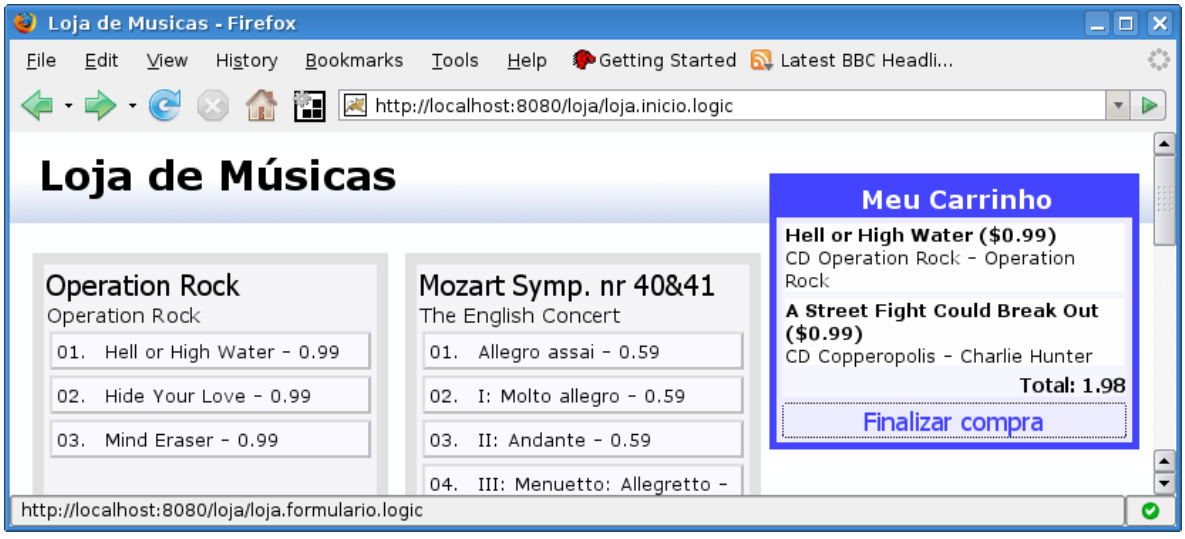
2) Crie o arquivo **web/loja/finalizaCompra.ok.jsp** pelo menu **File, New, Other, Amateras JSP File**

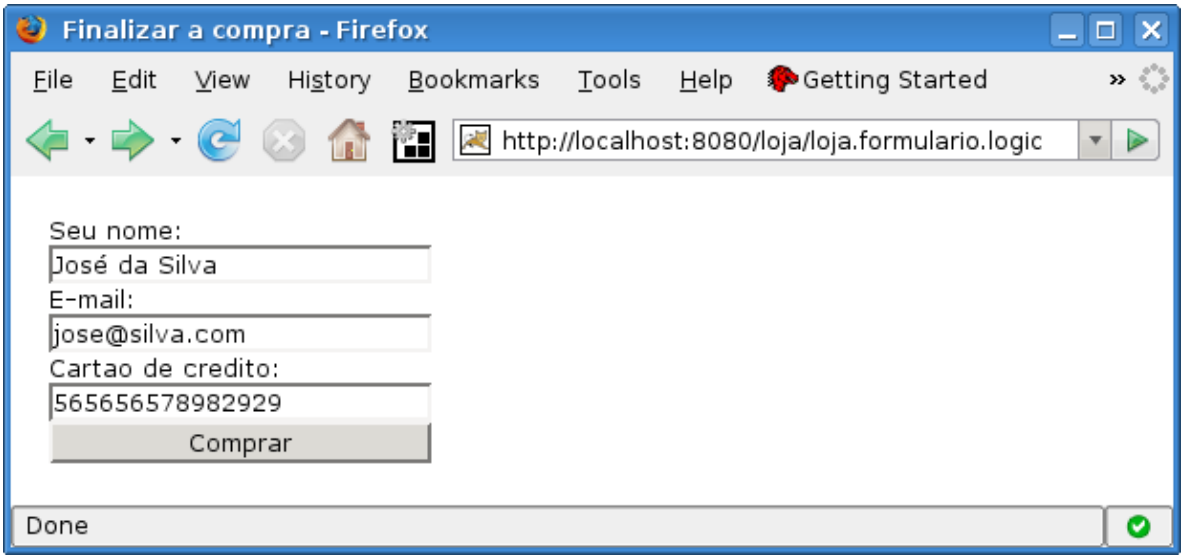
```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
<head>
	<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
	<title>Finalizar a compra</title>
	<link rel="stylesheet" type="text/css" href="css/style.css"/>
</head>
<body>

<h3>Compra efetuada com sucesso! Baixe seus mp3!</h3>
<ul>
<c:forEach var="musica" items="${venda.musicas}">
	<li class="compra">
		<b>${musica.titulo}</b> <br/>
		CD ${musica.cd.titulo} - ${musica.cd.artista}
	</li>
</c:forEach>
</ul>

</body>
</html>
```

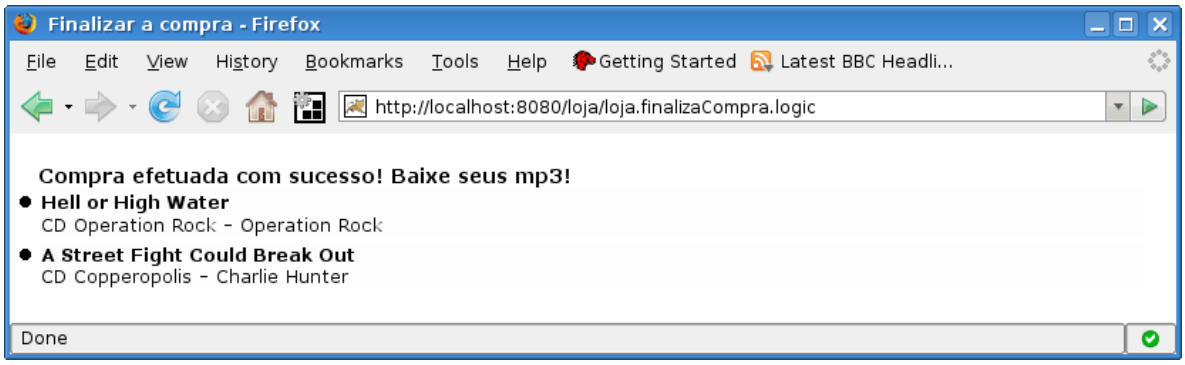
3) Localize o div com id=carrinho na pagina. Adicione um link para abrir o thickbox. Deve ficar assim:

```
<div id="carrinho">
	<h2>Meu Carrinho</h2>
	<div id="compras">
		<c:import url="carrinho.lista.logic" />
	</div>

	<a href="loja.formulario.logic" title="Finalizar compra">
		Finalizar compra
	</a>
</div>
```

4) Abra nossa loja e adicione algumas músicas no carrinho. Clique em finalizar compra. Preencha o form e envie.







10.14 - Usando o Thickbox

O ThickBox é um plugin do JQuery que, entre outras coisas, pode mostrar uma página dentro de outra através de uma janela animada bem interessante por meio de um iframe.

Vamos usá-la para nosso formulário. Precisamos mudar pouca coisa.

10.15 - Exercícios

1) Altere o link de Finalizar Compra para abrir um ThickBox. Basta adicionar o class e alterar o href:

```
<a href="loja.formulario.logic?KeepThis=true&TB_iframe=true&height=300&width=400"
 class="thickbox" title="Finalizar compra">
      Finalizar compra
</a>
```

2) Teste novamente o sistema e veja a janela funcionando!

Esvaziar o carrinho

Usando o thickbox o carrinho continua sujo após a compra. Na verdade, o que acontece é que apenas esvaziar o carrinho na Session no servidor não é o suficiente. Quando usamos Ajax, precisamos lembrar de atualizar o cliente também.

Nosso caso é bem simples: quando a compra for confirmada, precisamos fazer um novo request Ajax que atualize o estado do carrinho na página. Apenas lembre que, como estamos usando iframes, precisamos acessar o carrinho da página pai (parent).

Abra o arquivo finalizaCompra.ok.jsp e adicione dentro da tag head:

```
<script>parent.$('#compras').load('carrinho.lista.logic');</script>
```

capítulo 11

Apêndice – Melhorando a Loja Virtual

"Tantas pessoas que escrevem e tão poucas que lêem."
André Gide -

Este apêndice mostra algumas melhorias possíveis, como:
- só deixar o usuário adicionar uma musica de cada no carrinho
- remoção de itens do carrinho
- tocador de musica para ouvir antes de comprar
- e mais alguns detalhes

11.1 - Capinhas de mp3

Para deixar nosso sistema mais interessante, vamos adicionar a possibilidade de se capas nos Cds e também associar um arquivo mp3 à cada Musica.

11.2 - Exercícios

1) Na sua class **Cd** adicione um atributo imagemCapa e gere seu getter e setter:

```
private String imagemCapa;
// get e set
```

2) Na classe **Musica**, adicione um atributo **mp3** e gere seu getter e setter:

```
private String mp3;
// get e set
```

3) Abra o arquivo **web/cd/formulario.ok.jsp** e adicione dentro do form um campo para inserção da imagem da capa:

```
Caminho da imagem da capa:
<input type="text" name="cd.imagemCapa" value="${cd.imagemCapa}"/>
```

4) Abra o arquivo **web/musica/formulario.ok.jsp** e adicione dentro do form um campo para o mp3:

```
Caminho do mp3:
<input type="text" name="musica.mp3" value="${musica.mp3}"/>
```

5) Rode o **GeraBanco** novamente para atualizar as tabelas.

Dica: junto com o projeto, há um arquivo InsereDados com uma série de dados de teste que você pode usar para popular seu banco de dados.

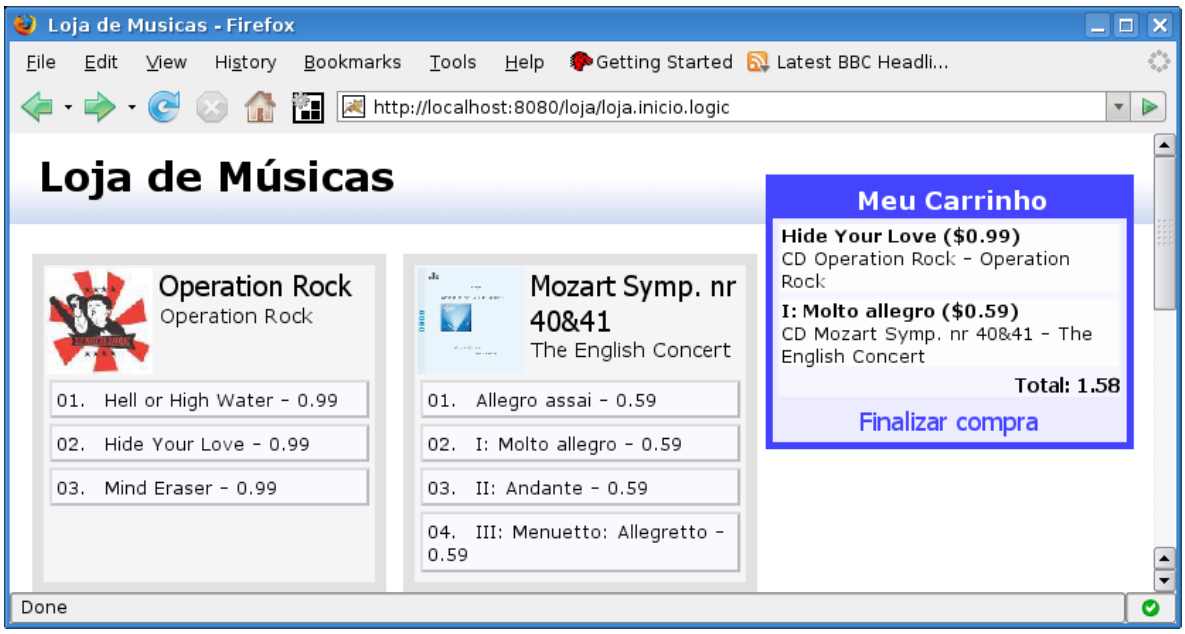
6) Altere nossa loja para exibir as capas. No arquivo **loja/inicio.ok.jsp** adicione uma <img> dentro do , desta forma:

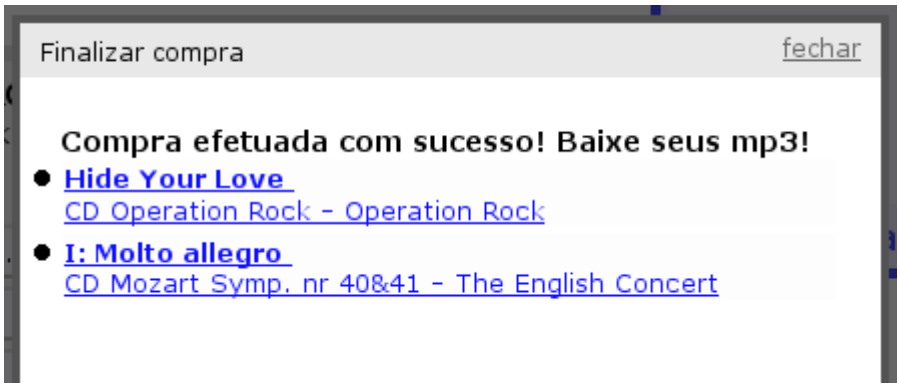
```
...
<div id="cd-${cd.id}" class="cd">
      <img src="img/cd/${cd.imagemCapa}"/>
      <span class="titulo">${cd.titulo}</span>
      ...
```

7) Altere a mensagem de finalizar compra para permitir que o comprador faça o download das músicas compradas. Abra o arquivo **loja/finalizaCompra.ok.jsp** e adicione um link para o mp3:

```
...
<li class="compra">
      <a href="mp3/${musica.mp3}">
            <b>${musica.titulo}</b> <br/>
            CD ${musica.cd.titulo} - ${musica.cd.artista}
      </a>
</li>
...
```

8) Teste sua loja. Abra a loja, veja as capas, efetue uma compra e depois baixe os mp3 que comprou.





11.3 - Unicidade das músicas

Será que faz sentido alguém comprar duas musicas iguais para download? Provavelmente não.

Podemos então mudar nosso carrinho de compras para não aceitar Musicas que já estejam no Carrinho. Ao invés de usar um List, podemos usar um Set, que garante não repetição.

O problema é que o Set precisa saber como consideramos duas Musicas iguais (no nosso caso, pelo id). Precisamos implementar o método equals para fazer tal verificação. E quando implementamos o equals, devemos implementar o hashcode também, para seguir o contrato.

11.4 - Exercício

1) Na classe Musica, sobrescreva os métodos equals e hashcode que consideram duas musicas com mesmo id iguais.

```
@Override
public int hashCode() {
	return this.id.hashCode();
}

@Override
public boolean equals(Object obj) {
	try {
		Musica outra = (Musica) obj;
		return this.getId().equals(outra.getId());
	} catch (Exception e) {
		return false;
	}
}
```

2) Na classe Carrinho, mude o atributo **musicas** para o tipo Set:

```
private Set<Musica> musicas = new LinkedHashSet<Musica>();
```

Não se esqueça de mudar o getter também!

E mude o método adicionaMusica para apenas somar o total quando a musica for realmente inserida:

```
public void adicionaMusica(Musica m) {
      if (this.musicas.add(m)) {
            this.total += m.getPreco();
      }
}
```

(usamos um LinkHashSet pois esse mantem a ordem de inserção)

3) Na classe Venda, mude musicas também para Set:

```
@ManyToMany
private Set<Musica> musicas;
```

4) Teste agora adicionar 2 musicas iguais no carrinho.

11.5 - Remoção do carrinho

Vamos fazer a remoção de musicas do carrinho através de drag and drop. Quando um usuário quiser remover alguma musica, bastara arrastá-la para fora do carrinho.

Precisaremos implementar a remoção em nosso Carrinho

```
public void removeMusica(Musica m) {
      if (this.musicas.remove(m)) {
            this.total -= m.getPreco();
      }
}
```

e criar uma logica remove no componente CarrinhoLogic:

```
public void remove (Musica m){
      Musica musica = this.daoFactory.getMusicaDao().procura(m.getId());
      this.carrinho.removeMusica(musica);
}
```

E no **views.properties**:

```
carrinho.remove.ok = carrinho.lista.logic
```

Precisamos definir os itens do carrinho como Draggables e definir uma função (callback) que remove do Carrinho que está na sessão através de ajax.

Antes, porem, precisamos definir um id para os elementos <li class=compra> na carrinho/lista.ok.jsp:

```
 <li class="compra" id="compra-${musica.id}">
```

Adicione no final do arquivo **loja.js**:

```
function remove() {
	id = this.id.substr(7);
	$('#compras').load('carrinho.remove.logic', {'musica.id': id});
}

$(function() {
	$('.compra').Draggable({
		ghosting:	true,
		opacity:	0.7,
		zIndex:		10,
		revert:		true,
		onStop:		remove
	});
});
```

Note que só conseguimos remover do Carrinho com essa facilidade porque estamos usando um Set e sobrescrevemos nosso método equals.

11.6 - Ouvir as musicas

Vamos adicionar um recurso ao nosso site que permita ao usuario ouvir a musica que ele vai comprar. Vamos usar um tocador de mp3 desenvolvido em flash e muito simples. De modo que o usuário terá apenas que ter o flash player instalado em sua maquina.

Para tocar a musica, também usaremos Drag and Drop: colocaremos uma caixa de som na loja e quando o usuário quiser ouvir, basta arrastar a música em cima da caixa de som.

Adicione ao **inicio.ok.jsp** o player com a caixa de som. Coloque depois do div do carrinho:

```
<div id="player">
	<img src="img/multimedia.png"/>
</div>
```

Para tocar precisamos do endereço do mp3 a ser tocado. Vamos criar um link com o endereço mp3 de cada musica na listagem de musicas:

```
<c:forEach var="musica" items="${cd.musicas}">
	<li class="musica" id="musica-${musica.id}">
		${musica.titulo} - ${musica.preco}"
		<a href="${musica.mp3}">ouvir</a>
	</li>
</c:forEach>
```

Em situações normais, o link ficara oculto. Ele servira apenas para pegarmos o endereço do mp3 e passarmos para a função tocar().

Vamos definir duas funções javascript: uma que toca uma determinada musica e outra que pára a musica que estiver tocando: (coloque no arquivo loja.js)

```
function tocar(musica) {
	parar_musica(); // para se tiver tocando
```

```
	// recupera o endereco do mp3
	var href = $('a', musica).attr('href')
	var caminho = "mp3/player.swf?autoplay=true&song_url=mp3/" + href;

	// constroi html pro flash
	var flash = '<object type="application/x-shockwave-flash" '
		    + ' data="'+caminho+'" width="17" height="17">'
		    + ' <param name="movie" value="'+caminho+'" />'
		    + ' </object>';

	$("#player").append(flash);
}
function parar_musica() {
	$("#player object").remove();
}
```

Repare o que estamos fazendo: através do jquery inserimos o flash para tocar e o removemos para parar de tocar.

Por fim, vamos definir que a imagem também é um lugar onde podemos soltar musicas, ou seja, um Droppable:

```
$(function() {
	$('img.logo').Droppable({
		accept: 'musica',
		onDrop: tocar
	});
}
```

E definir que, quando o usuário clicar em cima da imagem da caixa de som, a musica deve ser parada:

```
$(function() {
	$('img.logo').click(parar_musica);
}
```

11.7 - Formatando moeda

Para formatar a saída dos cálculos do carrinho como moeda, podemos usar a JSTL Format (fmt). Basta acrescentar o cabeçalho no topo do arquivo:

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

E usa a tag formatNumber:

```
<fmt:formatNumber value="${valor}" type="currency"/>
```

Por padrão, o Java pega o locale do navegador do usuário para definir o formato. Para forçar o uso de um locale, podemos usar o setLocale:

```
<fmt:setLocale value="pt_BR"/>
```

11.8 - Listar Vendas

Crie uma VendaLogic protegida da área administrativa do Site que

permita listar as vendas feitas na loja virtual. Use displaytag.

11.9 - I18N

Podemos aproveitar a existência da JSTL e usar a taglib fmt para a internacionalização. A tag a seguir procura no arquivo **messages.properties** a chave **bemvindo** e mostra o valor da mesma:

<fmt:message key=”bemvindo”/>

11.10 - Export no displaytaglib

O atributo export=”true” da tabela permite exportar para alguns tipos de formatos. Tente brincar com ele e crie relatórios simples para aquelas listagens que os clientes não haviam previsto!

capítulo 12

Apêndice – Melhorando o login

Os soberbos são ordinariamente ingratos; consideram os benefícios como tributos que se lhes devem
Marquês de Maricá -

Este apêndice mostra como criar um sistema de permissões com o Vraptor e anotações, de forma bem elegante.

12.1 - A permissão do Usuario

Adicione na sua classe Usuario um atributo para conter as permissões de acesso que este usuário tem. (por exemplo, "admin" ou "manager" etc).

```
private String role;
// gerar get e set
```

Além disso, adicione na classe Usuario um método que verifica se ele possui ou não alguma permissão. A idéia é receber uma String com o role a ser verificado e devolver um boolean indicando se o Usuario possui ou não aquela permissão:

```
public boolean hasRole(String s) {
      // role com espaÃ§o antes e depois
      String role = " " + this.role + " ";

      // verifica se tem o role
      return role.contains(" " + s + " ");
}
```

Nota: estamos considerando que o Usuario pode ter mais de uma permissão, basta colocar o role com as várias permissões separadas por espaço "admin manager".

12.2 - Exercícios

1) Na classe Usuario, crie um atributo role e gere seus gets e sets:

```
private String role;
// gerar get e set
```

2) Rode o GeraBanco novamente para regerar suas tabelas.

3) Adicione, na classe Usuario, o método hasRole:

```
public boolean hasRole(String s) {
      // role com espaco antes e depois
      String role = " " + this.role + " ";
```

```
            // verifica se tem o role
            return role.contains(" " + s + " ");
      }
```

4) Acrescente um input no seu **usuario/formulario.ok.jsp** para inserir o role do Usuario:

```
Role (separado por espaco):
<input type="text" name="usuario.role" />
```

12.3 - A anotação

Para facilitar a forma como configuramos as permissões de cada lógica, vamos criar nossa própria anotação. Imagine poder simplesmente fazer isso:

```
@Role("admin")
public void remove (Usuario usuario) {
      // só que tem role admin pode remover um Usuario
}
```

A partir do Java 5, podemos criar nossa anotação da seguinte forma:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Role {
      String value();
}
```

O @Retention indica que nossa anotação estará disponível em Runtime também (não só na compilação). O @Target indica que nossa anotação pode ser usada em métodos.

E o método value() do tipo String indica que nossa anotação recebe obrigatoriamente uma String como valor.

12.4 - Exercícios

1) Crie a anotação **Role** no pacote **br.com.caelum.lojavirtual.modelo**:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Role {
      String value();
}
```

12.5 - O interceptador

Agora precisamos mudar nosso interceptador para verificar essa anotação. Precisamos usar um pouco de Reflection para pegar a anotação do método. Mas com o Vraptor, isso fica muito simples. Veja o código completo comentado:

```
public class AutorizadorInterceptor implements Interceptor {
```

```
	@In(scope = ScopeType.SESSION, required = false)
	private Usuario usuario;

	public void intercept(LogicFlow flow) throws LogicException, ViewException {
		// pega request e response
		HttpServletRequest request = flow.getLogicRequest().getRequest();
		HttpServletResponse response = flow.getLogicRequest().getResponse();

		try {

			// usuario nao esta logado
			if (this.usuario == null) {
				// redireciona pro login
				response.sendRedirect("admin.login.logic");

			} else { // usuario esta logado

				// pega anotacao @Role no método da lógica
				Role role = flow.getLogicRequest().getLogicMethod()
							.getMetadata().getAnnotation(Role.class);

				// tem role definido
				if (role != null) {

					// se usuario tem o role pedido, executa
					if (this.usuario.hasRole(role.value())) {
						flow.execute();
					} else {
						// sem permissao, vai pra admin.semrole.logic
						request.getRequestDispatcher("semrole.jsp")
							.forward(request, response);
					}

				} else {
					// nao tem role, entao pode executar
					flow.execute();
				}
			}
		} catch (Exception e) {
			throw new LogicException(e);
		}

	}
}
```

12.6 - Exercícios

1) Altere seu AutorizadorInterceptador para o código visto acima

2) Crie um arquivo **web/semrole.jsp** com uma mensagem do tipo

```
<h1>Usuario nao tem permissao pra ver essa pagina.</h1>
```

3) Anote alguns métodos de sua lógica para testar. Por exemplo, na classe UsuarioLogic, coloque:

```
	@Role("admin")
```

```
public void remove (Usuario usuario) {
```

4) Teste seu sistema com usuarios diferentes, um que tenha o role "admin" e outro que não tenha.

12.7 - Desafios

Como fazer sua anotação aceitar mais de um Role para o mesmo método? Você pode mudar sua annotation para receber um array de valores:

```
public @interface Role {
      String[] value();
}
```

E usar esses valores nos métodos assim:

```
@Role({"admin", "manager"})
```

Mas você precisa mudar seu interceptador, a lógica que verifica se o usuário tem o role ou não (agora você precisa verificar se ele tem *algum* dos roles). Pense sobre isso.

12.8 - Para saber mais: Sessão e Usuario

O usuário logado esta na sessão e para que o restart do servlet conteiner não destrua sua sessão basta que suas classes armazenadas na sessão sejam serializáveis. Assim o servlet contêiner consegue serializá-las para disco e lembrar quem você era após um restart sempre que possível (existem alguns detalhes). Esse processo não é padrão dos servlet contêiners mas o Tomcat possui tal ferramenta.

capítulo 13

Apêndice B – Criando o Ambiente

Uma grande qualidade ou talento desculpa muitos pequenos defeitos.
Marquês de Maricá -

Como criar um projeto do zero.

13.1 - Introdução

Neste curso, para agilizar, já usamos um zip com toda a estrutura necessária para o projeto web. As pastas necessárias já estavam criadas. E, o que daria mais trabalho, todos os jars necessários já estavam inclusos.

Abaixo segue um passo-a-passo detalhado de como criar esse projeto do zero, de onde pegar os jars e etc.

13.2 - Instalando o eclipse

1-) Baixe o eclipse na página www.eclipse.org.

2-) Descompacte o arquivo e pronto.

13.3 - Instalando o plugin para o tomcat

Um dos plugins mais simples e famosos é o plugin para tomcat desenvolvido pela empresa francesa Sysdeo. O site oficial é:

1-) http://www.sysdeo.com/eclipse/tomcatplugin

2-) Baixe o arquivo tomcatPluginV31beta.zip.

3-) Descompacte o conteúdo desse zip no dentro do diretório plugins onde você instalou o eclipse. Por exemplo, se você instalou o eclipse em c:\eclipse, descompacte o arquivo em c:\eclipse\plugins.

13.4 - Instalando o plugin para arquivos jsp, html e xml

Iremos utilizar o plugin da amateras para dar suporte aos arquivos do tipo jsp, html e xml. O site oficial do plugin é o http://amateras.sourceforge.jp/ e iremos utilizar o plugin chamado EclipseHtmlEditor.

Um pré-requisito para o EclipseHtmlEditor é o GEF do próprio projeto Eclipse. O link é o próprio site do Eclipse, Downloads e então GEF. Baixe a versão relativa ao seu eclipse (por exemplo 3.1.1).

Para instalar o EclipseHtmlEditor basta descompactar o arquivo dentro do diretório c:\eclipse\plugins já o GEF no diretório c:\eclipse.

13.5 - Instalando o plugin Hibernate Tools

1) Baixe o plugin de http://tools.hibernate.org/

2) Descompacte o zip dentro da pasta do eclipse (c:\eclipse por exemplo)

13.6 - Plugins do eclipse no windows

No windows existe uma única diferença: inicie o eclipse com a opção -clean toda vez que você instalar um novo plugin.

13.7 - Firefox e Firebug

Baixe o Firefox em www.getfirefox.com

Instale o Firefox e abra o navegador. Vá até www.getfirebug.com e instale o Firebug.

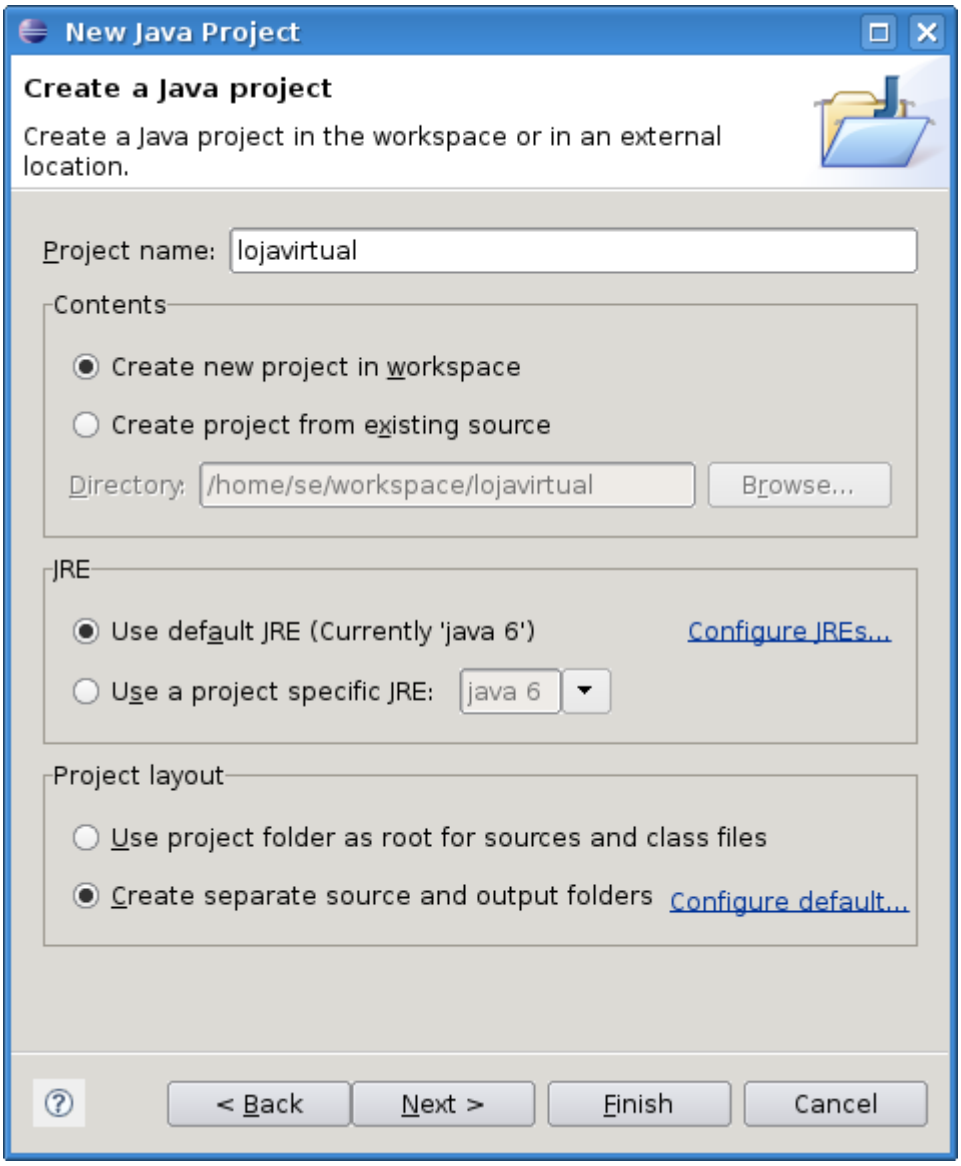
13.8 - Iniciando o projeto

1) Crie um novo projeto Java no Eclipse, chamado **lojavirtual**:

a) Vá em **File -> New -> Project**

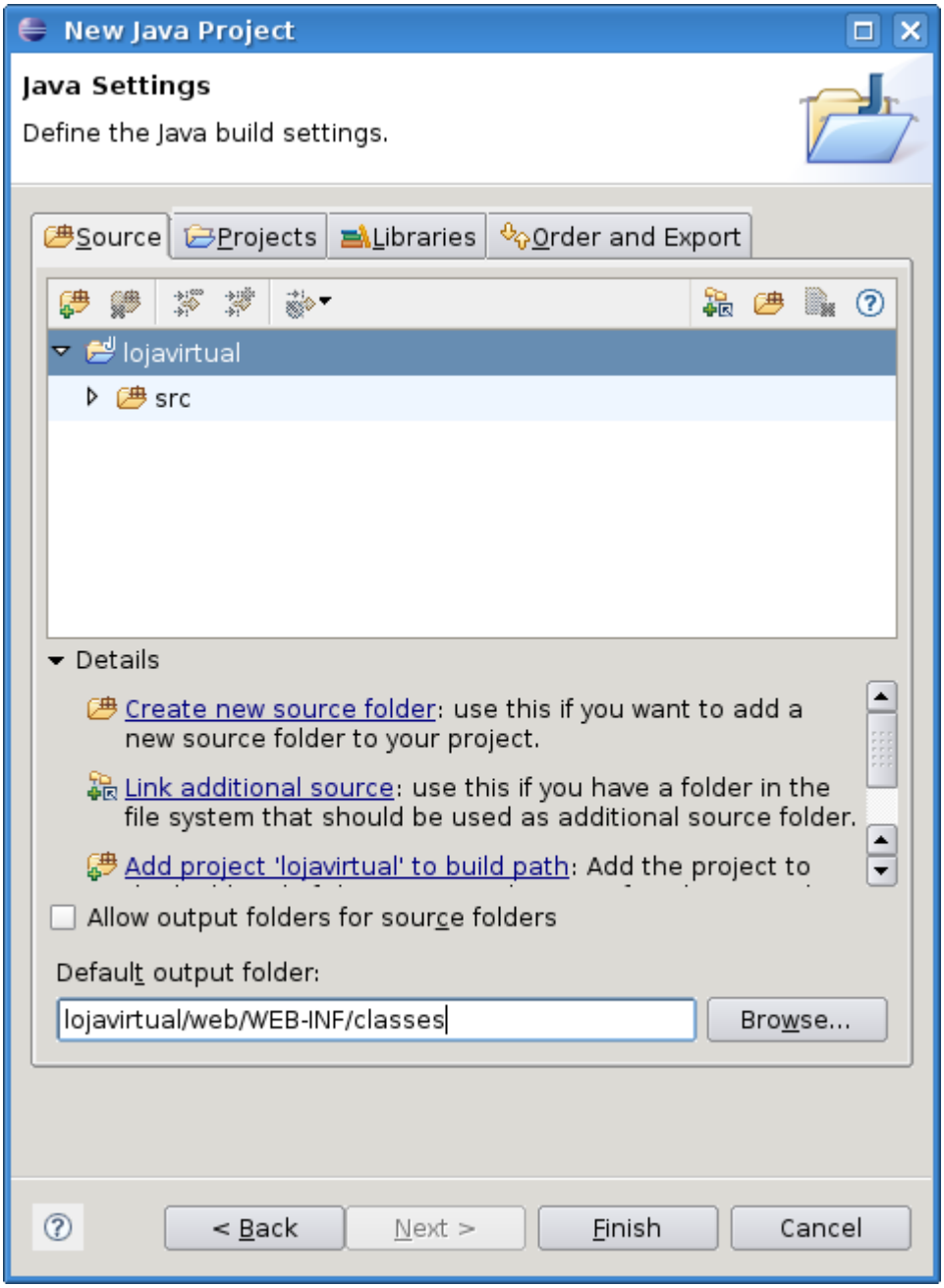
b) Selecione **Java Project** e clique em Next

c) Coloque o nome do projeto como **lojavirtual** e marque a opção **Create separate source and output folders**.



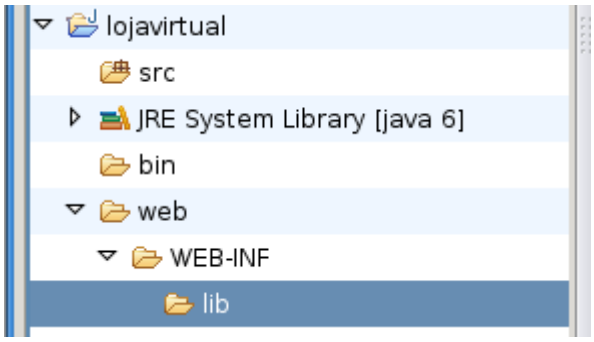
Clique em Next.

d) Coloque o **Default output folder** como **lojavirtual/web/WEB-INF/classes** e clique em Finish.



2) Crie um diretório **lib** dentro da pasta **web/WEB-INF**

Seu projeto deve estar assim:



13.9 - Preparando o hibernate

1) Vá em www.hibernate.org e baixe três pacotes:

- hibernate core
- hibernate annotations
- hibernate validator

Descompacte os três pacotes.

2) Dentro das pastas onde você descompactou os pacotes do hibernate, você encontrará alguns jars. Copie-os para a pasta lib do seu projeto. Dentro daquelas pastas você também encontrará uma pasta lib (uma em core e uma em annotations) com vários jars de dependência do hibernate. Copie também esses jars para o lib do seu projeto.

3) Como usaremos o mysql no nosso projeto, precisamos do driver jdbc do mysql. Baixe em http://www.mysql.com/products/connector/j/ , descompacte o arquivo e encontrará um jar chamado mysql-connector. Copie-o para o lib do seu projeto.

4) Adicione todos os jars copiados até agora no classpath da nossa aplicação. Expanda a pasta lib, selecione todos os jars, clique com o botão direito e escolha **Build Path** -> **Add to Build Path**.

Escondendo os jars no Eclipse

A lista de jars no nosso projeto é muito grande, o que pode prejudicar a visualização dos arquivos do projeto. No Eclipse, você pode filtrar a visualização de recursos do seu projeto de modo a omiti-los ou exibi-los.

Para fazer o Eclipse não mostrar todos os jars, vá no Menu na View Package Explorer, e selecione Filters. Marque o checkbox Name filter patterns e coloque ***.jar**

5) Tanto o hibernate como o vraptor usam o log4j para log. Vamos copiar a configuração padrão do log4j que vem no hibernate. Vá na pasta que você descompactou o hibernate core (não o annotations) e localiza a pasta etc. Dentro desta pasta, existe um arquivo log4j.properties. Copie-o para o diretório src do seu projeto.

13.10 - Instalando vraptor, jstl e displaytag

Baixe o vraptor em www.vraptor.org, a displaytag em http://displaytag.sf.net e a jstl em http://jakarta.apache.org/commons.

Fora isso, a displaytag precisa de mais duas bibliotecas, Jakarta commons lang e a commons beanutils (baixe em http://jakarta.apache.org/commons/)

Extraia todos os pacotes.

Do pacote do vraptor, copie todos os jars que estão dentro da pasta que você descompactou *exceto o jstl.jar* para a pasta **lib** do seu projeto.

Do zip da jstl, copie os jars jstl.jar e standard.jar para o lib do seu projeto.

Do hibernate validator copie o hibernate-validator.jar para o seu lib.

Da displaytag, copie o arquivo **displaytag-x.x.jar** para o seu diretório **lib**.

E copie os jars dos dois commons (lang e beanutils) para o lib do seu projeto.

Volte ao Eclipse, dê um refresh (F5) no seu projeto.

Expanda a pasta WEB-INF/lib, selecione todos os jars lá de dentro, clique com o botão direito e vá em **Build Path -> Add to Build Path**

13.11 - O web.xml

Precisamos criar um arquivo **web.xml** no diretório WEB-INF. Vamos usar o vraptor como Controlador, então temos que configurar a Servlet do Vraptor.

1) Crie um arquivo web.xml dentro do diretório WEB-INF com o seguinte conteúdo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">

	<servlet>
		<servlet-name>vraptor2</servlet-name>
		<servlet-class>org.vraptor.VRaptorServlet</servlet-class>
		<load-on-startup>1</load-on-startup>
	</servlet>

	<servlet-mapping>
		<servlet-name>vraptor2</servlet-name>
		<url-pattern>*.logic</url-pattern>
	</servlet-mapping>
```

```
</web-app>
```

13.12 - O tomcat no windows

Para instalar o tomcat no windows basta executar o arquivo .exe que pode ser baixado no site do tomcat. 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.

Baixe o Tomcat em http://tomcat.apache.org. Para instalá-lo, basta descompactar o zip no lugar que desejar.

13.13 - iniciando o tomcat sem plugin

Entre no diretório de instalação e rode o programa startup.sh: (ou bat se estiver no windows)

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

13.14 - Parando o tomcat sem plugin

Entre no diretório de instalação do tomcat e rode o programa shutdown.sh: (ou bat se estiver no windows)

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