Desenvolvimento Web no Eclipse – Parte II

Vítor E. Silva Souza (vitorsouza@gmail.com)

Implemente uma aplicação pequena completa utilizando *frameworks* e a IDE Eclipse.

Esta é a segunda parte de uma série de artigos sobre desenvolvimento *Web* no Eclipse. Na primeira parte, detalhamos, passo a passo, a instalação de um conjunto de ferramentas para desenvolvimento *Web* (Java SE 6.0 SDK, Apache Tomcat 5.5, Eclipse 3.2 com Web Tools Platform, Ivy e IvyDE) e criamos um projeto para o desenvolvimento de uma aplicação *Web* (*WebApp*) de controle de empréstimos de livros chamada Bookshelf, a partir de um projeto em branco. Neste processo, entendemos um pouco mais sobre os *frameworks* envolvidos e preparamos a base para o desenvolvimento do nosso projeto.

Neste artigo desenvolveremos várias funcionalidades do Bookshelf, demonstrando como o desenvolvimento com *frameworks* promove uma melhor organização dos artefatos de código, além de promover uma maior agilidade no desenvolvimento. Ao final, teremos uma aplicação funcional, pronta para ser refinada.

Incrementando nossa Aplicação *Web*

Antes de implementar funcionalidades, podemos já dar uma cara mais "profissional" ao nosso *site*. Prepararemos nossa *WebApp* para o desenvolvimento de todas as suas funções com duas tarefas simples: aplicação de um leiaute gráfico e internacionalização das mensagens.

Melhorando a Aparência

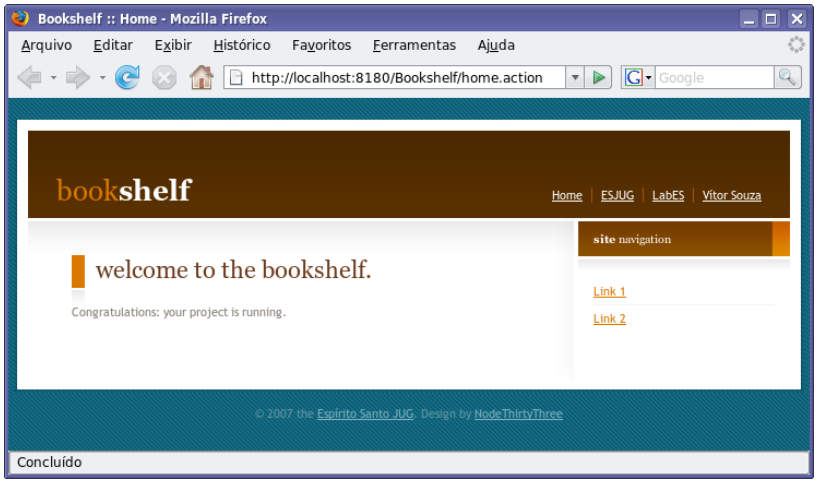

Figura 1. *Aparência da WebApp após aplicação do leiaute Bookish.*

Quando desenvolvemos aplicações *Web*, em geral um *webdesigner* ou uma equipe de *webdesign* fica responsável pela criação da aparência do *site*. Para nosso pequeno projeto, no entanto, resolvi fazer uma visita ao *Open Souce Web Design* (OSWD – ver referências) e pegar um pronto.

Efetuando uma busca pela palavra-chave *book*, encontrei o leiaute *Bookish*, baixei-o, copiei suas imagens para a pasta `images`, a folha de estilo para `stylesheets` (ambas em `WebContents/resources`) e seu código HTML para o meu decorador *default*, adaptando as referências às imagens e à folha de estilo quando necessário. A figura 1 mostra a aparência da aplicação após esta adaptação. Conhecimento básico de HTML e CSS já é suficiente para efetuar esta tarefa. No entanto, para casos mais complicados, a separação de leiaute e funcionalidade promovidas pelo SiteMesh e FreeMarker permitem que profissionais de *webdesign* cuidem desta parte, deixando os programadores e projetistas por conta do desenvolvimento das funcionalidades.

Internacionalização

O Struts[2] provê facilidades de internacionalização (i18n) para que construamos aplicações flexíveis, que sirvam ou sejam extensíveis para diferentes povos. O projeto em branco inclui um pacote `com.opensymphony.xwork2` com dois arquivos: `ActionSupport.properties` e `ActionSupport_pt_BR.properties`. Estes arquivos contém as mensagens que poderão ser exibidas em todas as páginas.

A explicação é simples: o Struts[2] procura arquivos `.properties` (conhecidos como *resource bundles*) com o mesmo nome da ação executada ou de uma de suas superclasses. Como geralmente nossas classes de ação estendem, direta ou indiretamente, a classe `com.opensymphony.xwork2.ActionSupport`, as mensagens destes dois *bundles* estarão disponíveis nas execuções de todas elas. O sufixo `_pt_BR` indica que o *bundle* contém mensagens em português do Brasil. Ele será escolhido se a localização da

WebApp for esta.

Adicione diversos textos ao arquivo `ActionSupport.properties`, como na listagem 1. Em seguida, substitua os textos no decorador e na página `home.ftl` pela *tag* `<@s.text />` do Struts[2], indicando na propriedade `name` o nome da chave da mensagem que você deseja exibir. Por exemplo, substitua a mensagem de boas vindas pela *tag* `<@s.text name="text.welcome" />`. As mensagens `link.*` devem ser colocados no lugar dos *links*, na coluna direita do *site*.

```
# Global i18n messages in english.
text.sitenavigation01 = Site
text.sitenavigation02 = Navigation

# Links to the WebApp's features.
link.registerUser = Register User
link.registerBook = Register Book
link.searchBooks = Search Books
link.listAvailableBooks = List Available Books
link.listBorrowedBooks = List Borrowed Books

# Text for the home page.
text.welcome = Welcome to the Bookshelf
text.home01 = The Bookshelf controls members and books from the Software Engineering Lab at UFES.
text.home02 = If you wish to borrow a book from the lab library, please register yourself and \
use the "Search Book" feature.
```

Listagem 1. Arquivo src/com/opensymphony/xwork2/ActionSupport.properties.

Em seguida, experimente copiar o conteúdo da listagem 1 para o arquivo `ActionSupport_pt_BR.properties`, traduzi-lo para o português e executar novamente a aplicação. Toda a interface será traduzida automaticamente para o português do Brasil, graças à configuração `struts.locale` no arquivo `struts.properties` (listagem). Experimente!

Implementando o Cadastro de Membros

Agora que entendemos um pouco mais dos *frameworks* envolvidos na arquitetura de nosso sistema e já preparamos nossa aplicação com relação à decoração e internacionalização, vamos implementar algumas funcionalidades do Bookshelf. Começaremos pelo cadastro de membros: gostaríamos de registrar os membros do laboratório para, posteriormente, cadastrar os livros, os empréstimos e, finalmente, saber com quem estão os livros que não estão no laboratório.

A princípio, criaremos uma classe de domínio para representar um membro do laboratório e registraremos seu nome, telefone, e-mail e data de cadastro. Em seguida, implementaremos a persistência desta classe no banco de dados. Finalmente, criaremos a ação e a classe de aplicação que implementarão as funcionalidade de cadastro.

O Domínio

Começando pela classe de domínio, a listagem 2 mostra um trecho do código-fonte da classe `Member`, que representa os membros do laboratório que podem pegar livros emprestado. Anotações à parte, a classe é simples, com alguns atributos, métodos *get* e *set*, um método de cópia (que será usado em nosso cadastro) e um método de comparação para fins de ordenação. Este tipo de classe é conhecido como POJO, abreviação do termo *Plain Old Java Object*, "velho e simples objeto Java" em inglês.

As anotações indicam como mapear esta classe para o banco de dados. Isso será utilizado pelo *framework* Hibernate quando pedirmos a ele que insira, recupere, atualize ou exclua um objeto desta classe do banco de dados. Estas anotações seguem o padrão de anotações de persistência do EJB 3.0 e estão amplamente documentadas no manual de referência do Hibernate Annotations (veja documentação do Hibernate nas referências deste artigo).

```java
package net.java.dev.esjug.bookshelf.domain;

@Entity
public class Member extends PersistentObjectSupport<Long, Long> implements Comparable<Member> {
  /** Full name. */
  @Basic @Column(length = 50)
  private String name;

  /** E-mail. */
  @Basic @Column(length = 100)
  private String email;

  /** Contact phone. */
  @Basic @Column(length = 200)
  private String phone;

  /** Register date. */
  @Temporal(TemporalType.DATE)
  private Date registerDate = new Date(System.currentTimeMillis());

  /** Getters e setters. */

  public void copy(Member member) {
     this.name = member.name;
     this.email = member.email;
     this.phone = member.phone;
  }

  /* @see java.lang.Comparable#compareTo(java.lang.Object) */
  public int compareTo(Member object) {
     return name.compareTo(object.name);
  }
}
```

Listagem 2. Classe de domínio que representa um membro do laboratório.

O leitor atento deve ter reparado que a classe Member herda da classe `PersistentObject<Long, Long>`. Esta classe é parte de um pacote de utilitários criado por mim e que segue junto ao projeto em branco para facilitar a criação de objetos de domínio. Fazer com que seus objetos de domínio herdem desta classe traz as seguintes vantagens:

- Um atributo de identificação (`id`) é herdado, já com as configurações de persistência, o que significa que todos os objetos de domínio já possuem um identificador (chave-primária);
- Um atributo de versão (`version`) também é herdado já configurado, permitindo que todos os objetos participem em transações com *locks* otimistas (seção 11.3 da referência do Hibernate);
- Um atributo de identificação universal (`uuid`) também é herdado, juntamente com implementações dos métodos `hashCode()` e `equals()` que utilizam o identificador universal, que é criado na construção do objeto e permanece com o mesmo até sua destruição. A idéia é baseada no artigo de Jason Carreira "Hibernate, null unsaved value, and hashcode: A story of pain and suffering" (veja nas referências).

Para concluir, indicaremos ao Hibernate que gostaríamos que a persistência dos objetos da classe `Member` fosse por ele gerenciada. Basta indicar em `hibernate.cfg.xml` que a classe é mapeada via anotações, como mostra o trecho na listagem 3.

```xml
<!-- Mappings. -->
<mapping class="net.java.dev.esjug.bookshelf.domain.Member" />
```

Listagem 3. Mapeamento da classe Member no arquivo hibernate.cfg.xml.

A Persistência

Nossa classe de domínio já foi mapeada, o que significa que agora basta que peçamos ao Hibernate que recupere, grave ou exclua objetos desta classe do banco de dados e ele o fará. Resta-nos uma questão: quem ficará responsável por chamar o Hibernate?

Uma forma de proceder é fazer com que as próprias classes que implementam os casos de uso (classes de aplicação) fiquem

responsáveis por isso. Não faremos desta forma por dois motivos principais: (a) a implementação do caso de uso ficaria atrelada ao Hibernate e (b) seria mais difícil realizar testes unitários nas classes de aplicação.

A solução adotada é um padrão bastante conhecido chamado *Data Access Object*, ou DAO. Cada classe de domínio possui um objeto responsável pela sua persistência. Em nosso caso, criamos uma interface `MemberDAO` (listagem 4) e uma implementação `HibernateSpringMemberDAO` (listagem 5) que utiliza a integração Spring-Hibernate para efetuar operações de acesso a dados.

```
package net.java.dev.esjug.bookshelf.persistence;

public interface MemberDAO extends BaseDAO<Member, Long> { }
```

Listagem 4. Interface DAO para a classe Member.

```
package net.java.dev.esjug.bookshelf.persistence;

public class HibernateSpringMemberDAO extends HibernateSpringBaseDAO<Member, Long> implements
MemberDAO {
  /* @see net.java.dev.esjug.util.persistence.HibernateSpringBaseDAO#getDomainClass() */
  @Override
  protected Class getDomainClass() {
     return Member.class;
  }
}
```

Listagem 5. Implementação DAO para a classe Member.

Mais uma vez, utilizamos aqui também classes de um utilitário de persistência que é distribuído juntamente com o projeto em branco. A classe `HibernateSpringBaseDAO` contém métodos para recuperação, gravação e exclusão de objetos persistentes, bastando que indiquemos a classe do objeto e de sua propriedade identificadora. Esta classe estende `HibernateDaoSupport`, uma classe do Spring preparada para delegar ao Hibernate todas as operações de persistência, bastando que seja provida a ela uma sessão do Hibernate.

E é exatamente isso que fazemos em seguida, editando o arquivo `applicationContext-core.xml` e, aproveitando o código comentado já existente no projeto em branco, declaramos nosso DAO e sua dependência à fábrica de sessões do Hibernate que ele usará para criar as sessões que necessitar. A listagem 6 mostra esta configuração. O *bean* `sessionFactory` foi declarado no `applicationContext-resource.xml`, apresentado na primeira parte desta série de artigos.

```
<!-- DAO class for Member objects. -->
<bean id="memberDAO" class="net.java.dev.esjug.bookshelf.persistence.HibernateSpringMemberDAO">
  <property name="sessionFactory"><ref bean="sessionFactory" /></property>
</bean>
```

Listagem 6. Configuração do DAO no arquivo applicationContext-core.xml.

A Aplicação

Se já temos nossa classe de domínio e um DAO para ela, podemos implementar o caso de uso que desejamos: cadastro de membros. Para não fugir à regra, usaremos aqui, também, classes utilitárias já empacotadas no projeto em branco. Veja a interface do serviço de cadastro e sua implementação nas listagens 7 e 8.

```
package net.java.dev.esjug.bookshelf.application;

public interface RegisterMemberService extends CrudService<Member, Long> { }
```

Listagem 7. Interface de aplicação para o caso de uso "Cadastrar Membros".

```
package net.java.dev.esjug.bookshelf.application;

public class RegisterMemberServiceImpl extends CrudServiceSupport<Member, Long>
                                                    implements RegisterMemberService {
  private MemberDAO memberDAO;

  public void setMemberDAO(MemberDAO memberDAO) {
     this.memberDAO = memberDAO;
  }

  @Override
  protected Member doCreate(Member valueObject) {
     Member object = new Member(valueObject.getName(), valueObject.getEmail(),
                                                                    valueObject.getPhone());
     memberDAO.save(object);
     return object;
  }

  @Override
  protected Member doDelete(Long id) {
     Member object = memberDAO.retrieveById(id);
     memberDAO.delete(object);
     return object;
  }

  @Override
  protected Member doRetrieve(Long id) {
     Member object = memberDAO.retrieveById(id);
     return object;
  }

  @Override
  protected Member doUpdate(Long id, Member valueObject) {
     Member object = memberDAO.retrieveById(id);
     object.copy(valueObject);
     memberDAO.save(object);
     return object;
  }

  public Collection<Member> list() {
     return new TreeSet<Member>(memberDAO.retrieveAll());
  }
}
```

Listagem 8. Classe de aplicação para o caso de uso "Cadastrar Membros".

A interface `CrudService` define vários métodos que um serviço de cadastro deve prover. Sua implementação `CrudServiceSupport` implementa alguns e, seguindo o padrão de projeto *Template Method*, delega para a classe concreta que estendê-la trechos destas implementações. O nome "Crud" vem da sigla em inglês que significa "Criar, Recuperar, Atualizar e Excluir" (*Create, Retrieve, Update and Delete*).

A classe `RegisterMemberServiceImpl` deve, portanto, implementar os métodos `doCreate()`, `doRetrieve()`, `doUpdate()`, `doDelete()` e `list()` que efetuam, respectivamente, a criação, recuperação de um, atualização, exclusão e recuperação de todos, relativos à classe `Member`.

Para efetuar todas estas operações, a classe de serviço precisará de acesso aos dados dos membros, dependendo de uma instância de um DAO da classe `Member`. Provemos este DAO a ela declarando-o como um atributo, adicionando um método `setter` e configurando o Spring, novamente no arquivo `applicationContext-core.xml`. A listagem 9 mostra esta configuração. Repare que o *bean* `memberDAO` que havíamos configurado anteriormente foi definido como dependência na propriedade `memberDAO`, que é o mesmo nome do atributo de `RegisterMemberServiceImpl` que deverá receber o DAO para o bom funcionamento da classe.

```
<!-- Service class for the 'Register Member' use case. -->
<bean id="registerMemberService"
                    class="net.java.dev.esjug.bookshelf.application.RegisterMemberServiceImpl">
  <property name="memberDAO"><ref local="memberDAO" /></property>
</bean>
```

Listagem 9. Configuração da classe de serviço no arquivo applicationContext-core.xml.

Aqui vale uma nota sobre a configuração do Spring: se o *bean* ao qual nos referenciamos foi definido no mesmo arquivo de configuração, podemos usar a referência local (`<ref local="" />`), como no caso do serviço de cadastro. Caso ele tenha sido declarado em outro arquivo do Spring, a referência utilizada é `<ref bean="" />`, como no caso do DAO, anteriormente.

O Controle

Nossa lógica de negócio encontra-se, neste momento, completamente implementada. Precisamos, agora, implementar a interface com o usuário, que desmembramos em controle e visão. O controle faz a mediação que desacopla a interface da aplicação permitindo, assim, que diferentes interfaces com o usuário (aplicativos gráficos, páginas *Web*, MIDlets, etc.) possam utilizar a mesma implementação dos casos de uso (e conseqüentemente as mesmas classes de domínio e persistência), sem alterações.

Neste pacote, o Struts[2] possui o papel principal. Criaremos, portanto, uma classe de ação que será acionada por ele quando acessarmos um determinado *link* ou enviarmos um formulário especifico. Esta classe de ação utilizará do mesmo utilitário de CRUD usado na camada de aplicação. Veja na listagem 10 a implementação da ação de cadastro de membros.

```
package net.java.dev.esjug.bookshelf.controller;

public class RegisterMemberAction extends CrudAction<Member, Long> {
  /** Service class for the Register Member use case. */
  private RegisterMemberService registerMemberService;

  public void setRegisterMemberService(RegisterMemberService registerMemberService) {
     this.registerMemberService = registerMemberService;
  }

  /* @see net.java.dev.esjug.util.frameworks.crud.CrudAction#createModel() */
  @Override
  protected Member createModel() {
     return new Member();
  }

  /* @see net.java.dev.esjug.util.frameworks.crud.CrudAction#getAplCrud() */
  @Override
  protected CrudService<Member, Long> getAplCrud() {
     return registerMemberService;
  }
}
```

Listagem 10. Configuração da classe de serviço no arquivo applicationContext-core.xml.

A classe `CrudAction` implementa a maior parte do código de tratamento dos dados recebidos via Struts[2]. A classe que dela herda toda esta funcionalidade pronta deve implementar apenas dois métodos: um que retorna um objeto vazio da classe que se deseja cadastrar e outro que retorna a classe de aplicação que efetuará as operações de cadastro.

Para que `RegisterMemberAction` consiga retornar uma instância da classe de aplicação de cadastro ela deve receber do Spring esta instância como dependência. Diferentemente das dependências do DAO e da classe de aplicação, as ações do Struts[2] tiram vantagem da integração do mesmo com o Spring e basta que declarem a dependência na classe com o mesmo nome do *bean* definido na configuração e prover, como de costume, o método de atribuição (*setter*) para ele. Veja na listagem 9 que o *bean* possui o mesmo nome da propriedade: `registerMemberService`.

Com a classe pronta, precisamos registrá-la na configuração do Struts e criar as páginas que irão interagir com ela. A primeira parte você confere na listagem 11. Como temos uma mesma classe para várias ações, são declarados resultados globais e as quatro ações e seus respectivos métodos dentro de `RegisterMemberAction`. Um pacote é criado especialmente para o caso de uso, declarando o *namespace* `/registerMember`, o que significa que, para acessar o cadastro, adicionamos um *link* no menu do decorador para `${base}/registerMember/home.action`.

```
<!-- Register Member use case. -->
<package name="registerMember" extends="default" namespace="/registerMember">
  <global-results>
     <result name="success" type="redirect">home.action</result>
     <result name="listing">/WEB-INF/pages/registerMember/list.ftl</result>
     <result name="input">/WEB-INF/pages/registerMember/form.ftl</result>
     <result name="confirmation">/WEB-INF/pages/registerMember/confirm.ftl</result>
  </global-results>
  <action name="home" class="net.java.dev.esjug.bookshelf.controller.RegisterMemberAction" />
  <action name="save" class="net.java.dev.esjug.bookshelf.controller.RegisterMemberAction"
     method="executeSave" />
  <action name="confirm" class="net.java.dev.esjug.bookshelf.controller.RegisterMemberAction"
     method="executeConfirmation" />
  <action name="delete" class="net.java.dev.esjug.bookshelf.controller.RegisterMemberAction"
     method="executeDelete" />
</package>
```

Listagem 11. Configuração da ação registerMember no struts.xml.

As páginas que interagem com a ação são escritas em FreeMarker, assim como nossa página inicial. A listagem 12 mostra o *template* que gera a primeira tela do cadastro. Esta tela exibe uma listagem dos membros existentes, permitindo que se cadastre um novo membro, altere os dados de algum membro e exclua o cadastro de um ou mais membros. A listagem 13 exibe o código da página de formulário, utilizada nos casos de criação de novos membros e de edição de dados de membros existentes. Finalmente, a listagem 14 traz o *template* de confirmação de exclusão, no qual pergunta-se ao usuário se ele tem certeza que deseja excluir um cadastro da base de dados.

```
<script language="JavaScript">
<!--
  /* Copies the id of an object to the update form. */
  function doUpdate(id) {
     document.forms['updateForm'].elements[0].value = id;
     document.forms['updateForm'].submit();
  }
  /* Activates the delete form. */
  function doDelete() {
     document.forms['deleteForm'].submit();
  }
// -->
</script>

<h2><@s.text name="link.registerMember" /></h2>

<@displayActionErrors title="${action.getText('text.errors')}" />

<div id="panel">
<div class="crud">
  <div class="crudToolbar">
     <a href="save!form.action"><img border="0" src="${base}/resources/images/iconNew16x16.png"
        width="16" height="16" alt="<@s.text name='form.button.create' />" title="<@s.text
        name='form.button.create' />"></a>
     <a href="javascript:doDelete();"><img border="0"
        src="${base}/resources/images/iconDelete16x16.png" width="16" height="16" alt="<@s.text
        name='form.button.delete' />" title="<@s.text name='form.button.delete' />"></a>
  </div>

  <div class="crudBody">
     <#-- Check if there are any existing objects. -->
     <#if (objects?exists && objects?has_content)>
        <@s.form action="confirm" method="post" theme="simple" name="deleteForm">
        <table class="crud">
        <tr>
           <td class="button"><img border="0" src="${base}/resources/images/iconDelete16x16.png"
              width="16" height="16" alt="<@s.text name='text.deletion' />" title="<@s.text
              name='text.deletion' />"></td>
           <th><@s.text name="form.label.name" /></th>
           <th><@s.text name="form.label.email" /></th>
```

```
            <th><@s.text name="form.label.phone" /></th>
            <th><@s.text name="form.label.registerDate" /></th>
            <td class="button"> </td>
         </tr>
         <#list objects as obj>
            <tr>
               <td class="button"><@s.checkbox name="ids" fieldValue="${obj.id}" /></td>
               <td>${obj.name}</td>
               <td>${obj.email}</td>
               <td>${obj.phone}</td>
               <td>${obj.registerDate?date?string.medium}</td>
               <td class="button"><a href="javascript:doUpdate(${obj.id});"><img border="0"
                  src="${base}/resources/images/iconEdit16x16.png" width="16" height="16"
                  alt="<@s.text name='form.button.update' />" title="<@s.text
                  name='form.button.update' />"></a></td>
            </tr>
         </#list>
         </table>
         </@s.form>
      <#else>
         <p><@s.text name="text.noObjects" /></p>
      </#if>
   </div>

   <#if (objects?exists && objects?has_content)>
   <div class="crudToolbar">
      <a href="save!form.action"><img border="0" src="${base}/resources/images/iconNew16x16.png"
         width="16" height="16" alt="<@s.text name='form.button.create' />" title="<@s.text
         name='form.button.create' />"></a>
      <a href="javascript:doDelete();"><img border="0"
         src="${base}/resources/images/iconDelete16x16.png" width="16" height="16" alt="<@s.text
         name='form.button.delete' />" title="<@s.text name='form.button.delete' />"></a>
   </div>
   </#if>
</div>
</div>

<!-- Update form: redirects to the input form that updates na object. -->
<@s.form action="save!form" method="post" theme="simple" name="updateForm">
   <@s.hidden name="model.id" />
</@s.form>
```

Listagem 12. Trecho da página de listagem de membros.

Repare na utilização diretivas (`<#if />`, `<#list />`), interpolações (`${obj.id}`, `${obj.name}`) e built-ins (`${obj..registerDate?date?string.medium}`). O FreeMarker provê uma linguagem relativamente simples e ao mesmo tempo poderosa para criação de modelos que são transformados em páginas HTML.

Estas páginas, assim como a página inicial, fazem uso de texto internacionalizado. *Resource bundles* podem ser providos para várias línguas diferentes, especificando os textos que devem aparecer nos diversos pontos do modelo. O arquivo `RegisterMemberAction.properties` é exibido na listagem 15 para ilustrar. Como exercício, sugerimos que o leitor crie o arquivo com o texto em português do Brasil.

```
<h2><@s.text name="link.registerMember" /></h2>

<div id="panel">
<div align="center">
  <@s.form action="save" method="post">
     <@s.hidden name="model.id" />
     <@s.textfield label="%{getText('form.label.name')}" name="model.name" size="40" />
     <@s.textfield label="%{getText('form.label.email')}" name="model.email" size="40" />
     <@s.textfield label="%{getText('form.label.phone')}" name="model.phone" size="30" />
     <tr><td colspan="2"><div align="right">
     <@s.reset value="%{getText('form.button.cancel')}"
        onclick="document.location.href='home.action';" theme="simple" />
     <@s.submit value="%{getText('form.button.save')}" theme="simple" />
     </td></tr>
  </@s.form>
</div>
</div>
```

Listagem 13. Trecho da página de formulário.

```
<h2><@s.text name="link.registerMember" /></h2>

<div id="panel">
<p><@s.text name="text.confirmation" /></p>

<@s.form action="delete" method="post" theme="simple">
  <p><ul>
  <#list objects as obj>
     <@s.hidden name="ids" value="${obj.id}" />
     <li>${obj.name} (${obj.email});</li>
  </#list>
  </ul></p>

  <p> </p>

  <p align="center">
     <@s.reset value="%{getText('form.button.cancel')}"
        onclick="document.location.href='home.action';" />
     <@s.submit value="%{getText('form.button.delete')}" />
  </p>
</@s.form>
</div>
```

Listagem 14. Trecho da página de confirmação de exclusão.

```
# Page text.
text.errors = The following error(s) occurred
text.deletion = To delete a member, mark the checkbox and click on the delete button
text.noObjects = No members have been registered yet.
text.confirmation = Are you sure you want to delete the following members?

# Form labels.
form.button.create = New member
form.button.delete = Delete selected members
form.button.update = Change member information
form.button.cancel = Cancel
form.button.save = Save
form.label.name = Name
form.label.email = Email
form.label.phone = Contact phone
```

Listagem 15. Resource bundle com texto em inglês para páginas de cadastro.

Aprenda fazendo!

Após seguir todos os passos detalhados acima e construir o cadastro de membros, exercite o que você acabou de aprender implementando

o cadastro de livros! Você vai reparar que muitas das configurações e classes serão bastante parecidas... Cuidado! Não abuse do copiar & colar pois você pode esquecer algo para trás e ter muitas complicações posteriormente!

Lembre-se de sempre iniciar o banco de dados antes de testar sua aplicação *Web* e de acompanhar o log para detectar erros ocorridos durante a execução da *WebApp* (ambos explicados na parte I da série de artigos). Como dissemos no artigo anterior, o Console do Eclipse não mostra as causas reais dos erros, portanto confie apenas no *log* do Bookshelf!

Implementando a Pesquisa de Livros

Se você já implementou o cadastro de livros, agora já pode implementar uma funcionalidade de pesquisa que permitirá localizar um livro da biblioteca e saber se o mesmo está ou não disponível para empréstimo. Dividimos esta tarefa em duas partes: encontrar os livros e saber se estão disponíveis para empréstimo.

Encontrando os Livros

Para começar, adicionaremos na classe `Book` duas consultas do Hibernate, uma para pesquisa por nome do livro, outra para pesquisa por nome dos autores. A listagem 16 mostra como usar anotações para esta tarefa.

```
@Entity
@NamedQueries ({
  @NamedQuery(name="book.searchByName", query="from Book as book where book.name LIKE :name"),
  @NamedQuery(name="book.searchByAuthorName", query="from Book as book where book.authors
LIKE :name")
})
public class Book extends PersistentObjectSupport<Long, Long> implements Comparable<Book> {
  ...
}
```

Listagem 16. Consultas declarada na classe Book como *named queries* para implementação da pesquisa de livros.

As consultas com nomes (*named queries*) são carregadas pelo Hibernate no momento da criação da fábrica de sessões. Durante a execução da aplicação, basta que eu me refira a ela pelo nome e forneça os parâmetros necessários (no caso, `:name`) e a consulta será efetuada.

Dois métodos são, então, adicionados ao DAO da classe BOOK (tanto na interface `BookDAO` quanto em sua implementação `HibernateSpringBookDAO`), um para cada tipo de pesquisa. Tais métodos que efetuam pesquisas específicas serão os únicos métodos que codificaremos em nossos DAOs, visto que os métodos básicos de recuperação, gravação e exclusão são herdados do DAO base.

A listagem 17 mostra estes dois métodos da classe `HibernateSpringBookDAO`. Utilizamos a integração do Spring com o Hibernate para efetuar uma procura com base em uma consulta por nome. Repare que o parâmetro `name` foi concatenado com o sinal `%` antes e depois, simbolizando que estamos buscando por `name` em qualquer parte do nome do livro ou de seus autores.

```
/* @see net.java.dev.esjug.bookshelf.persistence.BookDAO#searchByAuthorName(java.lang.String) */
@SuppressWarnings("unchecked")
public Collection<Book> searchByAuthorName(String name) {
  return getHibernateTemplate().findByNamedQueryAndNamedParam("book.searchByAuthorName",
     "name", "%" + name + "%");
}

/* @see net.java.dev.esjug.bookshelf.persistence.BookDAO#searchByName(java.lang.String) */
@SuppressWarnings("unchecked")
public Collection<Book> searchByName(String name) {
  return getHibernateTemplate().findByNamedQueryAndNamedParam("book.searchByName",
     "name", "%" + name + "%");
}
```

Listagem 17. Métodos da classe HibernateSpringBookDAO que efetuam a pesquisa de livros.

Com a persistência preparada, podemos passar para a camada de aplicação e implementar a classe que irá efetuar o caso de uso "Pesquisar Livros". Assim como no serviço de cadastro, o serviço de pesquisa também é composto de uma interface (`SearchBooksService`) e uma implementação (`ServiceBooksServiceImpl`, exibida na listagem 18). Podemos ver que a

classe de implementação depende do DAO da classe `Book` e implementa os dois tipos de pesquisa delegando a ele a atribuição de recuperar os dados do banco. Como no caso do cadastro, é necessário registrar a classe de aplicação no Spring, o que é mostrado na listagem 19.

```
package net.java.dev.esjug.bookshelf.application;

public class SearchBooksServiceImpl implements SearchBooksService {
  /** DAO for Book objects. */
  private BookDAO bookDAO;

  public void setBookDAO(BookDAO bookDAO) {
     this.bookDAO = bookDAO;
  }

  /* @see
net.java.dev.esjug.bookshelf.application.SearchBooksService#searchByAuthorName(java.lang.String)
*/
  public Collection<Book> searchByAuthorName(String name) {
     return new TreeSet<Book>(bookDAO.retrieveByAuthorName(name));
  }

  /* @see
net.java.dev.esjug.bookshelf.application.SearchBooksService#searchByName(java.lang.String) */
  public Collection<Book> searchByName(String name) {
     return new TreeSet<Book>(bookDAO.retrieveByName(name));
  }
}
```

Listagem 18. Classe de aplicação que implementa a funcionalidade de pesquisa de livros.

```
<!-- Service class for the 'Search Books' use case. -->
<bean id="searchBooksService"
     class="net.java.dev.esjug.bookshelf.application.SearchBooksServiceImpl">
  <property name="bookDAO"><ref local="bookDAO" /></property>
</bean>
```

Listagem 19. Declaração do serviço de pesquisa na configuração do Spring.

Finalizada a implementação do caso de uso, é hora de pensar na interface com o usuário, começando pelo controle. Acompanhe na listagem 20. Uma classe de ação é criada para receber dois parâmetros: o tipo de busca (propriedade `searchType`) e o termo que está sendo procurado (propriedade `name`). Após efetuada a busca (método `execute()`), o resultado é disponibilizado como uma coleção de livros (propriedade `books`). Desenvolvedores acostumados com Struts[2] percebem este fluxo com facilidade. É apenas uma questão de tempo para acostumar-se com a dinâmica dos *frameworks* MVC.

```java
package net.java.dev.esjug.bookshelf.controller;

public class SearchBooksAction extends ActionSupport {
  /** Represents the type of search by book name. */
  private static final int SEARCH_TYPE_BY_NAME = 1;

  /** Represents the type of search by authors name. */
  private static final int SEARCH_TYPE_BY_AUTHORS = 2;

  /** Service class for the 'Search Books' use case. */
  private SearchBooksService searchBooksService;

  /** Search type. */
  private int searchType = SEARCH_TYPE_BY_NAME;

  /** Search key: the name (of the book or the author). */
  private String name;

  /** Search result. */
  private Collection<Book> books;

  /* getter/setter para searchType e name, getter para books, setter para searchBooksService. */

  /* @see com.opensymphony.xwork2.ActionSupport#execute() */
  @Override
  public String execute() throws Exception {
     // Determines the type of the search and performs it.
     switch (searchType) {
     case SEARCH_TYPE_BY_NAME:
        books = searchBooksService.searchByName(name);
        break;
     case SEARCH_TYPE_BY_AUTHORS:
        books = searchBooksService.searchByAuthorName(name);
        break;
     }
     return SUCCESS;
  }
}
```

Listagem 20. Classe de ação que controla as requisições à pesquisa de livros.

Toda classe de ação deve ser registrada junto ao Struts[2] e esta não é exceção. A listagem 21 mostra a definição de um pacote com *namespace* `/searchBooks`, a ação `search` e seus dois resultados: um para o caso de queremos exibir o formulário de busca (`input`) e outro para o caso de sucesso na execução da pesquisa (`success`, implícito na segunda *tag* `<result />`).

```xml
<!-- Search Books use case. -->
<package name="searchBooks" extends="default" namespace="/searchBooks">
  <action name="search" class="net.java.dev.esjug.bookshelf.controller.SearchBooksAction">
     <result name="input">/WEB-INF/pages/searchBooks/input.ftl</result>
     <result>/WEB-INF/pages/searchBooks/success.ftl</result>
  </action>
</package>
```

Listagem 21. Configuração da ação de pesquisa no Struts[2].

Esta configuração nos diz para colocar em nosso decorador um *link* para a pesquisa apontando para `${base}/searchBooks/search!input.action`. Ou seja, as ações são acionadas por URLs que trazem em sua composição o *namespace* e o nome da ação, sempre terminando com `.action`. O termo `!input` indica que queremos executar o método input(), que a classe herda de `ActionSupport`. Este método não faz nada e retorna o resultado `INPUT`, permitindo que acessemos o formulário de pesquisa desta maneira.

Com o controle preparado, hora de criar as páginas usando o FreeMarker. Para o formulário de pesquisa, usamos as *tags* de formulário do Struts[2], como mostra a listagem 22. Repare que a string `#{'1' : 'By name', '2' : 'By author'}`, que representa as opções de tipos de busca, poderia ser colocada diretamente na propriedade `list` da *tag* `<@s.radio />`, não fosse um conflito de sintaxes entre o Struts[2] e o FreeMarker, que nos obriga a separá-la em uma concatenação.

```
<h2><@s.text name="link.searchBooks" /></h2>

<@s.form action="search" namespace="/searchBooks" method="post" name="frmSearchBooks">
  <#assign types="#" + "{'1' : 'By name', '2' : 'By author'}" />
  <@s.radio label="%{getText('form.label.searchType')}" name="searchType" list="${types}" />
  <@s.textfield label="%{getText('form.label.searchFor')}" name="name" />
  <@s.submit value="%{getText('form.button.search')}" />
</@s.form>
```

Listagem 22. Formulário de pesquisa de livros.

A listagem 23 mostra a página de resultados da pesquisa. Além das diretivas e interpolações do FreeMarker que já vimos no cadastro, esta página utiliza o recurso de folha de estilo CSS para gerar a tela exibida na figura 2.

```
<h2><@s.text name="link.searchBooks" /></h2>

<#if (books?exists && books?has_content)>
  <div id="bookSearchResult">
     <div class="info"><div class="title"><@s.text name="text.book" /></div></div>
     <div class="actions"><div class="title"><@s.text name="text.status" /></div></div>
     <div class="footer"></div>
  </div>

  <#list books as book>
     <#-- Determines if the book is available or not. -->
     <#assign status="Available" />

     <div id="bookSearchResult">
        <div class="info">
           <img border="0" src="${base}/resources/images/iconBook32x32.png" width="32"
              height="32" alt="${book.name}" title="${book.name}" align="left" />
           ${book.name}<br />
           <span class="authors">${book.authors}</span>
        </div>
        <div class="actions">
           <a href="#"><img border="0" src="${base}/resources/images/icon${status}16x16.png"
              width="16" height="16" alt="<@s.text name='text.book${status}' />" title="<@s.text
              name='text.book${status}' />" /></a>
        </div>
        <div class="footer"></div>
     </div>
  </#list>
<#else>
  <p><@s.text name="text.noBooksFound" /></p>
</#if>
```

Listagem 23. Página que exibe o resultado da pesquisa.

A primeira parte da pesquisa foi resolvida, no entanto o *status* mostrado na tela de resultados foi colocado como disponível (`<#assign status="Available" />`) sem verificarmos se há algum empréstimo. De fato, a funcionalidade “Pegar Livro Emprestado” não foi implementada e não existe nem a classe que representa um empréstimo. Vamos, então, à segunda parte da pesquisa, que já vai preparar o campo para as demais funcionalidades.

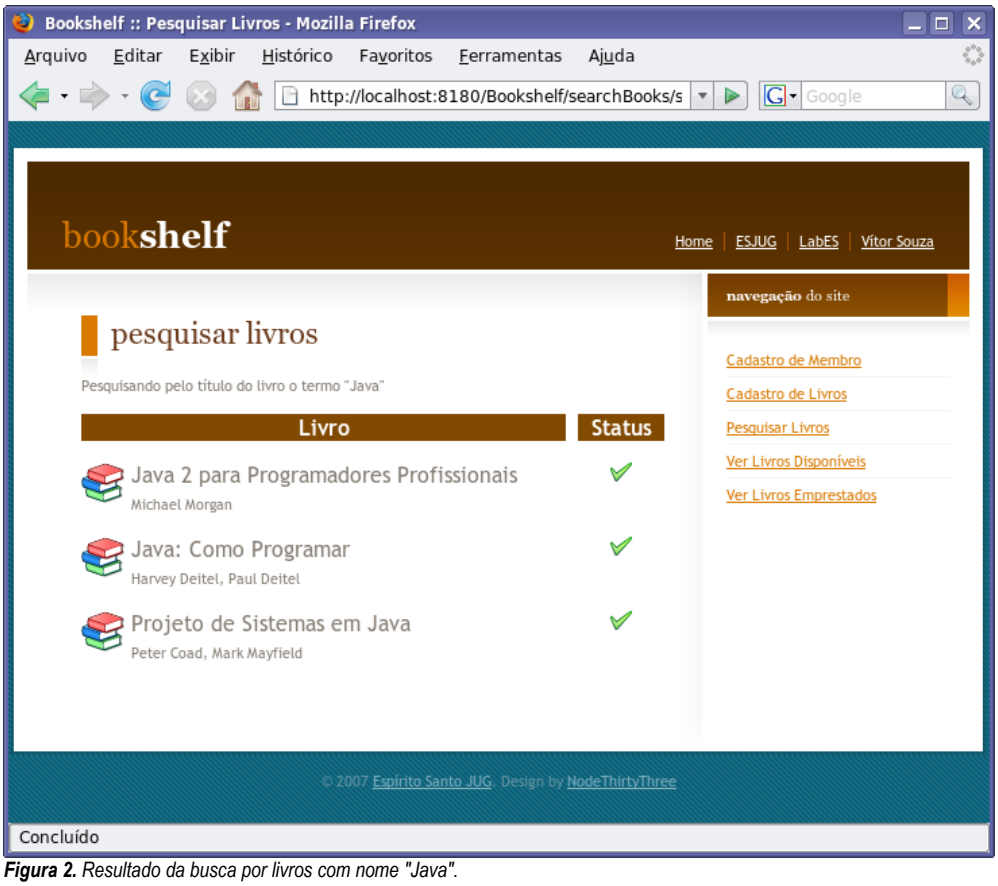

Figura 2. *Resultado da busca por livros com nome "Java".*

Verificando a Disponibilidade dos Livros

Para sabermos se um livro está ou não disponível, precisamos criar uma classe de domínio que represente o empréstimo, associando o membro ao livro e registrando a data de empréstimo e a data de devolução. A listagem 24 traz o código desta classe.

```
@Entity
@NamedQuery(name="loan.retrieveOpenLoans", query="from Loan as l where l.returnDate is null")
public class Loan extends PersistentObjectSupport<Long, Long> implements Comparable<Loan> {
  /** Member that borrowed the book. */
  @ManyToOne
  private Member borrower;

  /** Book that has been borrowed. */
  @ManyToOne
  private Book book;

  /** Date the book was borrowed. */
  @Temporal(TemporalType.DATE)
  private Date loanDate = new Date(System.currentTimeMillis());

  /** Date the book was returned. */
  @Temporal(TemporalType.DATE)
  private Date returnDate;

  public boolean isClosed() {
     return returnDate != null;
  }
}
```

Listagem 24. Trecho da classe Loan, que associa um membro a um livro que pegou emprestado.

Neste trecho de código, destacamos:

- A *named query* `loan.retrieveOpenLoans`, que recupera todos os empréstimos em aberto, ou seja, livros que ainda não foram devolvidos;
- A anotação `@ManyToOne`, do Hibernate Annotations, que faz o mapeamento de uma associação muitos-para-um da classe `Loan` à classe `Member` (e também à classe `Book`);
- A anotação `@Temporal`, que já havíamos utilizado nas classes `Member` e `Book`, que indica que queremos registrar apenas a data, desprezando a hora.

A listagem 25 traz um trecho da implementação do DAO da classe `Loan`, que além de herdar de `HibernateSpringBaseDAO` como os demais DAOs que construímos, implementa um método para recuperação dos empréstimos em aberto, utilizando a *named query* definida na classe `Loan`.

```
  /* @see net.java.dev.esjug.bookshelf.persistence.LoanDAO#retrieveOpenLoans() */
  @SuppressWarnings("unchecked")
  public Collection<Loan> retrieveOpenLoans() {
     return getHibernateTemplate().findByNamedQuery("loan.retrieveOpenLoans");
  }
```

Listagem 25. Método da classe HibernateSpringLoanDAO que recupera todos os empréstimos ainda não devolvidos.

Havendo uma classe que representa um empréstimo, precisamos alterar a classe `Book` para adicionar uma associação um-para-muitos com a classe `Loan`. Precisamos desta associação para, em seguida, implementar um método em `Book` que diz se o livro está ou não emprestado. O trecho de código criado para representar e gerenciar este relacionamento encontra-se na listagem 26.

```
package net.java.dev.esjug.bookshelf.domain;

@Entity
@NamedQueries ({
  @NamedQuery(name="book.searchByName", query="from Book as book where book.name LIKE :name"),
  @NamedQuery(name="book.searchByAuthorName", query="from Book as book where book.authors
LIKE :name")
})
public class Book extends PersistentObjectSupport<Long, Long> implements Comparable<Book> {
  /** Book loans. */
  @OneToMany(mappedBy = "book")
  private List<Loan> loans = new ArrayList<Loan>();

  /* Getter/setter privados para loans. */

  protected void addLoan(Loan loan) {
     loans.add(loan);
  }

  public Loan getCurrentLoan() {
     if ((loans == null) || (loans.size() == 0)) return null;
     Loan lastLoan = loans.get(loans.size() - 1);
     return lastLoan.isClosed() ? null : lastLoan;
  }

  public boolean isBorrowed() {
     return getCurrentLoan() != null;
  }
}
```

Listagem 26. Novos trechos da classe Book relacionados à associação com a classe Loan.

A classe `Book` agora possui uma associação com uma lista de `Loans`, que são mantidas na ordem em que foram feitas. Esta propriedade é mapeada como uma associação um-para-muitos no Hibernate, indicando que a propriedade `book` na classe `Loan` é o inverso dela (`@OneToMany(mappedBy = "book")`). Sempre que são mapeadas associações bidirecionais, o Hibernate exige que seja indicado o inverso no lado um-para-muitos.

Os métodos `getLoans()` e `setLoans()` foram colocados como privados para somente *frameworks* que realizam reflexão (como o Hibernate) poderem acessá-los. Os demais objetos não devem manipular esta coleção diretamente. Se há um empréstimo aberto para o

livro, pode ser obtido pelo método `getCurrentLoan()`. Para saber se o livro está emprestado, utilizamos o método `isBorrowed()`. Dados nossos requisitos atuais, não precisamos de nada mais.

Agora que temos um método da classe `Book` que diz se o livro está emprestado ou não, podemos alterar a página de resultado da pesquisa, substituindo a linha `<#assign status="Available" />` pelo código da listagem 27. O termo `book.borrowed` é substituído por uma chamada a `isBorrowed()`, no objeto `book`.

```
<#-- Determines if the book is available or not. -->
<#if book.borrowed>
  <#assign status = "NotAvailable" />
<#else>
  <#assign status = "Available" />
</#if>
```

Listagem 27. Trecho da página de resultado da pesquisa que verifica se o livro está ou não disponível.

Naturalmente, como ainda não implementamos as funcionalidades de pegar e devolver livros, os livros continuam marcados todos como disponíveis. Nosso próximo passo será implementar a funcionalidade de empréstimo partindo da tela do resultado da busca: um clique no ícone de disponível de um livro qualquer deve levar-nos a um formulário de confirmação do empréstimo daquele livro.

Aprenda fazendo!

Se conseguiu implementar o Bookshelf até aqui, não será nenhum desafio fazer por conta própria a listagem de livros disponíveis e a de livros emprestados. Note que a figura 2 já traz estes dois *links* na navegação do *site*, bastando que seja implementada uma ação similar à da busca, porém não são necessários parâmetros de busca. Basta escrever duas consultas do Hibernate (uma para os disponíveis e outra para os emprestados) e, em seguida, implementar métodos de aplicação e classes de ação para os mesmos. Você pode aproveitar a aplicação `SearchBooksService` e a classe de ação `SearchBooksAction` ou criar classes novas. Fica a critério do projetista!

Implementando o Empréstimo e a Devolução

Faltam apenas duas funcionalidades para completar nossa simples *WebApp*: o empréstimo e a devolução de livros. Vamos implementá-las usando uma abordagem um pouco diferente das anteriores. Se antes começamos a implementar o caso de uso pela classe de aplicação até chegar na interface, começaremos agora na interface, passaremos pela classe de ação até chegar na classe de aplicação.

Começando, portanto, pela página do resultado da pesquisa, o ícone que aparece à direita do nome do livro (veja na figura 2) poderia ser um *link* para uma página em que seria confirmado o empréstimo daquele livro. Alteramos a página em dois trechos, mostrados na listagem 28.

A página de resultados da pesquisa agora aponta um *link* para `${base}/borrowBook/return!input.action` para devolver livros emprestados e `${base}/borrowBook/borrow!input.action` para pegar emprestado livros disponíveis. Precisamos, então, declarar estas ações na configuração do Struts[2], com seus resultados `input` apontando para a página de solicitação de empréstimo/devolução e com `success` direcionando a uma página que confirma que o empréstimo/devolução foram efetuados. Essa configuração é mostrada na listagem 29.

Podemos ver na configuração que uma mesma classe de ação irá implementar o controle para ambos os casos de uso (tanto empréstimo quanto devolução). No caso de empréstimo, o método `executeBorrow()` será chamado, no outro caso, será o método `executeReturn()`. Para construirmos nossa classe de ação, precisamos saber quais dados serão enviados e quais outros serão exibidos. As listagens 30 a 33 mostra as páginas de solicitação e confirmação do empréstimo e da devolução. A partir das referências a propriedades como `book`, `borrowerId`, `loan` e `members`, sabemos quais dados devem estar disponíveis na classe de ação, que está resumida na listagem .

```
<#-- Determines if the book is available or not. -->
<#if book.borrowed>
  <#assign status = "NotAvailable" />
  <#assign action = "borrowBook/return!input.action" />
<#else>
  <#assign status = "Available" />
  <#assign action = "borrowBook/borrow!input.action" />
</#if>

...

<div class="actions">
  <a href="${base}/${action}?book.id=${book.id}"><img border="0"
     src="${base}/resources/images/icon${status}16x16.png" width="16" height="16" alt="<@s.text
     name='text.book${status}' />" title="<@s.text name='text.book${status}' />" /></a>
</div>
```

Listagem 28. Trechos da página de resultado da pesquisa alterados para chamar a ação de empréstimo ou devolução.

```
<!-- Borrow Book use case. -->
<package name="borrowBook" extends="default" namespace="/borrowBook">
  <action name="borrow" class="net.java.dev.esjug.bookshelf.controller.BorrowBookAction"
        method="executeBorrow">
     <result name="input">/WEB-INF/pages/borrowBook/borrowInput.ftl</result>
     <result>/WEB-INF/pages/borrowBook/borrowSuccess.ftl</result>
  </action>
  <action name="return" class="net.java.dev.esjug.bookshelf.controller.BorrowBookAction"
        method="executeReturn">
     <result name="input">/WEB-INF/pages/borrowBook/returnInput.ftl</result>
     <result>/WEB-INF/pages/borrowBook/returnSuccess.ftl</result>
  </action>
</package>
```

Listagem 29. Configuração das ações de empréstimo e devolução no Struts[2].

A página de solicitação de empréstimo (listagem 30) mostra os dados do livro e um formulário com uma lista de todos os membros cadastrados. No futuro, podemos implementar acesso via *login* e senha ao *site* de forma a não precisarmos mais escolher a pessoa que está pegando o livro emprestado. Para que todas estas informações apareçam na página, a classe de ação disponibiliza os atributos (e métodos) `book` (e `getBook()`) e `members` (e `getMembers()`).

```
<h2><@s.text name="text.borrowBook" /></h2>

<div id="bookSearchResult">
  <div class="info">
     <img border="0" src="${base}/resources/images/iconBook32x32.png" width="32" height="32"
        alt="${book.name}" title="${book.name}" align="left" />
     ${book.name}<br />
     <span class="authors">${book.authors}</span>
  </div>
  <div class="footer"></div>
</div>

<@s.form action="borrow" namespace="/borrowBook" method="post" name="frmBorrowBook">
  <@s.hidden name="book.id" />
  <@s.select label="%{getText('form.label.borrower')}" name="borrowerId" list="members"
     listKey="id" listValue="name" />
  <@s.submit value="%{getText('form.button.confirmLoan')}" />
</@s.form>
```

Listagem 30. Página de solicitação de empréstimo.

Submetido os campos `book.id` e `borrowerId`, a classe de ação deve capturar estes valores (por meio dos atributos `book` e `borrowerId` e os métodos `getBook().setId()` e `setBorrowerId()`) e efetuar o caso de uso. Para isso, ela recupera o membro e o livro a partir dos seus ids, utilizando os respectivos serviços de cadastro. No final, as informações do livro estão novamente disponíveis na propriedade `book` e informações sobre o empréstimo que acabou de ser feito estão no atributo `loan`. A página de

confirmação (listagem 31) exibe dados de ambos.

```
<h2><@s.text name="text.borrowBook" /></h2>

<div id="bookSearchResult">
  <div class="info">
     <img border="0" src="${base}/resources/images/iconBook32x32.png" width="32" height="32"
        alt="${book.name}" title="${book.name}" align="left" />
     ${book.name}<br />
     <span class="authors">${book.authors}</span>
  </div>
  <div class="footer"></div>
</div>

<p><@s.text name="text.borrowConfirmation">
  <@s.param value="'${loan.borrower.name}'" />
  <@s.param value="'${loan.loanDate?datetime?string.medium_short}'" />
</@s.text></p>
```

Listagem 31. Página de confirmação de empréstimo.

Passando para a devolução, a página de solicitação (listagem 32) exibe os dados do empréstimo aberto (obtidos da classe de ação via `getBook().getCurrentLoan()`) e monta um formulário com o `id` do empréstimo em um campo escondido e um botão de confirmação de devolução.

```
<h2><@s.text name="text.loanDetails" /></h2>

<div id="bookSearchResult">
  <div class="info">
     <img border="0" src="${base}/resources/images/iconBook32x32.png" width="32" height="32"
        alt="${book.name}" title="${book.name}" align="left" />
     ${book.name}<br />
     <span class="authors">${book.authors}</span><br />
     <span class="loanInfo">
     <@s.text name="text.loanInfo">
        <@s.param value="'${book.currentLoan.borrower.name}'" />
        <@s.param value="'${book.currentLoan.loanDate?date?string.medium}'" />
     </@s.text></span>
  </div>
  <div class="footer"></div>
</div>

<p> </p>

<h2><@s.text name="text.returnBook" /></h2>

<p><@s.text name="text.confirmReturn" /></p>

<@s.form action="return" namespace="/borrowBook" method="post" name="frmBorrowBook">
  <@s.hidden name="loan.id" value="${book.currentLoan.id}" />
  <@s.submit value="%{getText('form.button.confirmReturn')}" />
</@s.form>
```

Listagem 32. Página de solicitação de devolução.

Os dados do formulário são enviados para a ação que recupera o empréstimo que deve ser devolvido, devolve-o e retorna. Esta método utiliza apenas o serviço `borrowBookService`, que além de implementar os casos de uso de empréstimo e devolução, possui um método que recupera um empréstimo dado seu id. A página de confirmação da devolução (listagem 33) mostra dados do empréstimo obtidos da propriedade `loan` (e método `getLoan()`). Note que, como não foi atribuído à propriedade `book` um objeto, a diretiva `<#assign book = loan.book />` teve que ser inserida no início da página para atribuir à variável `book` o resultado de `getLoan().getBook()`.

```
<h2><@s.text name="text.borrowBook" /></h2>

<#assign book = loan.book />

<div id="bookSearchResult">
  <div class="info">
     <img border="0" src="${base}/resources/images/iconBook32x32.png" width="32" height="32"
        alt="${book.name}" title="${book.name}" align="left" />
     ${book.name}<br />
     <span class="authors">${book.authors}</span>
  </div>
  <div class="footer"></div>
</div>

<p><@s.text name="text.loanConfirmation">
  <@s.param value="'${loan.borrower.name}'" />
  <@s.param value="'${loan.loanDate?datetime?string.medium_short}'" />
  <@s.param value="'${loan.returnDate?datetime?string.medium_short}'" />
</@s.text></p>
```

Listagem 33. Página de confirmação de devolução.

```
package net.java.dev.esjug.bookshelf.controller;

public class BorrowBookAction extends ActionSupport {
  /* Classes de serviços. */
  private RegisterMemberService registerMemberService;
  private RegisterBookService registerBookService;
  private BorrowBookService borrowBookService;

  /* Parâmetros que são enviados para as páginas ou recebidos delas. */
  private Book book = new Book();
  private Long borrowerId;
  private Loan loan = new Loan();
  private Collection<Member> members;

  /* Getters e setters apropriados. */

  public Collection<Member> getMembers() {
     if (members == null) members = registerMemberService.list();
     return members;
  }

  public String input() throws Exception {
     book = registerBookService.retrieve(book.getId());
     return super.input();
  }

  public String executeBorrow() throws Exception {
     Member borrower = registerMemberService.retrieve(borrowerId);
     book = registerBookService.retrieve(book.getId());
     loan = borrowBookService.borrowBook(borrower, book);
     return SUCCESS;
  }

  public String executeReturn() throws Exception {
     loan = borrowBookService.viewLoan(loan.getId());
     borrowBookService.returnBook(loan);
     return SUCCESS;
  }
}
```

Listagem 34. Classe de ação para os casos de uso “Pegar Livro Emprestado” e “Devolver Livro”.

Agora que já temos nosso controlador, precisamos implementar o serviço de empréstimo de livros. Sabemos exatamente quais métodos precisamos, pois foram os métodos utilizados pela classe de ação: `borrowBook()`, `returnBook()` e `viewLoan()`. A implementação do serviço de empréstimo/devolução está na listagem 35 e sua configuração no Spring encontra-se na listagem 36.

```java
package net.java.dev.esjug.bookshelf.application;

public class BorrowBookServiceImpl implements BorrowBookService {
  /** DAO for Loan objects. */
  private LoanDAO loanDAO;

  public void setLoanDAO(LoanDAO loanDAO) {
     this.loanDAO = loanDAO;
  }

  public Loan borrowBook(Member borrower, Book book) {
     Loan loan = new Loan(borrower, book);
     loanDAO.save(loan);
     return loan;
  }

  public void returnBook(Loan loan) {
     loan.returnBook();
     loanDAO.save(loan);
  }

  public Loan viewLoan(Long loanId) {
     return loanDAO.retrieveById(loanId);
  }
}
```

Listagem 35. Serviço de empréstimo e devolução de livros.

```xml
<!-- Service class for the 'Borrow Book', 'List Borrowed Books' and 'Return Book' use cases. -->
<bean id="borrowBookService"
     class="net.java.dev.esjug.bookshelf.application.BorrowBookServiceImpl">
  <property name="loanDAO"><ref local="loanDAO" /></property>
</bean>
```

Listagem 36. Configuração do serviço de empréstimo/devolução no Spring.

Visto que a interface `LoanDAO` e sua implementação `HibernateSpringLoanDAO` já estão implementadas, bem como a classe `Loan` e suas *named queries*, finalizamos aqui a implementação das funcionalidades de empréstimo e devolução de livros. Deixamos a escrita dos arquivos `BorrowBookAction.properties` e `BorrowBookAction_pt_BR.properties` por conta do leitor. Quando for testar, lembre-se sempre de iniciar o banco de dados e monitorar o *log* antes.

Código-fonte da Aplicação Completa

O código-fonte do projeto Bookshelf está disponível para *download* no site do ESJUG (**`http://esjug.dev.java.net`**), na seção **`Artigos`**. Para compilar e implantar a aplicação Web, siga as instruções do artigo anterior para criar um novo *Dynamic Web Project*, copie os arquivos-fonte do Bookshelf para o diretório do projeto criado, baixe as dependências usando o Ivy, configure-as como *J2EE Module Dependencies* nas propriedades do projeto e implante-o no servidor Tomcat embutido.

Conclusões

Neste artigo, continuamos o projeto iniciado na primeira parte da série de artigos sobre desenvolvimento *Web* no Eclipse, implementando todas as funcionalidades de uma aplicação simples de controle de empréstimos de livros. Obviamente, há ainda muito espaço para melhorias em nossa *WebApp*. Pretendemos abordar algumas delas nos próximos artigos dessa série, como:

- Validação dos campos dos formulários;
- Controle de exceções, mostrando páginas de erro amigáveis ao invés de imprimir a exceção na tela ou retornar “Erro 500”;
- Implementação de *login* e senha para acesso;
- Envio de foto da capa do livro por meio de *upload* de arquivo.

Referências:

- Open Source Web Design: http://www.oswd.org;
- NodeThirtyThree (criadores do leiaute *Bookish*): http://www.nodethirtythree.com/;
- Nuvola (ícones utilizados no Bookshelf): http://icon-king.com/?p=15;
- "Hibernate, null unsaved value and hashcode: A story of pain and suffering" (Jason Carreira): http://www.jroller.com/page/jcarreira?entry=hibernate_null_unsaved_value_and;
- Grupo de Usuários de Java do Estado do Espírito Santo: http://esjug.dev.java.net/.