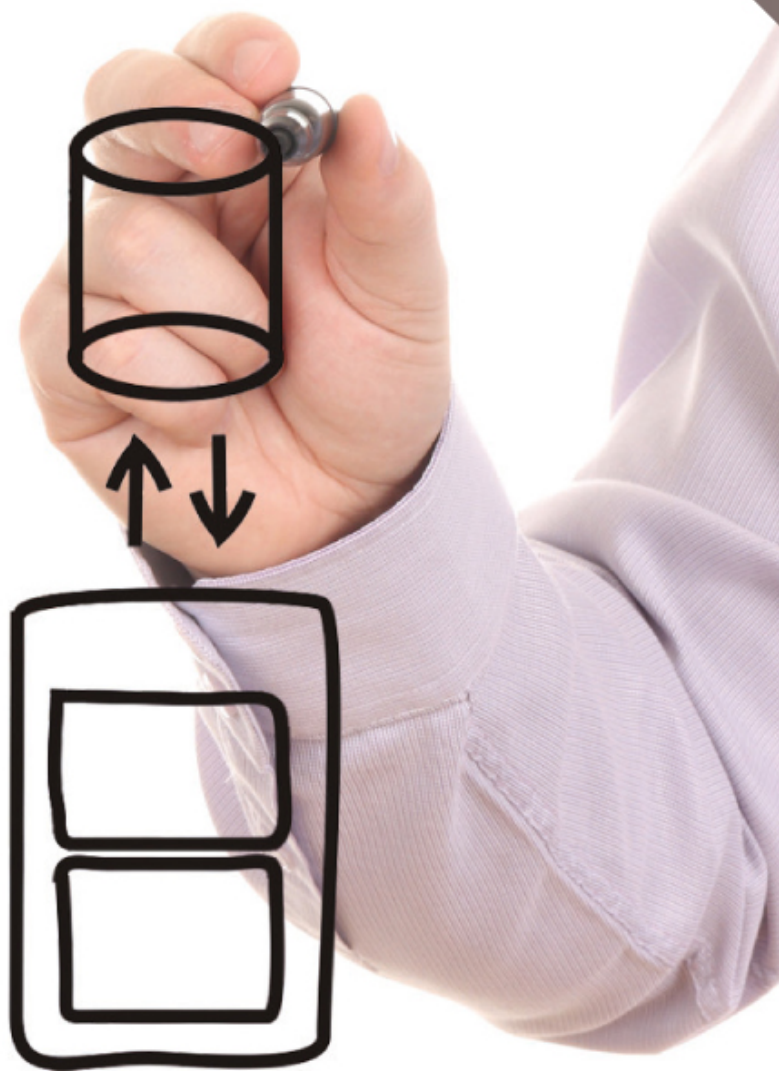


Aplicações Java para web com **JSF e JPA**



ao leitor, e portanto, motivando-o. A cada capítulo, iremos aprender conceitos que podem fazer evoluir a aplicação criada, olhando mais de perto para questões críticas, como boas práticas, performance e componentização. Vamos saber não apenas como fazer ou o que fazer, mas também o motivo das coisas serem de determinada forma. O objetivo desse livro não é ser uma receita de bolo para o leitor e sim, ser um material que desmistifique as questões e dúvidas que existem na grande maioria das aplicações que são desenvolvidas para a web.

Desenvolvedores sem experiências em desenvolvimento para a Web em Java aproveitarão esse livro de forma que, paralelamente à leitura, já consigam ir implementando seu próprio projeto e colocando em prática os novos conceitos e ideias aprendidas. O objetivo é que, ao final do livro, o leitor saiba como criar uma aplicação nova, desde a parte web, até o *back-end* com as regras de negócio e a camada de persistência. E que conheça quais as armadilhas em que ele pode cair e como se livrar delas, além de conseguir utilizar as boas práticas tanto do JSF quanto da JPA.

Para o desenvolvedor experiente, o livro contém diversos detalhes e boas práticas que podem fazer a diferença no uso das ferramentas indicadas. Por exemplo, como configurar o cache de segundo nível para a aplicação, ou como otimizar o consumo da sessão do JSF, entre outras situações que serão abordadas no decorrer da leitura.

O livro está estruturado em três partes, cada uma com seu objetivo específico:

Na primeira parte, teremos o primeiro contato com cada ferramenta, ainda sem nos preocuparmos muito com como cada uma funciona internamente. O principal objetivo é tornar familiar o funcionamento geral da JPA e do JSF, mostrando como integrá-los e construindo uma tela de cadastro junto com a camada de persistência.

Na segunda parte, o objetivo é apresentar as ferramentas de forma mais aprofundada, mostrando como funcionam e como usá-las da forma correta. É um conteúdo útil tanto para quem teve o primeiro contato na primeira parte do livro, quando para quem já desenvolve há muito tempo, mas nunca teve a oportunidade de explorar as ferramentas a ponto de saber como elas se comportam em situações específicas.

Na terceira e última parte, veremos como melhorar nossa aplicação em cima do conhecimento adquirido na segunda parte. O foco é principalmente usabilidade e performance. Exploraremos temas que foram vistos na segunda parte, como AJAX no JSF e *Lazy Load* na JPA, mas com enfoque muito mais específico em tirar o máximo proveito das ferramentas.

Agradecimentos

Podem considerar clichê, mas não há como iniciar qualquer tipo de agradecimento sem primeiro agradecer a Deus, pois sem ele não estaria aqui agradecendo a outras pessoas muito importantes, tanto na minha vida, quanto na escrita deste livro.

Gostaria de agradecer aos meus pais, José Cordeiro de Souza e Cleunice dos Santos Cordeiro, pela criação, pelos valores e pelo incentivo em todos os momentos da minha vida. Desde as tarefas da escola, cafés feitos altas horas da noite para aguentar a madrugada de trabalhos da faculdade e agora os períodos de “entocamento” para escrever este livro.

Ainda na seção família, não poderia deixar de agradecer minha irmã, Giselly Santos Cordeiro, por ser um exemplo de estudos e dedicação em uma época em que eu andava meio preguiçoso.

Um agradecimento muito especial à minha namorada, quase esposa, Dantiele de Freitas Queiróz que tem sido uma verdadeira parceira nesses oito anos que estamos juntos. Isso mesmo, oito anos de namoro :). Agradeço pela companhia nos momentos descontraídos, mas principalmente pela parceria nos momentos de muito trabalho e estudo. São meses sem poder curtir um final de semana tranquilo, sem sair para pescar, ou ficar horas fazendo qualquer outra coisa. Em vez de reclamar, era eu escrevendo em uma escrivantina e ela estudando em outra.

Agradeço também aos meus sogros, Wanderley e Iraci, pois quando estava na casa deles, além de ficar no meu canto escrevendo, ainda carregava a filha deles comigo. E apesar de toda brincadeira que se faz com sogro e sogra, eles sempre foram paizões comigo.

E não poderia terminar essa lista sem agradecer a dois caras em especial: Paulo Silveira e Adriano Almeida. Primeiramente ao Paulo pela confiança e pelo convite de escrever um livro, algo que eu nunca tinha feito. Admiro essa capacidade dele em transformar conhecimento em algo palpável, primeiro com o GUJ, depois a Caelum e agora a Casa do Código. E também um muito obrigado ao Adriano, que teve o

duro trabalho de transformar quilos de trechos de código e explicação técnica em algo leve e prazeroso de ler. Sem ele como editor, em vez de um livro, teria escrito um post gigante de blog.

Sumário

Contato inicial	1
1 Introdução	3
1.1 A evolução da integração entre Java e Bancos de dados	4
1.2 Como era a vida com o JDBC?	5
1.3 Diminuição da impedância através do mapeamento	9
1.4 Bibliotecas ORM e o Hibernate	10
1.5 Muitos frameworks ORM, como evitar o vendor lock-in?	12
1.6 O desenvolvimento web com Servlets e o padrão MVC	13
1.7 Tirando HTML do código Java com as JavaServer Pages	16
1.8 Organize seu código com a separação de responsabilidades	17
1.9 O MVC e os frameworks	17
1.10 Por que usar JSF?	22
1.11 O primeiro contato com o JSF	22
1.12 Nos próximos capítulos	24
2 Primeiros passos com a JPA	27
2.1 Definição do modelo	27
2.2 Configuração básica do persistence.xml	29
2.3 Escolha o dialeto do banco de dados	31
2.4 Automatização da criação e evolução das tabelas	32
2.5 Gravação do primeiro dado no banco de dados	33
2.6 Consultas simples no banco de dados com a JPA	37
2.7 Exclusão de dados com a JPA	39
2.8 O que mais vem pela frente?	40

3	Primeiros passos com o JSF	41
3.1	Onde e como defino telas com o JSF?	41
3.2	Criação da tela e dos inputs básicos de dados	42
3.3	commandButton, Managed Beans e a submissão de formulários	46
3.4	Passe dados da tela para o Managed Bean	48
3.5	Como receber os parâmetros direto no método	52
3.6	Gravação do automóvel no banco de dados	53
3.7	Liste os dados com o dataTable	54
3.8	Mas minha listagem está executando várias consultas no banco...	59
3.9	Exclusão de dados e o commandLink	60
3.10	O primeiro CRUD integrado	62
	 Domine as ferramentas	 63
4	Entendendo a JPA	65
4.1	O padrão de projetos Data Access Object, ou DAO	65
4.2	JPA não é um DAO genérico, é contextual	67
4.3	Ciclo de vida de um objeto na JPA	69
4.4	A JPA vai mais além do que um simples executar de SQLs	72
5	Como mapear tudo... ou nada!	73
5.1	Definições de entidade	75
5.2	Faça atributos não serem persistidos com o @Transient	77
5.3	Mapeie chaves primárias simples	78
5.4	Mapeie chaves compostas	82
5.5	A anotação @Basic	86
5.6	@Table, @Column e @Temporal	87
5.7	@Version e lock otimista	89
5.8	Relacionamentos muitos para um com @ManyToOne	90
5.9	@OneToOne	92
5.10	Relacionamentos bidirecionais	94
5.11	O que tem a ver o dono do relacionamento com a operação em cascata?	96
5.12	Organize melhor suas entidades e promova reaproveitamento com @Embeddable e @Embedded	100

5.13	Relacionamentos um para muitos com o @OneToMany e @ManyToOne	104
5.14	A importância do Lazy Loading, o carregamento preguiçoso	107
5.15	@Lob	108
5.16	@ElementCollection	109
5.17	Relacionamentos muitos para muitos com o @ManyToMany	110
5.18	Customize as colunas de relacionamentos com @JoinColumn e @JoinColumns	112
5.19	Configure as tabelas auxiliares com @JoinTable	114
5.20	Conclusão	115
6	Consultas com a JPQL e os problemas comuns na integração com o JSF	117
6.1	Filtre dados com a Java Persistence Query Language - JPQL	118
6.2	Como aplicar funções nas consultas	119
6.3	Onde estão meus joins?	121
6.4	Execute suas consultas e use parâmetros	122
6.5	Utilize funções de agregação	123
6.6	Faça sub-consultas com a JPQL	124
6.7	Agrupamentos e HAVING	125
6.8	Consultas complexas... resultados complexos?	125
6.9	Use o Select New e esqueça os arrays de Object	126
6.10	Organize suas consultas com Named Queries	127
6.11	Execute as Named Queries	129
6.12	Relacionamentos Lazy, N+1 Query e Join Fetch	130
6.13	Evite a LazyInitializationException com o OpenEntityManagerInView	133
6.14	O problema das N+1 consultas e como resolvê-lo	135
6.15	Foi bastante, mas não acabou...	136
7	Entenda o JSF e crie aplicações web	137
7.1	Se prepare para um mundo diferente, baseado em componentes . . .	137
7.2	A web stateless contra a web stateful	138
7.3	O ciclo de vida das requisições no JSF	140
7.4	Fase 1 - Criar ou restaurar a árvore de componentes da tela (Restore View)	140

7.5	Fase 2 - Aplicar valores da requisição na árvore de componentes (Apply Request Values)	142
7.6	Fase 3 - Converter e Validar (Validate)	142
7.7	Fase 4 - Atualizar o modelo (Update Model)	143
7.8	Fase 5 - Invocar ação da aplicação (Invoke Application)	144
7.9	Fase 6 - Renderizar a resposta (Render Response)	144
7.10	Aja sobre as fases do JSF com os PhaseListeners	146
7.11	Conheça os componentes do JSF	148
7.12	h:form	148
7.13	h:inputText e h:inputTextarea	149
7.14	h:inputSecret	150
7.15	h:inputHidden	151
7.16	h:selectOneMenu, f:selectItem e f:selectItems	151
7.17	h:selectOneRadio	152
7.18	h:selectOneListbox	153
7.19	h:selectManyMenu e h:selectManyListbox	153
7.20	h:selectManyCheckbox	154
7.21	h:selectBooleanCheckbox	155
7.22	Novidade do JSF 2.2: h:inputFile	156
7.23	h:panelGrid	157
7.24	h:panelGroup	158
7.25	h:outputText	158
7.26	h:outputLabel	159
7.27	h:outputFormat	160
7.28	h:outputScript e h:outputStylesheet	160
7.29	h:graphicImage	161
7.30	h:dataTable	161
7.31	ui:repeat	163
7.32	h:commandButton e h:commandLink	163
7.33	A diferença entre Action e ActionListener	165
7.34	Padronização no carregamento de recursos	166
7.35	Entenda os conversores nativos, o f:convertDateTime e o f:convertNumber	169
7.36	Conversores customizados	171

7.37	Conheça os validadores nativos	173
7.38	E quando os validadores nativos não fazem o que eu quero? Crie seus validadores	178
7.39	Novidade: JSF com GET e bookmarkable URLs	181
7.40	h:button e h:link	182
7.41	Regras de navegação	184
7.42	Entenda os escopos e saiba como e quando trabalhar com cada um	188
7.43	A curva de aprendizado do JSF	194
8	Validações simplificadas na JPA e no JSF com a Bean Validation	195
8.1	Trabalhe com os validadores e crie sua própria validação	196
8.2	Organize grupos de validação	198
8.3	A integração entre Bean Validation e JPA	199
8.4	A integração entre Bean Validation e JSF	200
	Desenvolvendo a fluência	203
9	Enriquecendo nossa aplicação JSF	205
9.1	Combos em cascata e commands em dataTable	206
9.2	Mostre mensagens com h:message e h:messages	211
9.3	Internacionalizando nossa aplicação	216
9.4	JSF e Ajax	219
9.5	Organize o código das telas com templates	226
9.6	Composite Components: criando nossos próprios componentes a partir de outros	231
9.7	Criar um jar com componentes customizados	240
10	Truques que podem aumentar a escalabilidade e a performance da sua aplicação	241
10.1	Utilize o cache de segundo nível da JPA	242
10.2	Faça cache de consultas	247
10.3	Colhendo estatísticas da nossa camada de persistência	251
10.4	Relacionamentos extra-lazy	254
10.5	Paginação virtual e real de dados	257

10.6	Utilizando Pool de conexões	262
10.7	Tornando o JSF mais leve com uso inteligente de AJAX	264
10.8	Quando possível, manipule componentes JSF no lado do cliente . . .	267
10.9	Considerações finais sobre otimizações	269
11	Conclusão	271
12	Apêndice: iniciando projeto com eclipse	275
12.1	Adicionando o tomcat no eclipse	275
12.2	Criando o projeto web	277
12.3	Baixando os jars do JSF	280
12.4	Configurando a JPA	280
	Índice Remissivo	284
	Bibliografia	285

Parte I

Contato inicial

Nessa primeira parte do livro teremos o primeiro contato com cada ferramenta, ainda sem nos preocuparmos muito com o funcionamento interno de cada uma.

CAPÍTULO 1

Introdução

Neste capítulo inicial, veremos um pouco da história por trás dessas duas bibliotecas, JPA e JSF. Qual o caminho percorrido até chegarmos no que temos hoje? Quais problemas essas ferramentas vieram solucionar? Em todos os casos, consideraremos o desenvolvimento orientado a objetos. A discussão dos benefícios da orientação a objetos é tema de diversos livros e não entraremos nesse mérito. Tanto JPA quanto JSF são ferramentas para projetos para esse paradigma. Veremos agora como cada uma delas colabora para um desenvolvimento mais produtivo.

Para que fique mais simples acompanhar os exemplos do livro, olhe o capítulo 12, que é um apêndice mostrando toda a montagem do ambiente utilizando a IDE eclipse. Além disso, o código do livro está disponível no github: <https://github.com/gscordeiro/faces-motors>.

1.1 A EVOLUÇÃO DA INTEGRAÇÃO ENTRE JAVA E BANCOS DE DADOS

A grande maioria das aplicações construídas atualmente, de alguma maneira, necessita se integrar com um banco de dados relacional disponibilizado em um servidor. Esse banco de dados, costumeiramente, era criado baseado em uma modelagem, como por exemplo, o diagrama entidade-relacionamento. Dessa forma, facilitava-se a visualização de todas as entidades que haveria na aplicação, bem como a análise de todo o contexto que a englobaria.

Com isso, podemos modelar uma aplicação de compra e venda de automóveis. No entanto, para não nos perdermos nos requisitos da aplicação e mantermos o foco nas ferramentas, vamos nos concentrar basicamente no cadastro e na recuperação dos dados de `Marca`, `Modelo` e `Automovel`.

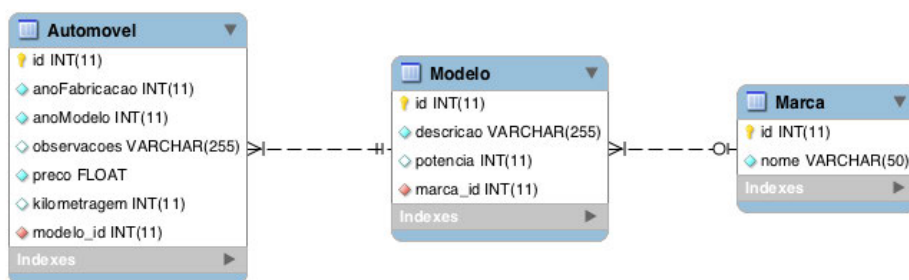


Figura 1.1: Diagrama entidade-relacionamento de Marca, Modelo e Automovel

Enquanto isso, no mundo orientado a objetos, estamos acostumados a representar as entidades envolvidas na aplicação por meio de um outro diagrama. Nesse caso, pelo diagrama de classes, que possui o mesmo objetivo da modelagem entidade-relacional: permitir uma visualização da aplicação num contexto global e de como todas as classes interagem entre si.

Uma possível abordagem para modelar a aplicação ilustrada na imagem 1.1 pode ser vista na figura 1.2.

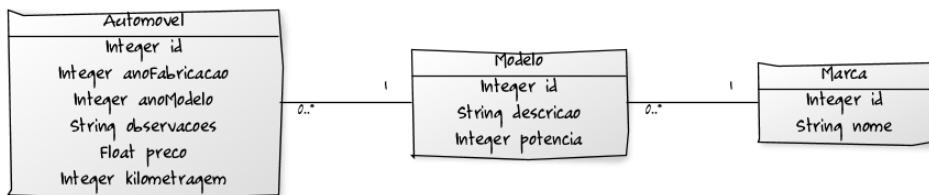


Figura 1.2: Diagrama das classes Marca, Modelo e Automovel

Apesar de serem voltados a paradigmas diferentes, é possível notar algumas similaridades entre ambos os diagramas. Sendo assim, qual a dificuldade de mapearmos nosso modelo orientado a objetos para o banco de dados relacional?

Uma possível diferença é que no modelo orientado a objetos temos as classes Automovel, Modelo e Marca. Essas classes possuem - além das propriedades simples, como nome, descrição e preço - relacionamento umas com as outras. Um exemplo disso é o automóvel que está ligado a um modelo, que está ligado a uma marca. Já o modelo entidade-relacionamento mostra um automóvel que tem um id de um modelo que por sua vez tem o id de uma marca. A diferença parece sutil ao analisarmos, mas na prática torna-se grande e pode se tornar bastante traiçoeira.

No mundo OO, objetos não carregam `ids` de outros objetos, eles possuem um vínculo com o outro objeto inteiro. Dessa maneira, um objeto pode ter listas de outros objetos e até estruturas de dados mais complexas, formadas por outras classes.

Quando vamos persisti-los em um banco de dados relacional, precisamos ajustar as diferenças existentes entre ambos os modelos. Some-se a isso o fato de que na orientação a objetos possuímos herança, polimorfismo, composição e diversas outras características que não estão presentes no modelo relacional. A essas diferenças fundamentais da forma dos objetos no mundo orientado a objetos e no mundo relacional, chamamos de **Impedância Objeto-Relacional** (*Impedance Mismatch*).

1.2 COMO ERA A VIDA COM O JDBC?

Durante algum tempo, a principal ferramenta que permitia aos desenvolvedores integrarem seus códigos Java com o banco de dados, possibilitando a manipulação das informações neles, era a *API JDBC* (*Java Database Connectivity*). Ela consiste de um conjunto de classes e interfaces que provêem uma forma de acesso aos bancos de dados, introduzidas através dos conhecidos *drivers*, necessários para a integração com o banco de dados.

Esses *drivers* fazem a comunicação entre a nossa aplicação e o banco de dados, comunicando-se em um protocolo que o próprio banco entenda.

Considerando uma entidade chamada `Automovel`, que contém algumas informações como o nome, ano de fabricação e cor, podemos ter uma classe Java que a representa:

```
public class Automovel {
    private Long id;

    private Integer anoFabricacao;
    private String modelo;
    private Integer anoModelo;
    private String marca;
    private String observacoes;

    // getters e setters se necessário
}
```

Nessa mesma aplicação, podemos ter uma interface que determina quais operações podem ser realizadas no banco de dados, a partir de um `Automovel`. Essa interface será implementada por classes cujo único objetivo é cuidar da persistência das informações e são conhecidas como DAO (*Data Access Object*).

```
public interface AutomovelDAO {

    void salva(Automovel a);
    List<Automovel> lista();

}
```

Então, podemos ter como implementação dessa interface, uma classe chamada `JDBCAutomovelDAO`, que implemente os métodos `salva` e `lista`.

```
public class JDBCAutomovelDAO implements AutomovelDAO {

    public void salva(Automovel a) {
        // código para salvar vem aqui
    }

    public List<Automovel> lista() {
        // código para listar vem aqui
    }
}
```



```
    }  
}
```

Para implementar o método salvar, através do JDBC, é preciso abrir a conexão com o banco de dados, escrever o SQL que irá executar, passar os dados do objeto para a query que será feita e assim por diante. Uma possível implementação para o método `salva` é a seguinte:

```
@Override  
public void salva(Automovel automovel) {  
    String sql = "insert into automoveis " +  
        "(anoFabricacao, anoModelo, marca, modelo, observacoes)" +  
        " values (?, ?, ?, ?, ?)";  
  
    Connection cn = abreConexao();  
  
    try {  
        PreparedStatement pst = null;  
        pst = cn.prepareStatement(sql);  
  
        pst.setInt(1, automovel.getAnoFabricacao());  
        pst.setInt(2, automovel.getAnoModelo());  
        pst.setString(3, automovel.getMarca());  
        pst.setString(4, automovel.getModelo());  
        pst.setString(5, automovel.getObservacoes());  
  
        pst.execute();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    } finally {  
        try {  
            cn.close();  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Nesse código é possível observar diversas complexidades, como o tratamento obrigatório da `SQLException` em dois momentos e a repetitiva invocação dos métodos `setInt` e `setString` para montar o comando.

Podemos ir além e ver a implementação do método `lista`, cujo objetivo é fazer uma consulta no banco de dados para devolver uma `List` de `Automovel`.

```
public List<Automovel> lista() {
    List<Automovel> automoveis = new ArrayList<Automovel>();
    String sql = "select * from Automovel";
    Connection cn = abreConexao();
    try {
        PreparedStatement pst = cn.prepareStatement(sql);
        ResultSet rs = pst.executeQuery();
        while( rs.next() ) {
            Automovel automovel = new Automovel();
            automovel.setId(rs.getLong("id"));
            automovel.setAnoFabricacao(rs.getInt("anoFabricacao"));
            automovel.setAnoModelo(rs.getInt("anoModelo"));
            automovel.setMarca(rs.getString("marca"));
            automovel.setModelo(rs.getString("modelo"));
            automovel.setObservacoes(rs.getString("observacoes"));

            automoveis.add(automovel);
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        try {
            cn.close();
        } catch (SQLException e) {
            throw new RuntimeException(e)
        }
    }
    return automoveis;
}
```

Esse código é ainda mais complexo que o da gravação de um novo `Automovel`. Isso se deve ao fato de que aqui precisamos retirar os dados do banco de dados, que foi devolvido no `ResultSet`, e atribuí-los ao objeto `automovel` criado dentro do `while`.

Note que escrevemos o código que trabalha tanto com a orientação a objetos e o relacional, transformando o dado que é adequado em um deles para o outro, fazendo os ajustes necessários.

Claro, o exemplo mostrado é trivial, pois é uma simples operação de consulta e inserção. Porém, como exercício, imagine se a consulta devolvesse informações de 3 tabelas diferentes, por meio da realização de operações de *joins* - a impedância nesse caso ficaria mais evidente ainda.

Fazer essa conversão do mundo OO para o relacional cada vez que precisamos salvar ou recuperar qualquer objeto do banco de dados exige tanto esforço de desenvolvimento, que acabamos gastando alguma energia com algo que não é o objetivo final do sistema. Não seria muito melhor se essa integração entre os dois paradigmas já estivesse pronta e não precisássemos nos preocupar com ela?

1.3 DIMINUIÇÃO DA IMPEDÂNCIA ATRAVÉS DO MAPEAMENTO

Vimos que, quando trabalhamos com o JDBC, misturamos dois paradigmas diferentes na aplicação Java: o orientado a objetos e o relacional. Mas não seria mais interessante se utilizássemos **um só paradigma** e, magicamente, o que fizéssemos nele fosse refletido no outro? Foi justamente com esse intuito que começaram a surgir bibliotecas para permitir que, em vez de misturarmos o paradigma relacional no nosso código Java, possamos apenas nos preocupar com objetos. Algo como:

```
public void salva(Automovel automovel) {  
    conexao.save(automovel);  
}
```

No código anterior, repare que apenas trabalhamos com objetos e que, quando invocado, o método `save` se encarrega de fazer todo o SQL necessário, com os relacionamentos necessários para nós. Muito melhor, não?

Mas como o método `save` sabe em qual tabela o `insert` terá que ser feito? Quais colunas terão que ser adicionadas ao SQL? Qual é o *datatype* de cada uma das colunas? Essas são algumas das questões que podem vir à mente nesse instante.

De alguma maneira, ele precisará ter essa informação. Ou seja, deverá haver uma forma de indicar que objetos da classe `Automovel` terão registros inseridos na tabela `automoveis`. Essa definição chamamos de mapeamento, que no caso, é uma ligação feita entre os dois modelos diferentes, o relacional e o orientado a objetos. Temos, então, o *Mapeamento Objeto Relacional* (*Object Relational Mapping* - ORM).

1.4 BIBLIOTECAS ORM E O HIBERNATE

Bibliotecas que fazem o trabalho descrito acima chamamos de frameworks ORM. Temos diversas implementações desses frameworks a disposição, como o Eclipse-Link, Hibernate, OpenJPA entre outros. Dentre esses, muito provavelmente, o mais famoso e mais usado é o Hibernate.

Com ele, é possível indicar que um `Automovel` vai poder ser persistido no banco de dados, somente pela introdução da anotação `@javax.persistence.Entity` na classe:

```
@Entity
public class Automovel {

    private Long id;

    private Integer anoFabricacao;
    private String modelo;
    private Integer anoModelo;
    private String marca;
    private String observacoes;

    // getters e setters se necessário
}
```

Além disso, podemos indicar qual dos atributos representa a chave primária da tabela. No caso, é o atributo `id`, bastando anotá-lo com `@Id` e `GeneratedValue` caso queiramos um valor auto-incremento para ele. Durante o livro aprenderemos o completo significado dessas anotações e de várias outras:

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

Pronto, isso feito, basta persistir o objeto através do próprio Hibernate. Para isso, podemos implementar novamente a interface `AutomovelDAO`, porém agora, para o Hibernate.

```
public class HibernateAutomovelDAO implements AutomovelDAO {

    public void salva(Automovel a) {
        Session s = abreConexao();
        Transaction tx = s.beginTransaction();
```

```
s.save(a);

tx.commit();
s.close();
}

public List<Automovel> lista() {
    Session s = abreConexao();

    List<Automovel> automoveis =
        s.createQuery("from Automovel").list();

    s.close();

    return automoveis;
}
}
```

Note como o código anterior, utilizando Hibernate, é extremamente mais simples que o escrito previamente com JDBC. Em poucas linhas de código atingimos o mesmo resultado. Claro que algumas coisas ainda parecem meio mágicas e outras ainda podem ser melhoradas nesse código, como o tratamento das transações, um melhor lugar para o fechamento da `Session`, que é uma espécie de conexão do Hibernate, e assim por diante. Mais para a frente aprenderemos essas melhores práticas, técnicas e detalhes do uso do framework.

Ao executar o método que realiza a busca de automóveis do banco de dados, o seguinte código SQL é disparado:

```
select automovel0_.id as id0_, automovel0_.anoFabricacao as
    anoFabri2_0_, automovel0_.anoModelo as anoModelo0_,
    automovel0_.marca as marca0_, automovel0_.modelo as modelo0_,
    automovel0_.observacoes as observac6_0_
from Automovel automovel0_
```

Apesar de parecer um pouco estranho, esse comando é apenas um `select` que discrimina as colunas que deverão ser buscadas no banco, uma a uma.

1.5 MUITOS FRAMEWORKS ORM, COMO EVITAR O VENDOR LOCK-IN?

A partir do momento em que os desenvolvedores perceberam que os frameworks ORM poderiam aumentar a produtividade e facilitar o desenvolvimento de aplicações, muitas bibliotecas como o Hibernate surgiram, enquanto outras que já existiam serviram de inspiração para o próprio Hibernate.

Dessa forma, chegou-se a um cenário em que existiam muitas bibliotecas para resolver o mesmo problema, mas cada uma fazendo o trabalho à sua maneira. Portabilidade passou a ser um problema, já que se você usasse qualquer uma biblioteca, ela não seria compatível com outras que já existiam no mercado. Isso criava o famoso *vendor lock-in*, ou seja, uma vez usando determinada distribuição, ficava-se preso à mesma.

Justamente com o intuito de resolver essa situação, em 2006, foi criada uma especificação para as bibliotecas ORM, no caso, a *Java Persistence API - JPA*. Essa nova especificação não representou grandes alterações na forma de usar um ORM, para quem já estava acostumado como Hibernate. Entre algumas poucas diferenças, destacam-se os nomes dos métodos e das classes que devem ser usadas. Adaptando o DAO anteriormente criado para o Hibernate, para a JPA, teremos o seguinte código:

```
public class JPAAutomovelDAO implements AutomovelDAO {

    public void salva(Automovel a) {
        EntityManager em = abreConexao();
        em.getTransaction().begin();

        em.persist(a);

        em.getTransaction().commit();
        em.close();
    }

    public List<Automovel> lista() {
        EntityManager em = abreConexao();

        List<Automovel> automoveis =
            em.createQuery("select a from Automovel a").getResultList();

        em.close();
    }
}
```

```
        return automoveis;
    }
}
```

Repare no código que houve apenas algumas mudanças de nomes com relação ao uso do Hibernate. Por exemplo, `Session` virou `EntityManager`, o método `save`, na JPA é `persist` e outros pequenos detalhes.

Nos próximos capítulos vamos aprender a usar a JPA, através do Hibernate como implementação, além de ver quais as melhores práticas para integrar ambas as tecnologias com uma aplicação web.

1.6 O DESENVOLVIMENTO WEB COM SERVLETS E O PADRÃO MVC

Com a popularização das aplicações web, muitas plataformas ganharam popularidade, como foi o caso do PHP e do ASP, e não demorou para que o Java também suportasse de alguma maneira o desenvolvimento web.

A infraestrutura para desenvolvimento Web em Java é baseada em Servlets. Hoje chamamos de infraestrutura, mas, nos primórdios, elas eram usadas diretamente para desenvolver aplicações.

Uma Servlet é uma classe Java que fica no servidor e é invocada com base em algum padrão de URL. Por exemplo, vamos considerar que temos uma aplicação de cadastro de veículos executando localmente no endereço <http://localhost:8080/automoveis>. Nessa aplicação, entre várias Servlets, vamos destacar uma:

- `/visualizar?id=<id do automóvel>` - mostra para o usuário os dados de um determinado automóvel;

O código dessa Servlet pode ser similar ao:

```
@WebServlet(urlPatterns="/visualizar")
public class VisualizaAutomovelServlet extends HttpServlet {

    protected void doGet(HttpServletRequestRequest req,
                        HttpServletResponse res)
                        throws IOException, ServletException {
        AutomovelDAO dao = new AutomovelDAO();
```

```
Automovel auto = dao.busca(request.getParameter("id"));

PrintWriter out = res.getWriter();
out.println("<html>");
out.println("<body>");

out.println("Modelo: " + auto.getModelo() + "<br />");
out.println("Ano do Modelo: " + auto.getAnoModelo()
            + "<br />");

out.println("</html>");
out.println("</body>");
}
}
```

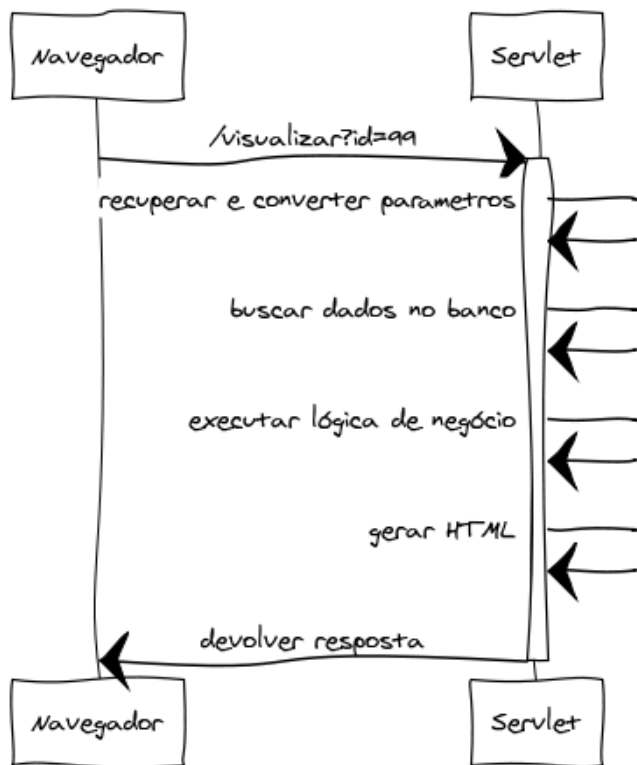



Figura 1.3: Fluxo de execução onde tudo fica na Servlet

Note como escrevemos o código HTML dentro da nossa classe Java. Bem estranho, não? E o pior é a complexidade na manutenção disso. Imagine que precisássemos devolver uma resposta mais elaborada para o cliente, com CSS e Javascript misturados nessa resposta. Ficaria quase impossível manter esse código.

Evidentemente que essa forma de desenvolver aplicações apresenta vários problemas e talvez o mais claro seja a presença de HTML dentro de um código Java. Uma alternativa para solucionar esse problema foi justamente os `JSPs`.

1.7 TIRANDO HTML DO CÓDIGO JAVA COM AS JAVASERVER PAGES

As JSPs, são páginas HTML que podem ter dentro de si trechos de código Java. Como em um projeto real, a quantidade de HTML dentro de uma Servlet seria muito maior do que o visto nos exemplos anteriores, e na prática o código Java de verdade (e não o que escrevia HTML) era exceção. Com as JSPs em vez de ter algo parecido com o servlet que respondia por */visualizar*, poderíamos ter uma página como visto a seguir.

```
<html>
  <body>
    Modelo: <%= req.getAttribute("modelo") %>
    Ano do Modelo: <%= req.getAttribute("anoModelo") %>
  </body>
</html>
```

Veja como nosso código agora pareceu melhor. Agora a proporção parece estar mais correta, sendo o HTML a regra e o Java a exceção. Nas JSPs temos a possibilidade da escrita direta do HTML, que além de deixar o código mais legível, deixa-o mais manutenível.

Apesar da notória melhoria, na mistura entre HTML e Java permaneceram, através dos Scriptlets, aqueles estranhos sinais de `<% %>`. Isso provocou o surgimento de técnicas para eliminar esse código Java da visualização do resultado. Foi justamente o papel da JSTL (*Java Standard Tag Library*) e da EL (*Expression Language*).

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <body>
    Modelo: <c:out value="${modelo}" />
    Ano do Modelo: <c:out value="${anoModelo}" />
  </body>
</html>
```

Mas como os dados chegam na tela? De onde estão vindo o `modelo` e o `anoModelo` para serem usados pela Tag `c:out`?

1.8 ORGANIZE SEU CÓDIGO COM A SEPARAÇÃO DE RESPONSABILIDADES

Precisamos agora fazer com que a Servlet `VisualizaAutomovelServlet` disponibilize as informações para o JSP que será exibido em seguida. É uma simples alteração, bastando adicionar as informações no `request` e despachar os mesmos para o JSP que será exibido:

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException {
    AutomovelDAO dao = new AutomovelDAO();

    Automovel auto = dao.busca(request.getParameter("id"));

    // coloca as informações no request para seguirem adiante
    req.setAttribute("modelo", auto.getModelo());
    req.setAttribute("anoModelo", auto.getAnoModelo());

    RequestDispatcher rd =
        req.getRequestDispatcher("/visualiza.jsp");

    // encaminha para o JSP, levando o request com os dados
    rd.forward(req, res);
}
```

Note que agora temos responsabilidades melhores definidas e separadas nesse código, sendo que a da classe é realizar a lógica necessária para buscar o automóvel no banco de dado, e o JSP é responsável por exibir os dados que foram passados pela Servlet.

Esse foi um primeiro passo para atingirmos um bom nível de organização em nosso código. Porém, perceba que não está perfeito ainda, já que estamos tendo todo o trabalho de pegar as informações do `request` e repassá-las ao JSP. Como será que podemos melhorar isso?

1.9 O MVC E OS FRAMEWORKS

No problema anterior, acabamos por deixar nossa Servlet ainda com bastante responsabilidade, visto que ela precisa recolher os dados do `request`, se necessário convertê-los, além de disponibilizar os objetos para o JSP e despachar a requisição para a página que será exibida para o usuário.

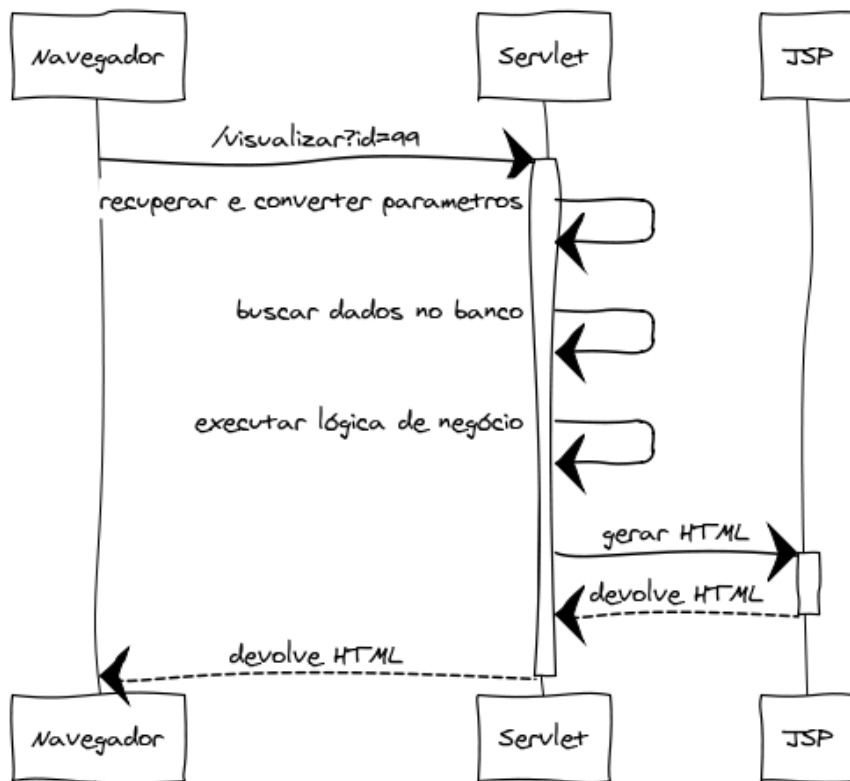


Figura 1.4: Fluxo de execução onde apenas a geração do HTML sai da Servlet

Seria muito melhor que a regra de negócio, no caso a busca no banco de dados, ficasse isolada da infraestrutura. Poderíamos fazer com que toda requisição recebida pela aplicação caísse em um lugar único, cuja responsabilidade fosse fazer as tarefas chatas e repetitivas de infraestrutura que não queremos colocar no meio do código de nossa aplicação. Tarefas como pegar parâmetros do request e convertê-los, por exemplo, seriam feitas por esse componente.

Dessa maneira, teríamos o fluxo:

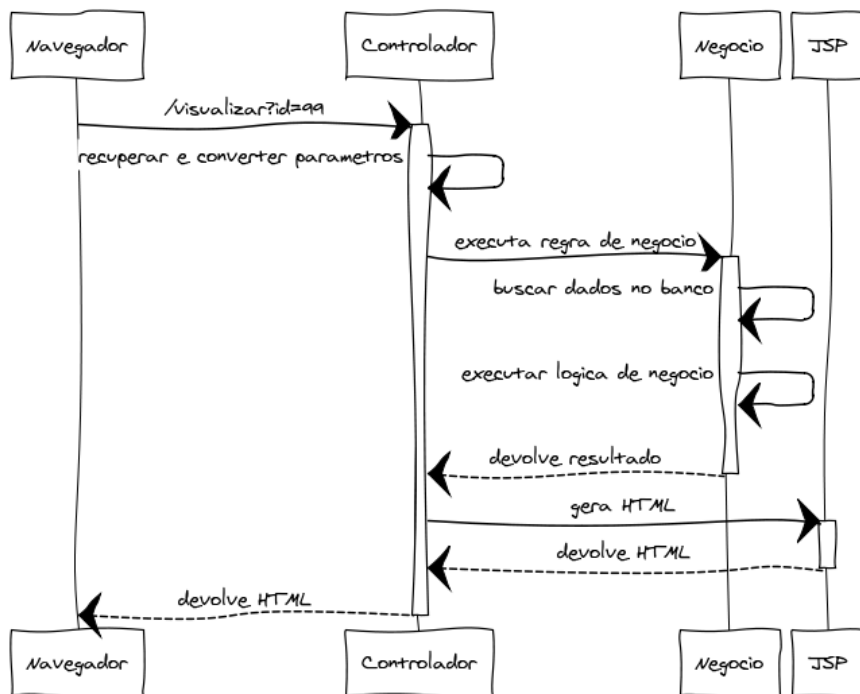


Figura 1.5: Fluxo de execução onde as tarefas ficam bem distribuídas

A partir dessa separação entre as diversas tarefas que nossa aplicação geralmente faz, chegamos ao modelo **MVC** (*Model-View-Controller*). Vejamos a responsabilidade de cada uma dessas partes.

Visão - View

A *Visão* (*View*) representa a parte que interage com o usuário, as telas, que podem ser formadas por arquivos JSP, HTML, imagens, JavaScript e CSS. Na *view* colocamos apenas coisas relacionadas com a apresentação ou visão dos dados, e não a lógica da nossa aplicação (que vai no *Model*). Isso não quer dizer que não podemos ter programação, como tomadas de decisão como um `if` ou iterar dados com um `while`. Significa que qualquer *script* da apresentação tem a responsabilidade de tratar coisas de apresentação: formatar uma data, esconder um campo, listar itens de uma lista.

Modelo - Model

O *Modelo (Model)* faz toda a parte inteligente do sistema: cálculos, validações de negócio, processamento, integrações etc. Com essas lógicas isoladas da infraestrutura da aplicação, podemos facilmente reaproveitá-la em outros lugares.

Por exemplo, podemos reaproveitar todo o módulo de cálculos de taxas do sistema de uma concessionária que é web em um módulo Desktop, porque o mesmo deveria se encontrar totalmente desacoplado de tecnologias usadas na camada de visão, como web ou Desktop.

Controlador - Controller

O *Controlador (Controller)* é responsável por receber as requisições do usuário, montar os objetos correspondentes e passá-los ao *Model* para que faça o que tem que ser feito. Depois de receber o retorno da operação executada, o controlador apresenta o resultado para o usuário, geralmente o redirecionando para a *View*.

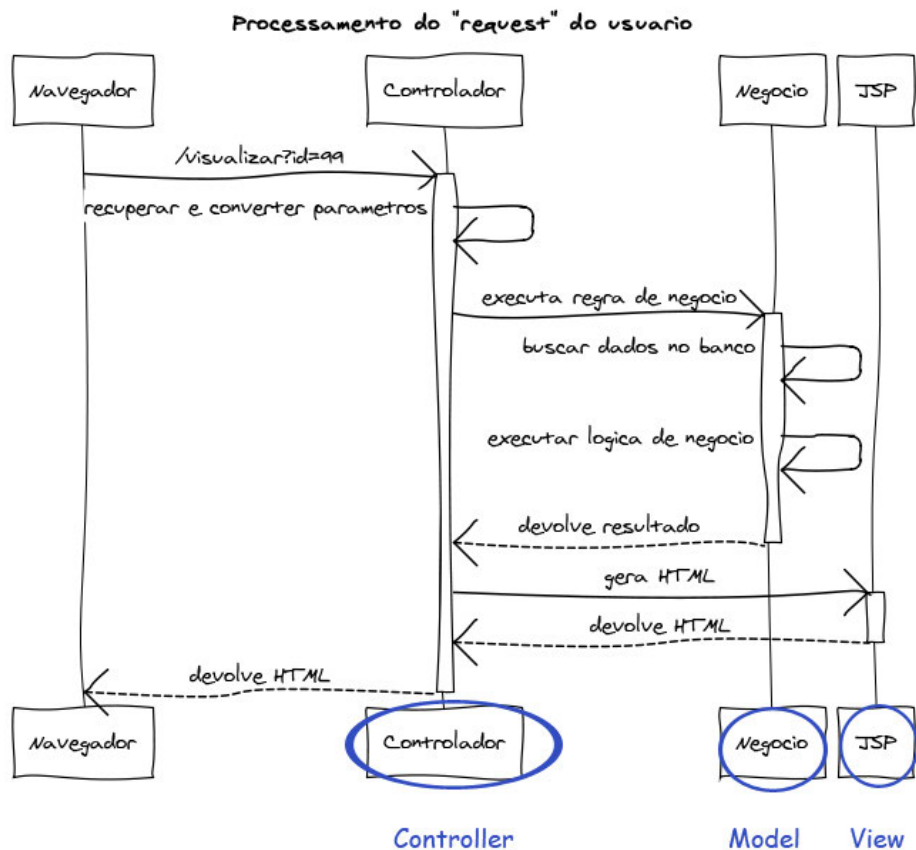


Figura 1.6: Fluxo de execução onde as tarefas ficam bem distribuídas

Mas como faço para separar todas essas responsabilidades em minha aplicação? Justamente para auxiliar nesse processo, a comunidade de desenvolvedores se organizou e começou a desenvolver diversas ferramentas que auxiliam nesse trabalho. Essas ferramentas são conhecidas como Frameworks MVC ou Controladores MVC.

Existem vários desses frameworks disponíveis para se usar, como o Struts, SpringMVC, JSF (Java Server Faces), VRaptor entre outros. Neste livro vamos nos ater ao JSF, explicando como ele pode facilitar o desenvolvimento de aplicações Web em Java.

1.10 POR QUE USAR JSF?

A exemplo do que aconteceu com os frameworks de mapeamento objeto-relacional, surgiram diversos frameworks MVC, porém estes ainda em maior número. Daí surgiu a necessidade de se padronizar um framework MVC, assim como a JPA veio padronizar o ORM.

Enquanto a JPA estabeleceu uma API muito familiar para quem era acostumado com o framework dominante, o Hibernate, o JSF veio muito diferente do que havia na época. Isso se deve também à grande segmentação entre os frameworks MVC. Enquanto muitos desenvolvedores utilizavam o Hibernate como framework ORM, o framework MVC mais utilizado na época do surgimento do JSF era o Struts, que possuía fama de ser complexo e pouco produtivo.

Por ter uma forma tão diferente do “status quo” da época, a aceitação do JSF não foi tão imediata quanto a da JPA. Enquanto a adoção desta foi quase imediata, o JSF enfrentou muita resistência, principalmente em sua versão 1.X. Neste livro abordaremos a versão 2.X, que acabou com muitas limitações da versão anterior, oferecendo uma ferramenta produtiva para o desenvolvimento de aplicações Web.

Outro fator que causou estranheza no primeiro contato com o JSF foi o fato de ele ser um framework baseado em componentes e não em ações, como eram os principais frameworks da época, e como ainda é até hoje. Veremos mais sobre como esses paradigmas se diferem em breve.

1.11 O PRIMEIRO CONTATO COM O JSF

Através do uso de um framework MVC, temos componentes específicos para realizar as tarefas da nossa aplicação. No caso do JSF 2, nossas telas serão escritas em um arquivo XHTML utilizando Tags do próprio JSF. Dessa forma, podemos fazer um simples formulário de cadastro de automóvel da seguinte forma:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>Cadastro de automóvel</title>
  </h:head>

  <h:body>
```



```
<h:form>
    Modelo: <h:inputText
        value="#{automovelBean.automovel.modelo}"/>
    <br/>
    <h:commandButton value="Grava"
        action="#{automovelBean.grava}" />
</h:form>
</h:body>
</html>
```

Repare no uso de Tags para criar os elementos como os inputs e os botões. Isso é necessário devido ao funcionamento interno do Framework, que vamos ver com detalhes no decorrer do livro. Abrindo essa página no navegador, você vê uma tela similar à figura 1.7

Figura 1.7: Formulário apenas com o campo do modelo do Automóvel

Note também que é indicado no código que, ao clicar no botão gravar, sua ação é `#{automovelBean.grava}`. Isso indica que será chamado o método `grava` de uma classe chamada `AutomovelBean`. Essa classe é o que chamamos de *Managed Bean* e é responsável por integrar o que temos na tela com a regra de negócio que será executada:

```
@ManagedBean
public class AutomovelBean {
    private Automovel automovel = new Automovel();

    public void grava() {
        new AutomovelDAO().grava(automovel);
    }

    // getter e setter
}
```

Eis outro ponto interessante: como na tela não criamos os inputs com Tags HTML e sim com o JSF, como eles são exibidos? O navegador entende as Tags do

JSF?

Na verdade, os componentes do JSF apenas geram o código HTML necessário. Se pedirmos para visualizar o código fonte da página, teremos algo similar a:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Cadastro de Automóvel</title>
  </head>
  <body>
    <form id="j_idt6" name="j_idt6" method="post"
      action="/faces-motors/faces/automovel/cadastro.xhtml"
      enctype="application/x-www-form-urlencoded">

      <input type="hidden" name="j_idt6" value="j_idt6" />

      Modelo: <input type="text" name="j_idt6:j_idt9" />

      <input type="submit" name="j_idt6:j_idt18" value="Grava" />

      <input type="hidden" name="javax.faces.ViewState"
        id="javax.faces.ViewState"
        value="2892395016760917637:4875835637171130451"
        autocomplete="off" />

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

Não se assuste com esse estranho código gerado. Nos próximos capítulos vamos aprender mais sobre como ele é gerado, e como podemos ter um controle maior sobre ele, bem como o significado de tudo isso.

1.12 NOS PRÓXIMOS CAPÍTULOS

Vimos rapidamente as tecnologias que vamos esmiuçar durante o livro, por enquanto sem muitos detalhes, apenas para que você sinta um pouco do que está pela frente. Nos próximos capítulos, vamos criar uma aplicação do zero, com diversas funcionalidades que são necessárias em aplicações comerciais, mostrando e explicando sempre novos recursos da JPA, do JSF e como integrá-los entre si.

Recomendamos também que o leitor que queira acompanhar o código em seu computador, testar os exemplos e customizá-los o faça. O livro está escrito de maneira a possibilitar essa dinâmica.

Por outro lado, também consideramos que o leitor queira usá-lo como material de referência em assuntos específicos, como o ciclo de vida do JSF. Para isso, basta encontrar a página adequada no sumário e estudar o assunto que lhe interessa.

CAPÍTULO 2

Primeiros passos com a JPA

Neste capítulo veremos os conceitos mais básicos de JPA e JSF. O objetivo é conseguirmos desenvolver uma primeira versão da nossa aplicação, já com uma interface gráfica e a persistência de dados funcionando plenamente.

2.1 DEFINIÇÃO DO MODELO

Como ponto de partida vamos analisar nossa classe `Automovel` cujo código pode ser visto a seguir:

```
@Entity
public class Automovel {

    @Id @GeneratedValue
    private Long id;
    private String marca;
    private String modelo;
    private Integer anoFabricacao;
```

```
private Integer anoModelo;  
private String observacoes;  
  
// getters e setters se necessário  
}
```

A única coisa que a difere de uma classe comum é a presença das anotações: `@Entity`, `@Id` e `@GeneratedValue`.

No contexto da JPA, chamamos de entidades as classes que estão vinculadas com uma unidade de persistência. Como a JPA trata de mapeamento objeto-relacional, a unidade de persistência é um banco de dados, e de modo geral, uma entidade estará associada a uma tabela nesse banco.

Por conta da relação entre a entidade e a tabela no banco de dados, quando quisermos persistir um objeto `Automovel`, será necessário fazer um `insert` em uma tabela. Porém, qual é o nome dessa tabela e quais colunas terão que ser utilizadas na inserção?

No nosso exemplo, não definimos nenhum nome de coluna e tabela. Dessa forma, estamos seguindo uma convenção da própria JPA. Nessa situação, a tabela terá exatamente o mesmo nome da entidade e para cada atributo da classe, teremos uma coluna com o mesmo nome. Ao persistirmos um objeto do tipo `Automovel` em um banco de dados relacional, a gravação é feita numa tabela com uma estrutura similar à figura 2.1:



Figura 2.1: Tabela Automovel

Percebemos que, para uma entidade simples, precisamos apenas dizer que ela é uma entidade, por meio da anotação `@Entity`, e especificar o atributo que representa a sua chave primária com a anotação `@Id`. Pronto.

Se, no nosso exemplo, desejássemos que a chave primária fosse algo relacionado à aplicação, e não um identificador com valor gerado automaticamente - por exemplo, o CPF de uma pessoa -, não precisaríamos colocar nada além das anotações `@Entity` e `@Id`, como na classe a seguir:

```
@Entity
public class Cliente {
    @Id
    private String cpf;

    // outros atributos
}
```

2.2 CONFIGURAÇÃO BÁSICA DO PERSISTENCE.XML

O próximo passo é indicarmos para a JPA as informações de conexão com o banco de dados, como o usuário, senha e endereço de acesso. Para isso, temos que criar um arquivo chamado `persistence.xml` e colocá-lo na pasta `META-INF`. Esse diretório deve estar no classpath do projeto, portanto, junto das classes.

Esse arquivo possui um cabeçalho, declarado com a tag `persistence`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

</persistence>
```

Dentro de `persistence`, é preciso indicar para a JPA qual conjunto de configurações teremos, para que seja possível indicar qual banco de dados será utilizado. Esse conjunto de configurações é chamado de `persistence-unit`, ao qual devemos dar um nome:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ... >
  <persistence-unit name="default">

    </persistence-unit>
</persistence>
```

Chamamos de `default`, mas **poderia ser dado qualquer outro nome**. Guarde-o com atenção, pois futuramente ele deverá ser referenciado no código Java.

Agora, basta indicar os dados de conexão com o banco de dados, como o usuário, senha e string de conexão com o banco. Essas configurações são feitas através de propriedades da unidade de persistência, descrita pela tag `properties`.

Dentro de `properties` são colocadas tags `property` contendo um atributo `name` indicando qual é a configuração que será feita, além de um atributo `value` com o conteúdo da configuração. Com isso, para o `username`, `password` e `url` de conexão, o XML fica similar a:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ... >
  <persistence-unit name="default">
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost/automoveis" />

      <property name="javax.persistence.jdbc.user"
        value="root" />

      <property name="javax.persistence.jdbc.password"
        value="password" />

    </properties>
  </persistence-unit>
</persistence>
```

Também é necessário indicar qual driver JDBC será utilizado e, como consequência, seu `jar` deverá estar na aplicação. Essa declaração é feita a partir da propriedade `javax.persistence.jdbc.driver`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ... >
```



```
<persistence-unit name="default">
  <properties>
    <!-- outras propriedades -->

    <property name="javax.persistence.jdbc.driver"
      value="com.mysql.jdbc.Driver" />

  </properties>
</persistence-unit>
</persistence>
```

Note que todas as propriedades até o momento começam com `javax.persistence.jdbc`. Isso acontece porque elas fazem parte da especificação da JPA. No entanto, não precisamos ficar presos a elas. Podemos utilizar propriedades que sejam específicas da implementação que será utilizada, como por exemplo, o Hibernate.

2.3 ESCOLHA O DIALETO DO BANCO DE DADOS

Quando trabalhamos diretamente com bancos de dados relacionais, escrevemos os códigos em SQL e podemos utilizar comandos que sejam específicos de um determinado banco de dados. A própria implementação da JPA que usaremos se encarrega de esconder esses detalhes de cada banco.

Uma das funcionalidades do Hibernate é a abstração de banco de dados. Dessa forma, não é necessário modificar o código da aplicação quando trocamos um banco de dados por outro. Uma das bases para isso é justamente a existência de diversos **dialeto**s, que representam as diferenças dos SQLs para cada banco de dados.

O dialeto que será usado na aplicação pode ser indicado no `persistence.xml` por meio da propriedade `hibernate.dialect`, em que dizemos qual classe existente dentro do próprio Hibernate possui a implementação desse dialeto. No caso do MySQL, temos:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ... >
  <persistence-unit name="default">
    <properties>
      <!-- outras propriedades -->

      <property name="hibernate.dialect"
```

```
        value="org.hibernate.dialect.MySQLInnoDBDialect" />
    </properties>
</persistence-unit>
</persistence>
```

OUTROS DIALETOS COMUNS

Além do dialeto para o MySQL, o Hibernate possui dialetos para diversos outros bancos de dados:

- **Oracle 10g** - `org.hibernate.dialect.Oracle10gDialect`
- **Oracle (genérico)** - `org.hibernate.dialect.OracleDialect`
- **SQL Server 2008** - `org.hibernate.dialect.SQLServer2008Dialect`
- **Postgre SQL** - `org.hibernate.dialect.PostgreSQLDialect`

Uma lista completa dos dialetos pode ser encontrada no Javadoc do Hibernate, disponível em <http://docs.jboss.org/hibernate/orm/4.1/javadocs/> ou verificando as classes no pacote `org.hibernate.dialect`.

Além de permitir que seja indicado qual o dialeto a ser usado, é possível realizar mais configurações que são específicas do Hibernate e também bastante úteis.

2.4 AUTOMATIZAÇÃO DA CRIAÇÃO E EVOLUÇÃO DAS TABELAS

Sempre que se trabalha com banco de dados, é necessário ter todas as tabelas criadas e atualizadas. No entanto, se o projeto estiver no início, não teremos nenhuma tabela. Justamente para facilitar isso, o próprio Hibernate consegue realizar essa criação, bastando indicar qual estratégia deverá ser utilizada, através da propriedade `hibernate.hbm2ddl.auto`.

Um valor possível para essa propriedade é `create`, com o qual o Hibernate cuida para que as tabelas sejam excluídas e recriadas. Dessa forma, irá trabalhar sempre a partir de um banco de dados vazio.

Além da opção `create`, existem mais 3 possibilidades:

- `create-drop` - as tabelas são criadas pelo próprio Hibernate e excluídas somente no final da execução do programa caso solicitado pelo desenvolvedor;
- `update` - nada é excluído, apenas criado, ou seja, todos os dados são mantidos. Útil para aplicações que estão em produção, das quais nada pode ser apagado;
- `validate` - não cria e nem exclui nada, apenas verifica se as entidades estão de acordo com as tabelas e, caso não esteja, uma exceção é lançada.

Um possível uso dessa propriedade no `persistence.xml` é:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ... >
  <persistence-unit name="default">
    <properties>
      <!-- outras propriedades -->

      <property name="hibernate.hbm2ddl.auto"
        value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

2.5 GRAVAÇÃO DO PRIMEIRO DADO NO BANCO DE DADOS

Com a configuração feita, é possível gravar as primeiras informações e para isso será necessária uma conexão com o banco de dados que acabamos de configurar no `persistence.xml`. Ou seja, precisaremos de uma conexão com a unidade de persistência (`persistence-unit`).

Para verificarmos se nossa configuração estava correta, vamos criar uma classe cujo objetivo é persistir um `Automovel`. Vamos chamá-la de `PersistidorDeAutomovel`. Essa classe terá um método `main` que fará tal trabalho.

Dentro do `main`, é preciso conseguir uma conexão com o banco de dados definido na `persistence-unit`. O primeiro passo para conseguirmos isso é descobrir quem sabe criar as conexões. Esse é justamente um dos papéis da classe `EntityManagerFactory`.

Para conseguirmos uma instância dela, basta utilizar o método `createEntityManagerFactory` da classe `Persistence` da própria JPA,

indicando qual é a `persistence-unit` que definimos no `persistence.xml`, que no caso chamamos de `default`:

```
public class PersistidorDeAutomovel {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.
            createEntityManagerFactory("default");
    }
}
```

Uma vez com a instância de `EntityManagerFactory`, basta invocarmos o método `createEntityManager`. Ele devolve um objeto do tipo `EntityManager`, que representa, entre outras coisas, uma conexão com a unidade de persistência.

```
public class PersistidorDeAutomovel {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.
            createEntityManagerFactory("default");

        EntityManager em = emf.createEntityManager();
    }
}
```

Pronto, agora basta instanciar e popular um objeto do tipo `Automovel`, e passarmos esse objeto para a `EntityManager` realizar a persistência.

```
public class PersistidorDeAutomovel {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.
            createEntityManagerFactory("default");

        EntityManager em = emf.createEntityManager();

        Automovel auto = new Automovel();
        auto.setAnoFabricacao(2010);
        auto.setModelo("Ferrari");
        auto.setObservacoes("Nunca foi batido");

        em.persist(auto);

        em.close();
    }
}
```

```
        emf.close();
    }
}
```

Ao executar esse código nada será gravado. Isso se deve ao fato de que em momento nenhum delimitamos a transação e dissemos que o `persist` estava envolvido. Mas corrigir esse problema é bem simples. Basta pedir a transação para a `EntityManager` e invocar o método `begin`, `commit` e `rollback` conforme necessário:

```
public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.
        createEntityManagerFactory("default");

    EntityManager em = emf.createEntityManager();

    Automovel auto = new Automovel();
    auto.setAnoFabricacao(2010);
    auto.setModelo("Ferrari");
    auto.setObservacoes("Nunca foi batido");

    EntityTransaction tx = em.getTransaction();
    tx.begin();
    em.persist(auto);
    tx.commit();

    em.close();
    emf.close();
}
```

Executando esse código, o novo carro será persistido no banco de dados, porém será possível perceber uma certa demora em sua execução, dando a sensação de que um simples `insert` leva tempo demais para se executar.

Isso acontece devido à invocação do método `createEntityManagerFactory`. Esse método realiza diversas tarefas, entre elas, ler as anotações das entidades anotadas e criar o pool de conexões com o banco de dados - o que vamos estudar e aprender a configurar futuramente.

Com isso, precisamos que esse código execute apenas uma vez em toda a aplicação, para que o único momento em que ocorre essa lentidão seja o da criação da

`EntityManagerFactory`. Dessa forma, as persistências subsequentes serão feitas em tempo normal.

Podemos garantir que a `EntityManagerFactory` seja criada uma vez só fazendo uma classe que tenha um atributo estático para armazenar essa instância, e que ela seja uma constante, para que não seja criada outras vezes. Podemos chamá-la de `JPAUtil`:

```
public class JPAUtil {  
    private static final EntityManagerFactory emf =  
        Persistence.createEntityManagerFactory("default");  
}
```

Podemos também fazer essa classe ter um método `getEntityManager` que devolva uma `EntityManager`, assim conseguimos ter acesso a ela sempre que precisarmos fazer as persistências:

```
public class JPAUtil {  
    private static final EntityManagerFactory emf =  
        Persistence.createEntityManagerFactory("default");  
  
    public static EntityManager getEntityManager() {  
        return emf.createEntityManager();  
    }  
}
```

Com essa alteração, podemos modificar o método `main` na classe `PersistidorDeAutomovel`, para usar essa nova classe `JPAUtil`:

```
public static void main(String[] args) {  
    EntityManager em = JPAUtil.getEntityManager();  
  
    // código para persistir o automóvel  
}
```

Por fim, ao executarmos o método `main` do `PersistidorDeAutomovel`, nada é exibido, mas as informações vão sendo persistidas no banco de dados. Atente-se para o fato de a estratégia de criação das tabelas poder realizar o `drop` nas suas tabelas. Caso não deseje que isso aconteça, utilize `update` na propriedade `hibernate.hbm2ddl.auto` no seu `persistence.xml`.

Outro ponto interessante é que você pode fazer com que o Hibernate imprima no console quais comandos SQLs estão sendo realizados no banco de dados. Para isso, basta adicionar uma nova propriedade no `persistence.xml`, chamada `show_sql`, com o valor `true`. Opcionalmente, você também pode usar a propriedade `format_sql` para que essa exibição saia formatada:

```
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />
```

Com essas duas propriedades habilitadas, o Hibernate vai exibir a seguinte saída:

Hibernate:

```
insert
into
    Automovel
    (anoFabricacao, anoModelo, marca, modelo, observacoes)
values
    (?, ?, ?, ?, ?)
```

2.6 CONSULTAS SIMPLES NO BANCO DE DADOS COM A JPA

Com dados gravados no banco de dados, podemos realizar pesquisas. A primeira consulta que vamos realizar através da JPA será um simples `select * from Automovel`, ou seja, vamos buscar todos os automóveis gravados.

Ao trabalharmos com a JPA, temos a opção de realizar consultas SQL, porém muitos desenvolvedores o evitam, principalmente pelo fato de poder prender a aplicação a recursos específicos de um banco de dados. Para isso, foi criada uma linguagem de consultas, parecida com SQL, da qual **uma das vantagens** é a abstração do banco de dados que está sendo usado. Essa linguagem é a JPQL (*Java Persistence Query Language*).

A JPQL possui sintaxe e estrutura muito parecidas com o SQL. Tanto que, para realizarmos uma pesquisa de todos os automóveis, pouca coisa muda, repare:

```
select a from Automovel a
```

Essa consulta JPQL realiza a busca de todos os automóveis. Na cláusula `from` **indicamos qual é a entidade** que deverá ser buscada e damos um *alias*, um apelido, para essa entidade, que no caso foi `a`. E indicamos na cláusula `select` à qual os automóveis serão devolvidos.

Para executarmos essa consulta, precisaremos, da mesma maneira que para a gravação, conseguir a `EntityManager`. Com ela em mãos, poderemos invocar o método `createQuery`, que recebe a `String` com a consulta que desejamos disparar e também o tipo do objeto que a consulta deve devolver - que no nosso caso é `Automovel`. Isso devolverá um objeto do tipo `javax.persistence.Query`:

```
EntityManager em = JPAUtil.getEntityManager();
Query q = em.createQuery("select a from Automovel a", Automovel.class);
```

Por fim, podemos executar essa consulta invocando o método `getResultList` no objeto `Query`, que devolverá uma lista com objetos do tipo indicado no momento da chamada ao `createQuery`:

```
EntityManager em = JPAUtil.getEntityManager();
Query q = em.createQuery("select a from Automovel a", Automovel.class);

List<Automovel> autos = q.getResultList();
```

Pronto, agora que conseguimos a lista, podemos iterar sobre ela para mostrar, por exemplo, a marca de todos os automóveis:

```
EntityManager em = JPAUtil.getEntityManager();
Query q = em.createQuery("select a from Automovel a", Automovel.class);

List<Automovel> autos = q.getResultList();

for(Automovel a : autos) {
    System.out.println(a.getMarca());
}
```

CONSULTAS COMPLEXAS COM A JPQL

No capítulo 6, aprenderemos a fazer consultas complexas envolvendo agrupamentos, ordenação, joins e vários outros recursos que costumamos usar no dia dia ao escrever *queries* que são executadas no banco de dados.

2.7 EXCLUSÃO DE DADOS COM A JPA

Muitas vezes, acabamos por inserir dados por engano no banco de dados, ou eles podem não ser mais necessários e passam a ser passíveis de serem descartados. Por isso, precisamos poder excluir as informações, ou no nosso caso, precisamos excluir automóveis do banco de dados.

Essa tarefa com a JPA se torna extremamente simples, já que, da mesma maneira que no cadastro, ela faz todo o trabalho para nós. Basta que chamemos o método adequado, no caso `remove`, passando um objeto `Automovel` com o `id` populado:

```
EntityManager em = JPAUtil.getEntityManager();
Transaction tx = em.getTransaction();

Automovel auto = em.getReference(Automovel.class, 1);

tx.begin();
em.remove(auto);
tx.commit();
```

O USO DO MÉTODO `getReference`

Mais a frente veremos mais do método `getReference`. Por enquanto nos importa saber que os trechos de código a seguir são equivalente, com o diferencial que usando o método da interface `EntityManager` temos como retorno um objeto gerenciado, mesmo sem acessar o banco de dados. Mas o que isso significa ao certo veremos em detalhes no capítulo 4.

```
//objeto gerenciado apenas com o id populado (não vai no banco de dados)
Automovel auto = em.getReference(Automovel.class, 1);

//objeto não gerenciado com o id populado
Automovel auto = new Automovel();
auto.setId(1);
```

Ao executarmos o código acima, por exemplo, dentro de um método `main`, o `delete` é executado, excluindo o automóvel cujo `id` era 1. Note que é imprescindível

que o objeto a ser excluído tenha seu `id` populado, caso contrário a exclusão não é feita, já que não se tem como saber qual é a informação que deve ser removida do banco de dados.

2.8 O QUE MAIS VEM PELA FRENTE?

Em poucos minutos já conseguimos ver como configurar e realizar 3 operações comuns com o banco de dados: o `select`, o `insert` e o `delete`. Ainda falta o `update`, mas esse é um caso à parte sobre o qual vamos falar no capítulo 4.

Durante o livro aprenderemos como realizar queries complexas, validações de dados de uma forma extremamente simples, otimização de performance e boas práticas para organizar seu código envolvendo a JPA.

CAPÍTULO 3

Primeiros passos com o JSF

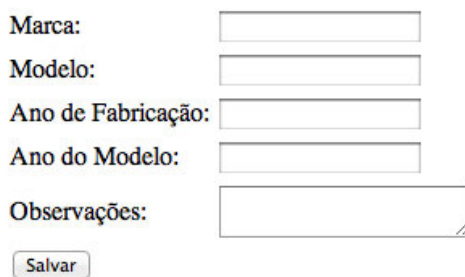
No capítulo 2, definimos a entidade `Automovel` com seus atributos e conseguimos persistir essa informação no banco de dados através da execução de uma classe contendo um método `main`.

A partir de agora, vamos levar essa mesma integração para a web e para isso vamos usar o JSF, que nos ajudará no trabalho de criar a aplicação web.

O nosso primeiro trabalho vai ser permitir o cadastro dos automóveis por meio de uma interface web, e não mais via aquele estranho método `main`.

3.1 ONDE E COMO DEFINO TELAS COM O JSF?

Precisamos construir a tela onde serão cadastrados novos automóveis. A tela que queremos como resultado final é parecida com a figura 3.1.



Formulário de cadastro de automóveis:

- Marca:
- Modelo:
- Ano de Fabricação:
- Ano do Modelo:
- Observações:
- Botão: Salvar

Figura 3.1: Formulário que vamos construir

Para criarmos essa tela, o primeiro passo é termos um arquivo XHTML, no qual escreveremos o código necessário para mostrar o formulário de cadastro de automóveis. Podemos chamar esse arquivo de `cadastraAutomoveis.xhtml`.

Quando desenvolvemos um projeto usando o JSF, a construção da tela se torna um pouco diferente de outros frameworks, como o Struts, VRaptor e SpringMVC, nos quais costumamos usar apenas elementos HTML para criar os formulários, tabelas e demais características da visualização das informações. No JSF, praticamente toda a tela é criada a partir de Taglibs próprias dele, que possuem o papel de renderizar o HTML adequado para o funcionamento do framework. Por esse motivo, no começo do nosso documento XHTML, precisamos importar essas Taglibs com o seguinte código:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

</html>
```

Note que na abertura da Tag `html` indicamos alguns *namespaces*. O primeiro é o *namespace* padrão para XHTMLs, enquanto o outro é específico para conseguirmos usar as Tags do JSF em nosso XHTML.

O `xmlns:h="http://java.sun.com/jsf/html"` indica que estamos habilitando através do prefixo `h` as Tags do JSF que renderizam HTML.

3.2 CRIAÇÃO DA TELA E DOS INPUTS BÁSICOS DE DADOS

Com as Tags habilitadas pelo *namespace* `h`, podemos agora construir o conteúdo da página, contendo o formulário e os campos para que os usuários possam co-

locar os dados do automóvel a ser cadastrado. Dentro da Tag `html`, podemos definir o corpo da página e um campo de texto para a marca do automóvel. Vamos fazer isso utilizando as Tags do JSF. Para isso, vamos escrever no arquivo `cadastraAutomoveis.xhtml`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

    <h:body>
        <h:form>
            Marca: <h:inputText />
        </h:form>
    </h:body>
</html>
```

Esse XHTML mínimo gera a tela da figura 3.2.



Figura 3.2: Formulário JSF com um único campo

Repare que usamos a Tag `h:body` em vez da `body` do HTML, a `h:form` no lugar de `form` e `h:inputText` em vez de `input type="text"`. Estamos usando os componentes do JSF para gerar o HTML para nós. Se acessarmos esse `cadastraAutomoveis.xhtml` no navegador e pedirmos para ver o código fonte gerado, ele será similar ao código a seguir:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <body>
        <form id="j_idt4" name="j_idt4" method="post"
              action="/faces-motors/automovel/editar.jsf"
              enctype="application/x-www-form-urlencoded">

            <input type="hidden" name="j_idt4" value="j_idt4" />

            Marca: <input type="text" name="j_idt4:j_idt25" />

            <input type="hidden" name="javax.faces.ViewState"
                  id="javax.faces.ViewState"
```

```

        value="-2173774569225484208:-2914758950899476002"
        autocomplete="off" />
    </form>
</body>
</html>

```

O código fonte é gerado pelo próprio JSF. Repare no `name` do `input`: é um código também gerado pelo JSF. Esses códigos que a princípio parecem malucos são primordiais para o bom funcionamento do JSF. Com o passar do tempo nos acostumaremos com eles e vamos entender por completo o motivo de eles serem tão importantes para o JSF.

Esse formulário pode ser expandido para abranger todos os campos que precisaremos para o automóvel, como o modelo, ano de fabricação, ano do modelo e observações.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

    <h:body>
        <h:form>
            Marca: <h:inputText /><br/>
            Modelo: <h:inputText /><br/>
            Ano de Fabricação: <h:inputText /><br/>
            Ano do Modelo: <h:inputText /><br/>
            Observações: <h:inputTextarea /><br/>
        </h:form>
    </h:body>
</html>

```

Ao abrirmos essa página no navegador, teremos uma tela parecida com a da figura 3.3.

A imagem mostra um formulário web com um layout desorganizado. As labels 'Marca:', 'Modelo:', 'Ano de Fabricação:', 'Ano do Modelo:' e 'Observações:' estão alinhadas à esquerda, mas os campos de entrada (inputs e textarea) não estão alinhados à direita delas, resultando em uma aparência desalinhada e 'torta'. Um botão 'Salvar' está posicionado abaixo dos campos.

Figura 3.3: Formulário com layout desorganizado

Note como o conteúdo ficou todo desalinhado. As descrições dos campos empurram os `inputs` para a direita e isso faz com que a tela pareça torta e estranha. Pensando em HTML, podemos contornar esse problema de pelo menos duas maneiras: a primeira, criando uma tabela para o formulário e dividindo a descrição e o campo em colunas diferentes, e a segunda seria fazendo esse trabalho de alinhamento com CSS.

Com o JSF, podemos fazer esse alinhamento de uma maneira bem simples. Basta envolver o conteúdo do formulário na Tag `h:panelGrid`, que divide o conteúdo que estiver dentro dela em colunas. Podemos indicar quantas colunas queremos pelo atributo `columns`.

```
<h:body>
  <h:form>
    <h:panelGrid columns="2">
      Marca: <h:inputText />
      Modelo: <h:inputText />
      Ano de Fabricação: <h:inputText />
      Ano do Modelo: <h:inputText />
      Observações: <h:inputTextarea />
    </h:panelGrid>
  </h:form>
</h:body>
```

O `h:panelGrid` quebra as linhas e colunas a partir dos elementos existentes dentro dele. Dessa forma, o texto `Marca:` fica em uma coluna e o `h:inputText` dele fica em outra coluna. Como já se completaram duas colunas, os próximos conteúdos virão na linha seguinte. E assim sucessivamente, até fazer a organização de todo o formulário.

3.3 COMMANDBUTTON, MANAGED BEANS E A SUBMISSÃO DE FORMULÁRIOS

Uma vez com o formulário pronto e organizado na tela, precisamos submeter os dados preenchidos pelo usuário para que, de alguma maneira, seja feita a gravação no banco de dados.

Para isso, o primeiro passo é termos na tela um botão que permita a submissão das informações, que pode ser adicionado por meio da Tag `h:commandButton`:

```
<h:body>
  <h:form>
    <h:panelGrid columns="2">
      Marca: <h:inputText />
      Modelo: <h:inputText />
      Ano de Fabricação: <h:inputText />
      Ano do Modelo: <h:inputText />
      Observações: <h:inputTextarea />

      <h:commandButton value="Salvar" />
    </h:panelGrid>
  </h:form>
</h:body>
```

Em `h:commandButton`, indicamos qual é o texto que aparece no botão através do atributo `value`, porém, não definimos qual ação deverá ser feita. Ou seja, precisamos indicar onde estará o código que vai fazer o cadastro no banco de dados.

Para realizarmos o cadastro, vamos precisar de uma classe que contenha algum método que faça a gravação. Esse método, por sua vez, deve ser invocado quando o botão for clicado. **Essa classe, que possui uma relação com a tela, é chamada *Managed Bean*** e não passa de um simples POJO - *Plain Old Java Object*. A única obrigatoriedade que temos ao criar um *Managed Bean* é o uso da anotação `@ManagedBean`:

```
@ManagedBean
public class AutomovelBean { }
```


POJO

O termo POJO - *Plain Old Java Object* foi cunhado por Martin Fowler, Rebecca Parsons e Josh MacKenzie para indicar classes simples Java, que não precisam implementar interface e nem realizar qualquer herança de um framework específico.

O trabalho que temos agora é de indicar que ao clicar no botão “Salvar” um método desse *Managed Bean* deve ser chamado. Para isso, vamos criar um método chamado `salva`:

```
@ManagedBean
public class AutomovelBean {

    public void salva() {
        // aqui vai o código para salvar
    }
}
```

Já estamos na metade do caminho para conseguir invocar o método ao clicar no botão, porém, ainda falta indicar para o botão que quando ele for clicado, o método `salva` deverá ser chamado. Para isso, a Tag `h:commandButton` possui um atributo chamado `action`, onde podemos indicar o que deve ser chamado quando clicado.

Para passarmos essa informação para o `action`, precisamos usar um recurso chamado *Expression Language*, que nos permite que no código da tela, no caso o `xhtml`, façamos uma referência para algo de um *Managed Bean*. Nesse caso, queremos fazer uma referência ao método `salva` do *Managed Bean* `AutomovelBean`, processo também conhecido como *binding*, então usamos:

```
<h:commandButton value="Salvar" action="#{automovelBean.salva}"/>
```

Nesse código, note que a *Expression Language* possui uma sintaxe um pouco diferente, um `#{ }`. O conteúdo das chaves é o que queremos referenciar com relação ao *Managed Bean*, nesse caso, o método `salva`. Repare também que a convenção para o nome do *Managed Bean* é *Camel Case*, porém com a primeira letra em minúsculo. Para usar outro nome, é possível usar o atributo `name` na anotação `@ManagedBean`:

```
@ManagedBean(name="OutroNomeParaOAutomovelBean")
public class AutomovelBean { }
```

E com isso, usáramos as *Expression Languages* `OutroNomeParaOAutomovelBean` em vez de `automovelBean`.

O QUE ACONTECE QUANDO CLICO NO BOTÃO?

É natural imaginar que quando o botão é clicado o método do *Managed Bean* é diretamente invocado. Essa sensação acontece devido ao fato de que o método efetivamente o executa. Porém, estamos trabalhando com uma aplicação web e no fundo, o que está acontecendo é uma requisição para a aplicação.

Ao receber essa requisição, o JSF “percebe” que você clicou no botão e que o método indicado no `action` deve ser invocado e portanto, ele faz essa chamada para você.

3.4 PASSE DADOS DA TELA PARA O MANAGED BEAN

A partir do momento em que é possível disparar um método do *Managed Bean* com o clique de um botão, queremos que sua ação seja fazer a gravação de um automóvel no banco de dados. Esse automóvel terá as informações que o usuário preencher na tela. Então o primeiro passo é fazer com que os dados cheguem de alguma maneira ao *Managed Bean*.

Agora que sabemos que para relacionar a tela com o *Managed Bean* basta ligá-los através da *Expression Language*, fica fácil inferir que temos que fazer o mesmo para os *inputs*.

Vamos adicionar o atributo `value` aos *inputs*, indicando qual propriedade do *Managed Bean* irá receber cada um dos valores. Com essa alteração, o formulário ficará com o seguinte código:

Marca:

```
<h:inputText value="#{automovelBean.marca}"/>
```

Modelo:

```
<h:inputText value="#{automovelBean.modelo}"/>
```

Ano de Fabricação:

```
<h:inputText value="#{automovelBean.anoFabricacao}"/>
```

Ano do Modelo:

```
<h:inputText value="#{automovelBean.anoModelo}"/>
```

Observações:

```
<h:inputTextarea value="#{automovelBean.observacoes}"/>
```

```
<h:commandButton value="Salvar" action="#{automovelBean.salva}"/>
```

Com isso, é necessário apenas termos os atributos no *ManagedBean* com seus respectivos *getters* e *setters*:

```
@ManagedBean
public class AutomovelBean {
    private String marca;
    private String modelo;
    private Integer anoFabricacao;
    private Integer anoModelo;
    private String observacoes;

    public String getMarca() {
        return this.marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    // outros getters e setters

    // método salva
}
```

Apenas isso já é o suficiente para conseguirmos receber os dados no nosso *AutomovelBean*, porém, ainda há algo que pode ser melhorado nesse código. No capítulo 1, criamos a classe *Automovel*, que já possuía todos esses atributos. Com isso, podemos deixar de ter essas várias informações na classe *AutomovelBean* e deixar apenas um atributo, no caso, o atributo *Automovel*:

```
@ManagedBean
public class AutomovelBean {
    private Automovel automovel = new Automovel();
}
```

```
public String getAutomovel() {  
    return this.automovel;  
}  
  
public void setAutomovel(Automovel automovel) {  
    this.automovel = automovel;  
}  
  
// método salva  
}
```

Para finalizar, basta ajustarmos os campos do formulário no `xhtml` para usar o novo atributo `automovel` em vez dos diversos atributos que tínhamos antes. Assim, cada campo do formulário deverá ser relacionado com seu respectivo atributo dentro de `automovel`:

Marca:

```
<h:inputText value="#{automovelBean.automovel.marca}"/>
```

Modelo:

```
<h:inputText value="#{automovelBean.automovel.modelo}"/>
```

Ano de Fabricação:

```
<h:inputText value="#{automovelBean.automovel.anoFabricacao}"/>
```

Ano do Modelo:

```
<h:inputText value="#{automovelBean.automovel.anoModelo}"/>
```

Observações:

```
<h:inputTextarea value="#{automovelBean.automovel.observacoes}"/>
```

```
<h:commandButton value="Salvar" action="#{automovelBean.salva}"/>
```

O que acontece quando o formulário é submetido é que o JSF avalia as *Expression Languages* e “percebe” que primeiro precisa conseguir o *ManagedBean* `automovelBean`.

Uma vez que conseguiu uma instância do *Managed Bean*, ele continua avaliando aquela expressão e percebe que precisa conseguir um `automovel` a partir daquele `automovelBean`. É aí que entra em ação o método `getAutomovel` no *Managed Bean*, pois o JSF vai chamá-lo.

Agora que chegou ao último nível da expressão e ele já conseguiu o objeto `automovel`, o JSF irá chamar o `set` adequado para cada propriedade. Então, para a marca, o JSF chamará o `setMarca` enviando como parâmetro a informação digitada no formulário. E assim fará para os outros campos.

CUIDADO COM A `NullPointerException`

É muito comum esquecermos de instanciar o objeto logo no atributo ou no construtor do *Managed Bean*. Nesse caso, quando o formulário fosse submetido, o JSF recuperaria a referência para aquele objeto, através do *getter*, porém, ele estaria nulo. Em seguida, tentaria chamar o *setter* adequado para aquela propriedade. Pronto, temos uma `NullPointerException` em nossa aplicação.

Para evitar cair nisso, lembre-se sempre de ter o objeto previamente instanciado.

Podemos inclusive, no método `salva`, pedir para mostrar algumas informações do automóvel. Assim, quando se clicar em salvar, veremos essa informação na saída do servidor:

```
@ManagedBean
public class AutomovelBean {
    private Automovel automovel = new Automovel();
    // getter e setter

    public void salva() {
        System.out.println("Marca: " + automovel.getMarca());
    }
}
```

Ao digitarmos “Ferrari” no campo Marca e submetermos o formulário, o log do servidor deverá exibir a mensagem:

Marca: Ferrari

Note também que ao submeter o formulário os campos voltam preenchidos com as informações que foram submetidas. Isso acontece porque, ao começar a gerar a resposta, que no caso será a mesma página, a renderização passa pelos *inputs* definidos no formulário. Como eles estão ligados com as propriedades do *Managed Bean*

que foram populadas na submissão do formulário pelo atributo `value`, na hora de renderizar os campos, o JSF vai buscar o valor, também definido no mesmo atributo `value` dos *inputs*.

Nesse momento, o JSF novamente avalia a *Expression Language*, primeiro recuperando o `automovelBean`, em seguida pegando o `automovel` através do `getAutomovel` no `automovelBean`, e depois ele recupera as informações desse automóvel, chamando o `getter` para cada *input* ligado ao *Managed Bean*.

Marca:	Ferrari
Modelo:	458 Spider
Ano de Fabricação:	2012
Ano do Modelo:	2013
Observações:	Carrinho do dia a dia

Salvar

Figura 3.4: Formulário com valores preenchidos

3.5 COMO RECEBER OS PARÂMETROS DIRETO NO MÉTODO

No *Managed Bean* `AutomovelBean`, quando precisamos receber os parâmetros da requisição, criamos um atributo e definimos seu *getter* e *setter*. **A partir da *Expression Language* 2.2**, utilizado pelo JBoss 7 e o Tomcat 7, por exemplo, não precisamos mais do *setter*. Com isso, é possível fazer com que o método definido na *action* do botão receba o parâmetro.

Dessa maneira, teremos no método `salva` um novo parâmetro do tipo `Automovel`, e no `xhtml` vamos referenciá-lo através do `automovel` do *Managed Bean*: `#{automovelBean.salva(automovelBean.automovel)}`. Dessa forma, teremos no *Managed Bean*:

```
@ManagedBean
public class AutomovelBean {
    private Automovel automovel = new Automovel();
    // getter

    public void salva(Automovel automovel) {
```

```

        // agora o método passou a receber automovel
    }
}

```

E o `xhtml` indicando o parâmetro para o método do *Managed Bean*:

```

<h:commandButton value="Salvar"
    action="#{automovelBean.salva(automovelBean.automovel)}/>

```

3.6 GRAVAÇÃO DO AUTOMÓVEL NO BANCO DE DADOS

Nesse momento, temos o *Managed Bean* com um método `salva` preparado para receber parâmetros, que, no nosso caso, será um objeto do tipo `Automovel`. No capítulo 2, configuramos o Hibernate e a JPA e fizemos o cadastro de um primeiro automóvel no banco de dados. Vamos usar a classe `JPAUtil` criada naquele capítulo para nos fornecer uma `EntityManager` e com isso, podemos fazer a persistência dos dados informados.

O primeiro passo é, no método `salva`, conseguirmos uma `EntityManager` por meio da `JPAUtil`:

```

@ManagedBean
public class AutomovelBean {
    private Automovel automovel = new Automovel();
    // getter e setter

    public void salva() {
        // conseguimos a EntityManager
        EntityManager em = JPAUtil.getEntityManager();
    }
}

```

Uma vez com a `EntityManager` em mãos, basta persistir o objeto `automovel`, envolvendo a persistência em uma transação:

```

@ManagedBean
public class AutomovelBean {
    private Automovel automovel = new Automovel();
    // getter e setter

    public void salva() {

```

```
// conseguimos a EntityManager
EntityManager em = JPAUtil.getEntityManager();
em.getTransaction().begin();

em.persist(automovel);

em.getTransaction().commit();
}
}
```

Um último passo, extremamente importante, é fecharmos a `EntityManager` que foi aberta, dessa forma, não mantemos recursos em uso desnecessariamente:

```
public void salva() {
    // conseguimos a EntityManager
    EntityManager em = JPAUtil.getEntityManager();
    em.getTransaction().begin();

    em.persist(automovel);

    em.getTransaction().commit();
    em.close();
}
```

Pronto, com isso conseguimos persistir o primeiro automóvel a partir da nossa aplicação JSF. É importante notar que o código escrito pouco difere do desenvolvido no capítulo 2. No decorrer do livro, aprenderemos melhores formas de se trabalhar com esse código de persistência e também como fazer um controle de transações mais transparente e simplificado.

3.7 LISTE OS DADOS COM O DATATABLE

Uma vez que temos os dados gravados no banco de dados, precisamos mostrá-los para os usuários. Portanto, vamos exibi-las em uma tabela, que listará todos os automóveis que estão gravados no banco de dados. Ao final do desenvolvimento, teremos uma tela parecida com a figura 3.5.

ID	Marca	Modelo	Ano Fabricação	Ano Modelo	Preço	Observações	Ações
1	Volkswagen	Fusca	1966	1966	R\$ 14.200,00	Fusquinha original	excluir
2	Fiat	Palio	2000	2001	R\$ 13.000,00	Muito bom	excluir
4	Fiat	Palio	2012	2013	R\$ 33.250,00	Modelo novo	excluir
5	Chevrolet	Camaro	2013	2013	R\$ 160.000,00	Completo	excluir

Figura 3.5: Listagem de Automóveis

O primeiro passo para a construção dessa tabela é criarmos um novo `xhtml`, que podemos chamar de `listaAutomoveis.xhtml`. Esse arquivo possui o mesmo cabeçalho importando a Tag do JSF que o formulário de cadastro:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

    <h:body>

    </h:body>
</html>
```

Agora podemos construir a tabela com o JSF através de um novo componente, o `dataTable`. Como o próprio nome diz, o objetivo desse componente é exibir dados tabulares, também conhecidos como *grids*. O `dataTable` precisa que seja fornecido um objeto para ele, através do atributo `value`, que no nosso caso será uma lista com os elementos que devem ser usados na listagem.

A lista com os elementos deve ser recuperada de um *Managed Bean* e, como continuamos trabalhando com automóveis, utilizaremos o mesmo `AutomovelBean` de antes. Dessa forma, precisamos ligar o `value` do `dataTable` com uma lista de automóveis no `AutomovelBean`. O primeiro passo, é criarmos a lista no *Managed Bean* e seu respectivo `getter`, que fará a consulta no banco de dados:

```
@ManagedBean
public class AutomovelBean {
    private Automovel automovel = new Automovel();

    private List<Automovel> automoveis;

    public List<Automovel> getAutomoveis() {
```

```

EntityManager em = JPAUtil.getEntityManager();
Query q = em.createQuery("select a from Automovel a",
                          Automovel.class);

this.automoveis = q.getResultList();
em.close();
return automoveis;
}

// método salva(), getter e setter do automovel
}

```

CÓDIGOS DE PERSISTÊNCIA DENTRO DO MANAGED BEAN

Muitos desenvolvedores são contra colocar códigos que envolvam JDBC, JPA e *queries* do banco de dados dentro de classes que auxiliam no trabalho com o framework, como o *Managed Bean*. Justamente para esses casos, diversos desenvolvedores tendem a usar classes para encapsular essas tarefas, fazendo o papel de um DAO (*Data Access Object*) ou um repositório.

Na seção 4.2 vamos ver como fazer essa organização.

Agora que temos o método `getAutomoveis` que devolve a lista recém buscada do banco de dados, podemos apontar o `dataTable` para ele:

```
<h:dataTable value="#{automovelBean.automoveis}" />
```

Para montarmos essa tabela, vamos precisar dizer quais colunas queremos e quais informações deverão ser mostradas nelas. Então, vamos definir uma variável que vai identificar cada um dos automóveis buscados para serem exibidos. Vamos chamar essa variável de `automovel` pelo atributo `var` do `dataTable`:

```
<h:dataTable value="#{automovelBean.automoveis}" var="automovel"/>
```

Agora, dentro dessa tabela, podemos criar a primeira coluna, através da Tag `h:column` e colocar o conteúdo que deverá ser mostrado dentro dela:

```
<h:dataTable value="#{automovelBean.automoveis}" var="automovel">
```

```

    <h:column>
        #{automovel.marca}
    </h:column>

</h:dataTable>

Pronto, já temos uma tabela que, quando a tela é acessada, mostra uma lista com
a marca de cada um dos automóveis cadastrados. Para criar novas colunas, bastaria
repetir blocos da Tag h:column:

<h:dataTable value="#{automovelBean.automoveis}" var="automovel">

    <h:column>
        #{automovel.marca}
    </h:column>

    <h:column>
        #{automovel.modelo}
    </h:column>

    <!-- outros colunas -->
</h:dataTable>

```

Nesse instante, temos uma tabela como a mostrada pela figura 3.6:

1	Volkswagem	Fusca	1966	1966	R\$ 14.200,00	Fusquinha original
2	Fiat	Palio	2000	2001	R\$ 13.000,00	Muito bom
4	Fiat	Palio	2012	2013	R\$ 33.250,00	Modelo novo
5	Chevrolet	Camaro	2013	2013	R\$ 160.000,00	Completo

Figura 3.6: Listagem de Automóveis sem borda e cabeçalho

Note que ficou confuso distinguir o que cada uma das colunas representa. Não temos uma identificação do que a cada uma delas se refere. Para resolver esse problema, podemos adicionar uma borda para separar cada célula e um cabeçalho a cada coluna para identificar seu conteúdo.

Adicionar a borda é mais simples, então faremos isso primeiro. Como nossa preocupação, por enquanto, não é *layout*, basta adicionar `border="1"` na nossa `h:dataTable`.

```
<h:dataTable value="#{automovelBean.automoveis}" var="automovel"
              border="1">

    <!-- colunas aqui dentro -->
</h:dataTable>
```

Com essa alteração o resultado será como na figura 3.7.

1	Volkswagem	Fusca	1966	1966	R\$ 14.200,00	Fusquinha original
2	Fiat	Palio	2000	2001	R\$ 13.000,00	Muito bom
4	Fiat	Palio	2012	2013	R\$ 33.250,00	Modelo novo
5	Chevrolet	Camaro	2013	2013	R\$ 160.000,00	Completo

Figura 3.7: Listagem de Automóveis com borda e sem cabeçalho

Agora sim vamos adicionar o cabeçalho de cada coluna. Para isso, precisamos modificar uma característica do `h:column`, que é o seu cabeçalho. Essas características que os componentes possuem, que não necessariamente são visuais, são o que chamamos de facetas, as quais podemos modificar através de um novo componente chamado `facet`.

O objetivo do `facet` é apenas mudar a característica de um outro componente, no caso o `h:column`, e não necessariamente gerar código HTML. Justamente por isso, essa Tag não fica sob o *namespace* `h`. Precisamos importá-la de outro lugar. A partir desse momento, temos que usar a Taglib `core` do JSF, que é quem possui as Tags que não geram conteúdo HTML:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:body>
        <h:dataTable value="#{automovelBean.automoveis}"
                      var="automovel" border="1">

            <!-- colunas aqui dentro -->

        </h:dataTable>
    </h:body>
</html>
```

Pronto, agora que importamos a nova Taglib por meio do prefixo `Ⓕ`, podemos usar a Tag `facet` indicando qual a característica que será alterada, que nesse caso é o cabeçalho da coluna:

```
<h:column>
  <f:facet name="header">
    Marca
  </f:facet>
  #{automovel.marca}
</h:column>
```

Quando fazemos os cabeçalhos para todas as colunas, temos uma tela como a da figura 3.8.

ID	Marca	Modelo	Ano Fabricação	Ano Modelo	Preço	Observações
1	Volkswagen	Fusca	1966	1966	R\$ 14.200,00	Fusquinha original
2	Fiat	Palio	2000	2001	R\$ 13.000,00	Muito bom
4	Fiat	Palio	2012	2013	R\$ 33.250,00	Modelo novo
5	Chevrolet	Camaro	2013	2013	R\$ 160.000,00	Completo

Figura 3.8: Listagem de Automóveis com borda e cabeçalho

Se quiséssemos, poderíamos mudar mais características, como um rodapé, através da `facet` chamada `footer`. Cada componente pode possuir facetas diferentes, portanto é importante ficar atento no momento de usá-lo.

3.8 MAS MINHA LISTAGEM ESTÁ EXECUTANDO VÁRIAS CONSULTAS NO BANCO...

Agora que temos a lista pronta e funcionando, não temos mais nada para melhorar nessa tela, certo? Errado! Existe uma falha gravíssima nessa funcionalidade que pode fazer com que o projeto, ao ser colocado em um ambiente de produção, tenha baixa performance e até mesmo quase nenhuma escalabilidade, dependendo do caso.

Se analisarmos o registro das consultas que a JPA está disparando no banco de dados, podemos reparar que, quando temos mais de uma informação no `dataTable`, ele **faz uma consulta para cada linha dessa tabela**. Ou seja, se tiver-

mos uma tabela com 100 linhas, serão disparadas 100 consultas no banco de dados, o que é inaceitável para qualquer aplicação.

Isso acontece devido ao funcionamento interno do JSF, que invoca o método `getAutomoveis` para cada linha do `dataTable`. Como solução, podemos proteger a nossa consulta e só fazê-la na primeira vez que chamado o `getAutomoveis`. Assim, nas vezes seguintes, apenas a lista buscada anteriormente é devolvida.

Essa é uma alteração simples de ser feita: basta que no método `getAutomoveis`, antes de fazer a consulta, um `if` verifique se a lista já está populada ou não. Caso esteja vazia, o que vai acontecer na primeira invocação, a consulta deve ser feita - caso contrário, apenas devolve a lista sem realizar a consulta:

```
@ManagedBean
public class AutomovelBean {

    private List<Automovel> automoveis;

    public List<Automovel> getAutomoveis() {
        if(this.automoveis == null) {
            EntityManager em = JPAUtil.getEntityManager();
            Query q = em.createQuery("select a from Automoveis a",
                                    Automovel.class);

            this.automoveis = q.getResultList();
            em.close();
        }
        return automoveis;
    }

    // atributos, método salva(), getter e setter do automovel
}
```

Repare que agora, dentro do método `getAutomoveis`, antes de fazer a consulta, verificamos se o atributo `automoveis` já foi populado. Caso tenha sido, significa que a consulta foi feita e apenas devemos devolver o que foi consultado anteriormente.

3.9 EXCLUSÃO DE DADOS E O COMMANDLINK

Para finalizar, vamos permitir a exclusão dos itens. Para isso precisaremos adicionar uma coluna que irá conter o link “excluir”. Chamaremos essa coluna de “Ações”, já

que futuramente poderemos adicionar outras operações para a mesma linha, como editar ou desabilitar.

O link é construído por meio do componente `commandLink`, que funciona quase como o `commandButton`, mudando apenas a forma de visualização, que passa a ser um *link* em vez de um botão. Vamos definir também a `action` desse *link* para enviar o `automovel` que for clicado na tabela como parâmetro para um método de exclusão dentro do *Managed Bean*:

Com isso, o código para a nova coluna ficará como:

```
<h:column>
    <f:facet name="header">Ações</f:facet>
    <h:commandLink action="#{automovelBean.exclui(automovel)}">
        excluir
    </h:commandLink>
</h:column>
```

Uma regra com relação aos componentes `commandLink` e `commandButton` é que eles devem sempre aparecer dentro de um `form`, caso contrário eles não funcionam. Por isso, vamos envolver o `dataTable` dentro de um formulário:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:body>
        <h:form>
            <h:dataTable value="#{automovelBean.automoveis}"
                          var="automovel">

                <!-- colunas -->

            </h:dataTable>
        </h:form>
    </h:body>
</html>
```

Por fim, falta a implementação do método `exclui` dentro do `AutomovelBean`:

```
public void exclui(Automovel automovel){
    EntityManager em = JPAUtil.getEntityManager();
```

```
Transaction tx = em.getTransaction();

tx.begin();
automovel = em.merge(automovel);
em.remove(automovel);
tx.commit();
em.close();
}
```

Nesse exmplo, o método `merge` é preciso para devolver o objeto ao estado gerenciado. Mas não se preocupe, no capítulo 4 veremos o que isso significa.

3.10 O PRIMEIRO CRUD INTEGRADO

Apesar de simples, esse capítulo serviu para fazermos um CRUD quase completo. Ainda falta a alteração que integra o JSF e a JPA.

Com pouco trabalho de codificação já temos algumas funcionalidades.

Nos próximos capítulos nos aprofundaremos mais nos princípios de cada framework, entenderemos como as ferramentas “pensam” e funcionam, além de evoluirmos mais ainda na nossa aplicação.

Parte II

Domine as ferramentas

Após um primeiro contato com JSF e JPA é hora de entender e dominar essas ferramentas.

CAPÍTULO 4

Entendendo a JPA

No capítulo 2, nós iniciamos um projeto usando JSF e JPA. Tivemos contato com ele, vendo apenas o mínimo necessário para conseguirmos fazer um mapeamento extremamente simples com JPA, bem como telas igualmente simples com JSF. Não vimos conceitos, não entendemos como as coisas funcionam, apenas fomos lá e usamos.

A partir de agora, aprenderemos muito mais sobre a JPA. Veremos o ciclo de vida dos objetos gerenciados pela JPA, como realizar consultas complexas e vários itens aos quais precisamos ficar atentos para não usarmos a ferramenta de forma incorreta.

4.1 O PADRÃO DE PROJETOS DATA ACCESS OBJECT, OU DAO

Quando não utilizamos JPA ou outra ferramenta de mapeamento objeto-relacional, temos que lidar com muito código de infraestrutura - abrir e fechar conexões, tratamento de exceções, SQL, `PreparedStatement` etc. Esse tipo de código nos

obriga a dar uma volta relativamente longa para chegarmos ao código que nos interessa, que é o nosso código do domínio.

Para que possamos separar as responsabilidades e também melhorar a legibilidade do nosso código, pegamos toda a lógica de acesso a dados e colocamos em uma classe que tenha isso como sua razão de existir, o DAO.

Essa classe tem a responsabilidade de prover para a aplicação operações de acesso a dados, como leitura, inclusão, alteração e exclusão.

Utilizando boas práticas de *design*, como programar orientado a interfaces e não a implementações, usamos uma interface para as operações de dados que precisamos e deixamos a implementação desacoplada do nosso código. Isso nos possibilita trocar de implementação sem alterá-lo.

A flexibilidade é tanta que no caso mais simples podemos ter implementações do mesmo DAO para banco de dados diferentes ou para mecanismos diferentes de bancos relacionais, como os bancos *NoSQL* ou mesmo arquivos texto.

A troca de banco de dados durante a vida de um projeto é rara, mas pode acontecer - já tive essa experiência. No entanto, essa facilidade de troca de banco acaba não sendo um apelo forte para o uso do padrão DAO por dois motivos principais. O primeiro é que, como já dito, isso é muito raro de acontecer. E o segundo é que com a JPA temos a possibilidade de mudar de banco como uma característica da ferramenta.

Uma vez que nossos objetos estão mapeados (mapeamento objeto-relacional), devemos trabalhar apenas orientados a objetos, e isso nos deixa livres para mudar de banco. No caso em que passei, por exemplo, não foi preciso mudar nenhuma linha de código nem mesmo do DAO, porque a JPA fez bem o seu papel. **Então, em que caso o DAO ainda se faz útil?**

Um caso em que o DAO se faz útil, hoje, é quando precisamos mudar de um banco relacional para um não relacional, já que a JPA não tem como foco trabalhar com esse tipo de banco de dados. Porém, assim como há uma década atrás mudar de banco de dados era o fator motivador do uso do DAO e hoje não é mais, em algum tempo mudar de mecanismo de persistência, como mudar de banco relacional para banco não relacional, pode também deixar de sê-lo. Já existem projetos que têm como objetivo utilizar as mesmas anotações da JPA para prover mapeamento de objetos para um banco não relacional.

Apesar de o assunto de uso ou não de DAO hoje em dia ser muito polêmico, é fato que ao utilizá-lo separamos melhor as responsabilidades, principalmente quando temos operações de recuperação de dados que são mais complexas que executar uma

consulta.

4.2 JPA NÃO É UM DAO GENÉRICO, É CONTEXTUAL

Muitas pessoas estão familiarizadas com o desenvolvimento de sistemas que acessam banco de dados através de DAOs. A evolução natural dos projetos foi trocar a JDBC por JPA na implementação desses DAOs, mas muitas vezes as pessoas acostumadas a utilizar esses objetos não compreendiam que a JPA não era só uma API que nos ajuda a ter DAOs genéricos e a gerar SQLs de maneira mais fácil.

Há uma grande diferença entre essas APIs. Talvez nesse caso, o papel das interfaces de deixar que o usuário do DAO não precise saber nada da sua implementação tenha contribuído para sistemas com comportamento inesperado. Obviamente que a separação de responsabilidades e a abstração que as interfaces nos proporcionam são benéficas, não há dúvidas em relação a isso.

O caso é que mudar a chave de JDBC para JPA não é tão trivial. É preciso que entendamos as diferenças envolvidas, e é para isso este capítulo serve.

Para que tenhamos algo mais palpável para analisar, observe o código a seguir, que é o mesmo *Managed Bean*, dessa vez utilizando o DAO.

```
@ManagedBean
public class AutomovelBean{
    private AutomovelDao dao;
    private Automovel automovel = new Automovel();

    public AutomovelBean(){
        // caso 1
        dao = new JdbcAutomovelDao();
        // caso 2
        dao = new JpaAutomovelDao();
    }

    public void salvar(Automovel automovel){
        dao.salvar(automovel);
    }
}
```

Como é possível ver, no construtor temos duas opções de instanciação de DAO.

A primeira opção seria a implementação via JDBC e a segunda via JPA. Como tudo está encapsulado dentro do DAO, há pouquíssimo impacto em nosso *Managed Bean*.

Para tirar, por enquanto, as especificidades da Web e tratarmos apenas de JPA, vamos imaginar um cenário que poderia perfeitamente ser executado através do *Managed Bean* apresentado anteriormente. Para vermos melhor, separei o trecho de forma linear.

Fluxo de execução que representa o usuário salvando o automóvel.

```
// inicia a transação

AutomovelDao dao = new // JdbcAutomovelDao() ou JpaAutomovelDao();
Automovel automovel = dao.buscarPorId(100);
automovel.setAnoFabricacao(1996);
dao.salvar(automovel);

// finaliza a transação
```

Nos dois casos estamos representando o cenário onde o usuário seleciona algum automóvel para atualizar, depois muda seu ano de fabricação e em seguida aperta o botão salvar.

Tanto o DAO implementado com JDBC quanto o com a JPA terão comportamentos parecidos, mas no segundo caso é que vem a principal diferença, e que por muitas vezes assusta quem inicia em JPA.

Na implementação com JDBC, o automóvel devolvido é um objeto como qualquer outro, ou seja, não há nada que o vincule com o banco de dados. Então, quando realizarmos alguma modificação nesse objeto e o abandonarmos, ele provavelmente será removido pelo coletor de *Garbage Collector* e o banco de dados nunca saberá dessa alteração.

Já quando buscamos um objeto através da JPA, ele fica vinculado à *EntityManager* que o buscou no banco.

Essa *EntityManager* passa a ter a responsabilidade de manter esse objeto atualizado com o banco de dados. Se alteramos qualquer atributo dele, por meio do vínculo que ele tem com a *EntityManager*, ele acabará sendo salvo no banco.

Para isso acontecer não precisamos pedir novamente que o objeto seja salvo ou sincronizado com o banco. Inclusive se olharmos os métodos da interface *EntityManager*, veremos que existe um método para salvar ou, de acordo com os termos usados pela especificação, para persistir o objeto, o `persist`, mas não temos um método chamado `update`, por exemplo.

Isso acontece porque realizar a atualização faz parte das responsabilidades da `EntityManager`, bastando que exista uma transação para isso, já que é ela quem “dispara” a escrita no banco de dados. Esta é, inclusive, outra diferença em relação a JDBC: a JPA só escreve no banco se houver transação; já utilizando JDBC, apesar de ser possível, o uso de transações é opcional.

Alguns desenvolvedores que já se aventuraram com a JPA antes talvez já tenham ouvido falar do método `merge` e devem se perguntar se ele não seria o equivalente ao `update`, que dissemos que não existe. A resposta para essa pergunta é não, mas vamos ver melhor o porquê estudando o ciclo de vida de um objeto quando ele é gerenciado pela JPA.

4.3 CICLO DE VIDA DE UM OBJETO NA JPA

Na JPA, todo objeto passa por um ciclo de vida que contempla desde o momento em que ele é instanciado pela primeira vez com `new`, passando pelo momento em que ele é salvo no banco e pode chegar até quando ele é removido do banco de dados.

A figura 4.1 serve para mostrar os possíveis estados de um objeto, e qual método da interface `EntityManager` utilizamos para fazer essas transições.

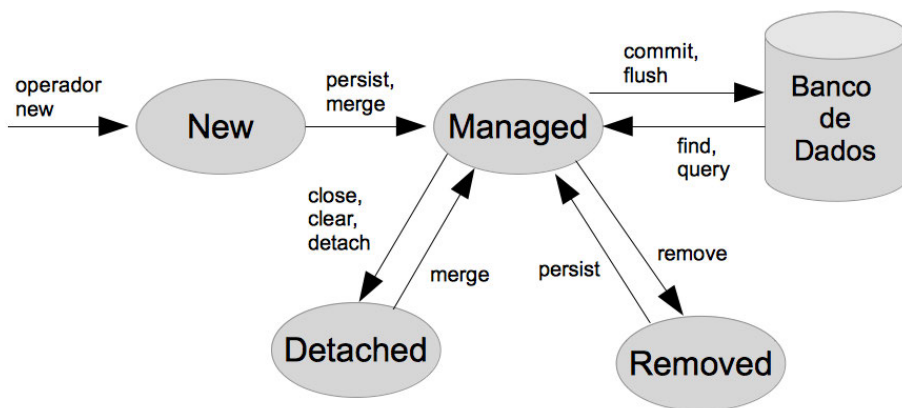


Figura 4.1: Ciclo de vida de um objeto na JPA

Quando criamos um objeto pela primeira vez e ele ainda não passou pela JPA,

dizemos que ele está no estado **New** (novo).

Para que nosso objeto se torne um objeto persistente, precisamos passá-lo para o estado **Managed** (gerenciado) e para isso devemos chamar o método `persist`.

Uma outra maneira de conseguirmos um objeto gerenciado é com o método `merge`, porém temos que nos atentar para uma diferença importante entre `persist` e o `merge`:

```
EntityManager em = JPAUtil.getEntityManager();

// cria o porsche no estado new
Marca porsche = new Marca("Porsche");

// passa o porsche para estado gerenciado
em.persist(porsche);

// cria a ferrari no estado novo
Marca ferrari = new Marca("Ferrari");

//devolve a instância gerenciada
Marca outraFerrari = em.merge(ferrari);

// vai imprimir false
System.out.println(ferrari == outraFerrari);
```

O método `persist` faz com que o objeto passado seja gerenciado, enquanto o método `merge` retorna o objeto que é gerenciado e não faz nada com o objeto passado.

Agora que nosso objeto está gerenciado, ele está no estado em que a JPA mantém um sincronismo entre o objeto e o banco de dados. Porém, como acabamos de passar esse objeto que era novo para gerenciado, ele só vai de fato para o banco quando fizermos `commit` na transação.

Outra forma de termos objetos **Managed** é por meio das buscas no banco. Toda entidade retornada é entregue em estado gerenciado.

Tendo nosso objeto em estado gerenciado, podemos removê-lo do banco através do método `remove`. Feito isso, o objeto passa para o estado **Removed**, mas, novamente, o `delete` no banco de dados só é realizado após o `commit`.

Um dos estados com que mais lidaremos no cotidiano é o **Detached**. Objetos detached são objetos que já foram gerenciados, ou seja, que existem no banco de

dados, mas a `EntityManager` que os trouxe do banco de dados já foi fechada, ou explicitamente desanexou o objeto do seu contexto via o método `detach`.

Mudanças feitas em objetos *detached* não são atualizadas no banco. Para que isso ocorra é necessário retornar o objeto para o estado gerenciado através do método `merge`. No trecho de código a seguir veremos mais detalhes do funcionamento do *detached*.

```
EntityManager em = JPAUtil.getEntityManager();
em.getTransaction().begin();

// Busca o porsche
Modelo porscheDetached = em.find(Modelo.class, 1);

// Desanexa o porsche
em.detach(porscheDetached);

// Modificamos o objeto que está detached
porscheDetached.setDescricao("Porsche 911 Turbo");
porscheDetached.setPotencia(500);

// Buscamos uma instância gerenciada da mesma informação
Modelo porscheGerenciado = em.find(Modelo.class, 1);

// Modificamos o que acabamos de buscar
porscheGerenciado.setDescricao("Porsche 911 T.");

// Reanexa o porsche que havíamos modificado lá no começo
Modelo porscheModificado = em.merge(porscheDetached);

// O porsche detached não é o mesmo objeto que o porsche modificado
System.out.println(porscheDetached != porscheModificado);

// O porsche gerenciado é o mesmo objeto que o porsche modificado
System.out.println(porscheGerenciado == porscheModificado);

em.getTransaction().commit();
```

O código acima serve para nos mostrar porque o método `merge` devolve uma instância em vez de modificar a instância passada por parâmetro. Se em vez de retornar, o `merge` transformasse a própria instância passada por parâmetro,

aconteceria um conflito no contexto da `EntityManager` quando adicionássemos o `porscheDetached`, visto que esse mesmo contexto já possuía a instância `porscheGerenciado`.

Se a `EntityManager` permitisse duas instâncias gerenciadas de `Modelo` com `id=1`, qual seria o `update` no momento do `commit` da transação? Qual objeto considerar?

Para que não haja dúvidas, a `EntityManager` tem sempre um único objeto gerenciado do mesmo tipo e com o mesmo `id`. Por esse motivo, em vez de passar a instância `porscheDetached` para gerenciado, como ela já tinha uma instância gerenciada, `porscheGerenciado`, a `EntityManager` usou esse objeto como base e jogou as alterações feitas no objeto *detached* em cima do gerenciado.

É como se as alterações feitas nos Porsches fossem realizadas no objeto `porscheGerenciado`. Na verdade isso foi feito dentro do método `merge`. Como a instância está gerenciada e foi alterada, no momento do `commit` ela será sincronizada com o banco, ou seja, o comando *update* do SQL será feito.

4.4 A JPA VAI MAIS ALÉM DO QUE UM SIMPLES EXECUTAR DE SQLs

Acabamos de perceber que a JPA faz mais coisas além de executar SQLs. Ela mantém uma organização entre os nossos objetos através desse controle do ciclo de vida deles. É importantíssimo entender todo esse ciclo para que seja possível aprender outros pontos primordiais da JPA.

CAPÍTULO 5

Como mapear tudo... ou nada!

A partir de agora veremos como mapear nosso modelo Orientado a Objetos para o relacional. Normalmente, existe um fator principal que influencia bastante no nosso mapeamento, que é o uso de um banco de dados novo ou de um existente.

Quando vamos criar uma aplicação nova, que não irá reutilizar nenhum banco existente, costumamos ter maior liberdade para apenas refletir no banco as decisões de *design* das nossas classes. Com isso, geralmente o mapeamento é mínimo, e utilizamos muito os comportamentos padrões da JPA.

Outra situação é quando precisamos mapear um banco de dados existente, especialmente quando segue padrões de nomenclatura e de estrutura provenientes de modelo Entidade-Relacionamento tradicional. Com isso podemos entender que as entidades geralmente terão sua chave primária definida por um conjunto de atributos que, combinados, torna a entidade única. Por exemplo, vamos considerar a classe abaixo representando a venda de um automóvel.

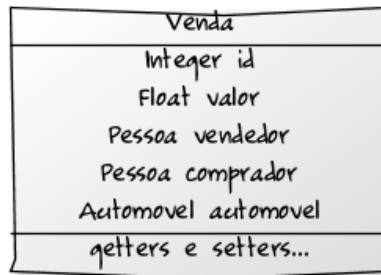


Figura 5.1: Classe venda

A respectiva tabela responsável por armazenar no banco de dados os dados dessa classe pode ser algo como:

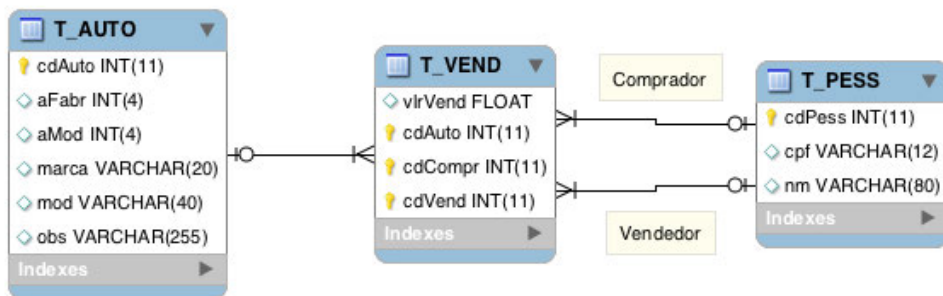


Figura 5.2: DER com nomes abreviados e chave composta

Podemos notar que foi definida uma chave composta que identifica uma venda como única. No exemplo, identificamos que uma venda é representada pelo código do automóvel vendido, pelo código de quem vendeu e pelo código de quem comprou, sendo que esses códigos são chaves estrangeiras que, combinadas, formam a chave primária. Repare também no nome da tabela e nos nomes das colunas, que geralmente seguem um padrão parecido com o apresentado, em que muitas vezes as palavras são abreviadas.

Agora vamos comparar com a tabela que teríamos se utilizássemos apenas as convenções da JPA.

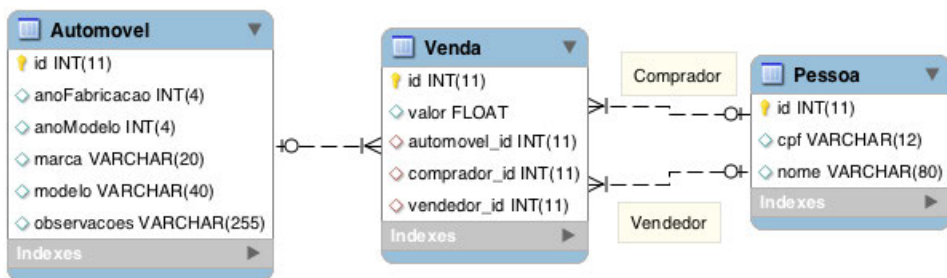


Figura 5.3: DER com nomenclatura padrão JPA

São notórias as mudanças na nomenclatura, que agora reflete os mesmos nomes utilizados na classe. Mas talvez a principal diferença seria utilizar uma chave primária artificial em vez da chave que utiliza os atributos já existentes na tabela.

Claro que nem sempre essa abordagem vai gerar chaves compostas. Por exemplo, no cadastro de uma pessoa, a chave primária poderia ser o CPF ou o e-mail. Mas o fato é que a abordagem da chave artificial autoincrementada tem ganhado muito espaço.

Nesse capítulo vamos ver como trabalhar com o mapeamento, para que ele se adeque da melhor maneira possível à nossa necessidade de banco de dados, seja ela mais próxima das convenções da JPA, seja ela totalmente customizada.

5.1 DEFINIÇÕES DE ENTIDADE

Como já vimos no capítulo 2, geralmente temos a anotação `@Entity` sobre a classe cujas informações dos objetos gostaríamos de persistir em um banco de dados. Elas são o que chamamos de **entidades**.

Além da anotação `@Entity`, precisamos registrar nossa classe no arquivo `persistence.xml`, como a seguir.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
```

```

<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
  <class>facesmotors.entities.Automovel</class>
  <properties>
    ...
  </properties>
</persistence-unit>
</persistence>

```

Esse passo habilita a entidade na `persistence-unit` `default`.

Pela especificação da JPA, ao utilizarmos um servidor de aplicações Java EE, não precisaríamos especificar as entidades no `persistence.xml` desde que tivéssemos apenas uma `persistence-unit`. Na prática, as implementações acabam estendendo isso para o ambiente JavaSE, como o que estamos utilizando. Sendo assim, a declaração das entidades através do `persistence.xml` é opcional na nossa aplicação.

Caso tenhamos mais de uma `persistence-unit`, declaramos todas as entidades em sua respectiva `persistence-unit`. No entanto, ele continuará tentando carregar as classes automaticamente, e com isso, uma `persistence-unit` pode ter classes carregadas de outra. Uma confusão só!

Para evitarmos que se tente carregar automaticamente entidades que não deveriam, já que não foram listadas no `persistence.xml`, utilizamos a configuração `<exclude-unlisted-classes>` com o valor `true`, assim a JPA não tentará atribuir automaticamente entidades na `persistence-unit` errada.

Para exemplificar a existência de mais de uma `persistence-unit`, considere as entidades `Funcionario` e `PrestadorServico`. O primeiro se refere aos funcionários da empresa e tem seus dados em um banco de dados. O segundo se refere a um prestador de serviço ou funcionário terceirizado e possui seus dados armazenados em outro banco de dados. Nesse caso, a configuração do `persistence.xml` ficaria como a seguir:

```

<persistence...>
  ...
  <persistence-unit name="funcionarios"
                    transaction-type="RESOURCE_LOCAL">
    <class>sistemarh.entities.Funcionario</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      ...
    </properties>
  </persistence-unit>
</persistence>

```

```

</persistence-unit>

<persistence-unit name="terceirizados"
                  transaction-type="RESOURCE_LOCAL">
  <class>sistemarh.entities.PrestadorServico</class>
  <exclude-unlisted-classes>true</exclude-unlisted-classes>
  <properties>
    ...
  </properties>
</persistence-unit>
</persistence>

```

E com isso nenhuma classe é carregada sem que indiquemos que ela realmente deve ser carregada.

5.2 FAÇA ATRIBUTOS NÃO SEREM PERSISTIDOS COM O @TRANSIENT

Por padrão, toda propriedade de uma entidade é persistente, a menos que especifiquemos o contrário utilizando a anotação `@Transient` ou a palavra reservada `transient` no atributo. Essa configuração é extremamente útil quando temos um campo que é calculado. Considere a entidade `Funcionario`:

```

@Entity
public class Funcionario {

    @Id @GeneratedValue
    private Long id;

    private Calendar dataNascimento;

    private int idade;
}

```

Nesse caso, se utilizássemos esse mapeamento, a JPA esperaria que houvesse uma tabela com 3 campos: `id`, `dataNascimento` e `idade`. No entanto, repare que há uma redundância entre esses campos. A `idade` pode facilmente ser calculada a partir da data de nascimento. Imagine o trabalho que seria ter que manter os dados da `idade` atualizados.

Nesses casos, podemos indicar que a `idade` não será persistida no banco de dados, através da anotação `@Transient`:

```
@Transient
private int idade;
```

Opcionalmente, podemos usar a palavra chave `transient`:

```
private transient int idade;
```

Pronto, agora a tabela possui apenas as colunas `id` e `dataNascimento`.

5.3 MAPEIE CHAVES PRIMÁRIAS SIMPLES

Também já vimos superficialmente como lidar com chaves primárias, mas agora vamos nos aprofundar. A configuração mínima é a anotação `@Id`.

```
@Entity
public class Automovel {

    @Id
    private String placa;
    //outros atributos e métodos
}
```

Nesse caso teremos uma chave primária na qual a própria aplicação deve informar o valor antes de persistir no banco. No nosso exemplo, utilizamos a placa de um automóvel, mas poderíamos utilizar o CPF como chave de uma entidade chamada `Pessoa` e assim por diante.

Chaves com valores autogerados

No exemplo do automóvel, por exemplo, poderíamos tê-lo cadastrado com a informação da placa errada. Mas como se trata da chave primária, mudar essa informação pode não ser tão simples, já que ela pode estar sendo referenciada como chave estrangeira em outra tabela. Por isso, e para evitar a complexidade de chaves compostas, é que cada vez mais se tem utilizado as chaves autoincrementadas. Bastaria adicionar a anotação `@GeneratedValue`.

```
@Entity
public class Automovel {
```



```
@Id @GeneratedValue
private Integer id;
private String placa;
//outros atributos e métodos
}
```

Agora a placa é um atributo como qualquer outro, e adicionamos uma chave artificial que, devido à presença da anotação `@GeneratedValue`, não precisa - e não deve - ser informada quando formos persistir um `Automovel`.

Apenas com isso a JPA já irá com a estratégia de geração de valores de cada banco de dados. Em alguns casos como MySQL e SQLServer a JPA criará chaves do tipo `auto-increment` e `identity`, respectivamente. Em bancos como PostgreSQL e Oracle serão geradas `sequences` para incrementar a chave.

Em bancos que não possuem suporte a `sequence` e nem a valores autogerados, é possível ainda termos uma tabela no banco que gerencie o contador da chave de todas as outras tabelas.

A essas diferentes formas de implementar uma chave cujo conteúdo é gerenciado pela JPA é dado o nome de **estratégia**.

A JPA possui basicamente 4 tipos de estratégias: `AUTO`, `IDENTITY`, `SEQUENCE` e `TABLE`. No entanto, é importante lembrar que `SEQUENCE` e `IDENTITY` podem não ser compatíveis com o banco de dados que você estiver usando. Todas essas estratégias são representadas através da `enum GenerationType`:

- **Deixe a JPA escolher:**

```
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private Integer id;
```

O tipo `AUTO` não chega a ser uma estratégia real, ela serve para deixar a JPA decidir a melhor estratégia dependendo do banco utilizado, tendo assim uma maior portabilidade. Além disso, quando não definimos a estratégia, esse é o valor utilizado.

- **Use valores auto incremento**

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer id;
```

Especificando a estratégia como `IDENTITY` forçamos a JPA a utilizar colunas com valores autoincrementáveis. Atente-se ao fato de que alguns bancos de dados podem não suportar essa opção.

- Gere valores auto incremento através de sequences

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE)
private Integer id;
```

Nesse caso, estaremos configurando nossa chave para trabalhar com uma `SEQUENCE` do banco de dados. O comportamento padrão é uma `SEQUENCE` global para a aplicação, em que todas as tabelas compartilhariam uma mesma sequência. Por exemplo, ao salvar um `Automovel` ele receberia o id 1; depois salvando um `Acessorio` ele receberia id 2; e ao salvar outro `Automovel` ele receberia id 3.

Em diversos casos, essa abordagem não chega a ser um problema, mas podemos querer determinar um gerador de chave específico para cada tabela; ou ainda deixar tabelas simples com o gerador global e algumas específicas com um gerador individual. Para fazer essa configuração, basta darmos um nome diferente para cada gerador de ids que desejarmos, pelo atributo `generator` na anotação `@GeneratedValue`.

```
@Entity
public class Automovel {
    @SequenceGenerator(name="automovelGenerator",
                      sequenceName="AUTOMOVEL_SEQ",
                      allocationSize=10)
    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE,
                      generator="automovelGenerator")
    private Integer id;
    //outros atributos e métodos
}
```

```
@Entity
public class Acessorio {
    @SequenceGenerator(name="acessorioGenerator",
                      sequenceName="ACESSORIO_SEQ",
                      allocationSize=10)
    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE,
                      generator="acessorioGenerator")
```

```
private Integer id;
//outros atributos e métodos
}
```

Dessa forma, teremos cada entidade seguindo uma sequência de `id` própria. Vimos ainda a anotação `@SequenceGenerator`, que usamos para especificar alguns detalhes da sequência, como o nome e sua `allocationSize`, onde definimos 10. Isso significa que a `SEQUENCE` será chamada pela primeira vez e devolverá 1, mas já serão buscados 10 números. Para os próximos 9 `id`, eles serão buscados da memória, e apenas após isso, a `SEQUENCE` seria chamada novamente. Isso evita diversas idas ao banco para gerar a chave. Se não especificarmos nada, o tamanho padrão é 50.

Uma outra opção interessante para o `@SequenceGenerator` é a definição do valor inicial da sequência, através do atributo `initialValue`.

- Use uma tabela auxiliar

```
public class Automovel {

    @TableGenerator(
        name="AUTOMOVEL_GENERATOR",
        table="GENERATOR_TABLE",
        pkColumnValue="AUTOMOVEL"
    )
    @Id @GeneratedValue(strategy=GenerationType.TABLE,
        generator="AUTOMOVEL_GENERATOR")
    private Integer id;
    //outros atributos e métodos
}
```

Quando utilizamos a estratégia `Table` temos uma tabela que irá gerenciar as chaves da nossa entidade. Podemos ter uma mesma tabela para todo o sistema, que irá diferenciar o dono da chave pelo conteúdo do atributo `pkColumnValue` da `@TableGenerator`.

Podemos também ter mais de uma tabela de controle. O único atributo obrigatório é o nome do gerador, e para todo o resto podemos não especificar nada e deixar a convenção fazer seu trabalho.

5.4 MAPEIE CHAVES COMPOSTAS

Em alguns casos podemos ter chaves compostas em nossas entidades. Isso é comum quando mapeamos um banco pré-existente, mas também pode acontecer quando, por exemplo, desejamos mapear uma tabela de relacionamento muitos para muitos. Esse último caso costuma aparecer quando precisamos ter um atributo no relacionamento, e a chave dessa entidade será composta das duas chaves do relacionamento.

Obviamente podemos ter chaves compostas em outros casos, mas como exemplo tomaremos um relacionamento muitos para muitos de `Consumidores` que tem `Produtos` em sua lista de desejos. Esse relacionamento tem um atributo que especifica se o `Consumidor` deseja receber e-mail a cada alteração feita no `Produto` que ele observa.

Como temos um atributo no relacionamento, iremos mapeá-lo como uma entidade `ConsumidorProduto`, e em vez de mapearmos um muitos para muitos como veremos na seção 5.17, mapearemos um para muitos de `Consumidor` para `ConsumidorProduto` e outra relação um para muitos de `Produto` para `ConsumidorProduto`. O mapeamento um para muitos será visto na seção 5.13.

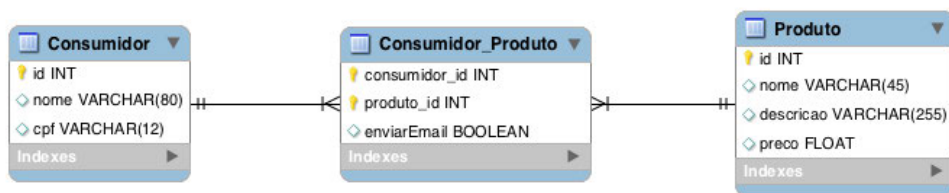


Figura 5.4: Consumidor e Produto com tabela de relacionamento

Lidando com chaves compostas temos duas formas de fazer o mapeamento, via `@EmbeddedId` ou via `@IdClass`. Em ambos os casos precisaremos de uma classe que implemente a interface `Serializable` para representar a chave. Isso porque na JPA a chave primária é representada por uma instância de `Serializable`. Podemos ver isso de forma mais clara na declaração dos métodos `find` e `getReference`, que possuem como objetivo recuperar uma instância de uma entidade no banco de dados através do seu `id`:

```
public interface EntityManager {
    ...
}
```

```

    <T> find(Class<T> clazz, Serializable id);
    <T> getReference(Class<T> clazz, Serializable id);
    ...
}

```

Então, por mais que nossa chave contenha dois ou mais atributos, precisamos de uma única classe que os represente. Veremos agora as pequenas diferenças envolvendo cada uma dessas formas de mapear as chaves compostas.

Mapeamento de chave composta com o @EmbeddedId e @Embeddable

Mapeando dessa forma, em vez de termos uma chave do tipo `Integer` ou `Long`, teremos uma chave do tipo da classe que criamos. Ao analisar os exemplos fica simples de se compreender.

```

@Embeddable
public class ConsumidorProdutoPK implements Serializable {
    ...
    private Integer consumidorId;
    private Integer produtoId;

    // equals e hashCode
}

```

Note que não anotamos a chave com `@Entity`, afinal ela não representa uma entidade, apenas sua chave. Para esse caso, anotamos com `@Embeddable`, indicando que essa classe complementarará outra, que no caso, será a classe `ConsumidorProduto`.

```

@Entity
public class ConsumidorProduto {

    @EmbeddedId
    private ConsumidorProdutoPk id;
    // outras propriedades e métodos
}

```

Para buscar usando a chave primária, basta passarmos uma instância da chave como no exemplo a seguir.

```

ConsumidorProdutoPK pk = new ConsumidorProdutoPK();
pk.setConsumidorId(consumidorId);

```

```
pk.setProdutoId(produtoId);
```

```
ConsumidorProduto cp = entityManager.find(ConsumidorProduto.class, pk);
```

Agora para realizarmos uma busca de `ConsumidorProduto` baseado na chave de algum consumidor, teríamos a seguinte consulta.

```
String jpql = "select cp from ConsumidorProduto as cp " +  
              "where cp.id.consumidorId = :consumidor";
```

Como podemos perceber, para acessar o id do consumidor, precisamos acessar a propriedade do tipo `ConsumidorProdutoPK` através de `"cp.id"`. Não se assuste com essa sintaxe diferente no `where` caso ainda não a conheça, ainda vamos falar bastante sobre ela.

Mapeamento de chave composta com o `@IdClass`

A principal diferença no uso de `@IdClass` com relação ao `@EmbeddedId` é que neste último precisávamos acessar uma propriedade intermediária para chegar às propriedades que compõem a chave.

No exemplo apresentado, por se tratar de uma tabela de relacionamento, isso pode não parecer tão mal. Mas quando temos uma tabela de chave composta na qual as colunas que compõem a chave são atributos da entidade, isso passa a ser um problema.

Considere um sistema de uma escola que tem uma tabela que armazena os dados de cada aluno. No entanto, esses alunos podem ser muito jovens e não possuir documentos, então uma possível solução seria criar a chave baseada no nome do aluno, o nome da mãe e a data de nascimento do aluno. Além dessas informações teríamos outras como podemos ver a seguir.

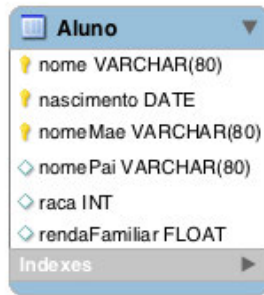


Figura 5.5: Tabela Aluno com chave tripla

Agora, se usássemos a abordagem que vimos com `@EmbeddedId` e quiséssemos consultar crianças em uma determinada faixa de idade, com renda familiar entre outra determinada faixa, precisaríamos de uma consulta parecida com a seguinte:

```
String jpql = "select a from Aluno a " +
    "where a.id.nascimento between :dataInicial and :dataFinal " +
    "and a.rendaFamiliar between :rendaMinima and :rendaMaxima";
```

Vemos que o tratamento dado às propriedades `nascimento` e `rendaFamiliar` foi diferente pelo fato de uma fazer parte da chave primária. Em casos como esse, o uso da abordagem que vamos ver agora resulta em tratamento igual a qualquer propriedade. Vejamos como fica o mapeamento dessa forma.

```
public class AlunoPK implements Serializable {
    ...
    private String nome;
    private String nomeMae;
    private Date nascimento;

    // equals e hashCode
}
```

```
@Entity
@IdClass(AlunoPK.class)
public class Aluno {
```

```
    @Id
```

```

    private String nome;
    @Id
    private String nomeMae;
    @Id
    private Date nascimento;
    private String nomePai;
    private String raca;
}

```

Dessa forma nós continuamos tendo uma classe que representa a chave para que possamos utilizar os métodos que esperam a chave primária como o `find` e o `getReference` da interface `EntityManager`. Mas como podemos observar, os mesmos atributos que estão na classe `AlunoPK` estão também na classe `Aluno`, o que nos permite refazer a pesquisa anterior da seguinte forma:

```

String jpql = "select a from Aluno a " +
    "where a.nascimento between :dataInicial and :dataFinal " +
    "and a.rendaFamiliar between :rendaMinima and :rendaMaxima";

```

Agora sim, podemos pesquisar atributos simples ou parte da chave da mesma forma.

5.5 A ANOTAÇÃO @BASIC

Utilizamos essa anotação para informar que uma propriedade da entidade é uma coluna no banco. No entanto, como esse já é o comportamento padrão, o uso dessa anotação é opcional e acaba ficando restrito a quando desejamos que mesmo uma propriedade comum, que não é um relacionamento, seja carregada sob demanda - ou também conhecido como *Lazy Load*, que vamos estudar mais na seção 5.14.

```

@Entity
public class Automovel {

    private Integer id;
    private String descricao;
    ...
    @Id @GeneratedValue
    public Integer getId(){
        return id;
    }
}

```



```
@Basic(fetch=LAZY)
public String getDescricao(){
    return descricao;
}

}
```

Com isso, a descrição só será buscada no banco de dados quando invocarmos o seu *getter* pela primeira vez. Porém percebemos uma diferença nesse código, que é a anotação estar no *getter* e não na propriedade. Precisamos disso porque o carregamento sob demanda de propriedades da própria entidade só funciona se anotarmos o *getter*. Anotando a propriedade, o carregamento será feito normalmente junto com a entidade.

5.6 @TABLE, @COLUMN E @TEMPORAL

A convenção da JPA indica que as colunas do banco terão o mesmo nome dos atributos das entidades, e é muito comum encontrar desenvolvedores que para deixar as classes mais limpas, sem anotações, adotam esse comportamento padrão. No entanto, quando mapeamos tabelas ligadas, muitas vezes o padrão de nomenclatura é diferente. Por exemplo, podemos ter nossa classe e nossa tabela da seguinte forma:

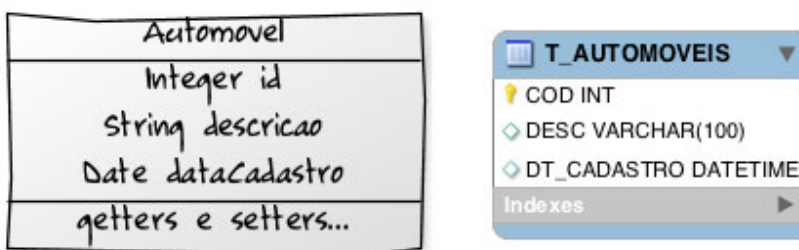


Figura 5.6: Exemplo de classe e tabela de Automovel lado a lado

Portanto, precisamos de alguma maneira fazer com que nossa entidade seja persistida nessa tabela. Claro que não vamos mudar os nomes dos atributos da entidade

e acabar abrindo mão da clareza no código Java. A solução é simples, bastam alguns mapeamentos.

```
@Entity
@Table(name="T_AUTOMOVEIS")
public class Automovel {

    @Id @GeneratedValue
    @Column(name="COD")
    private Integer id;

    @Column(name="DESC", nullable=false, length=100)
    private String descricao;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name="DT_CADASTRO", nullable=false, updatable=false)
    private Date dataCadastro;
    ...
}
```

Para mapear o nome da tabela, usamos a anotação `@Table`, que nos permite também informar o `schema` do banco, caso necessário. Já para mapear a coluna, usamos a anotação `@Column`, que nos permite colocar vários detalhes de como essa propriedade Java vai virar uma coluna no banco.

Ao mapear a coluna podemos especificar, além do nome, se a coluna aceita valores nulos ou não, por meio do atributo `nullable`, e o seu tamanho, com o `length`. Essas alterações refletem na estrutura da tabela, ou seja, a descrição em que indicamos o tamanho 100 e que não é anulável será `NOT NULL` e possivelmente um `VARCHAR(255)` dependendo do banco de dados em uso.

Outra configuração interessante que fizemos é indicar se a coluna pode ter seu valor atualizado ou não. Essa última configuração não será refletida no banco de dados em si, mas sim na geração dos comandos de `update` no banco. Mesmo que o usuário ou a aplicação alterem o valor da propriedade `dataCadastro`, quando o `Automovel` for atualizado no banco, o comando `update` irá ignorar essa propriedade.

É possível até especificar um trecho de SQL nativo do banco para gerar a coluna, para isso podemos usar a propriedade `columnDefinition`, mas tome cuidado, porque você poderá perder a portabilidade entre os bancos de dados:

```
@Column(name="DESC", columnDefinition="CLOB NOT NULL")  
private String descricao;
```

E por fim, no nosso mapeamento temos a configuração da precisão de data que iremos utilizar. Fazemos isso usando a anotação `@Temporal`. Esse mapeamento pode ser usado em atributos do tipo `Date` ou `Calendar` e com ele podemos especificar se ao conversar com o banco de dados queremos considerar:

- `DATE`: somente a data, sem informação de hora;
- `TIME`: somente informação de hora, sem data;
- `TIMESTAMP`: valor padrão, que considera tanto data quanto hora.

O *datatype* a ser utilizado pode variar de acordo com o banco de dados em uso.

5.7 @VERSION E LOCK OTIMISTA

Existem três formas de tratarmos concorrência de alteração de registros no banco de dados. A primeira, e mais usada, é não fazer nada. Nesse caso, se dois usuários leem a mesma versão de um registro, fazem alguma alteração e depois salvam, a alteração feita primeiro irá se perder, já que será sobrescrita pelo que salvar por último.

A outra opção é o que chamamos de *lock* pessimista. Ele tem esse nome porque parte do princípio que provavelmente irá ocorrer uma concorrência na edição do registro. Quando alguém o lê, ele já fica bloqueado somente para essa pessoa. Dessa forma, evita-se que outros usuários também editem o mesmo registro.

Muitas vezes quando carregamos as informações para o usuário, ele tem na mesma tela a opção de visualizar ou editar. A simples visualização não causa problema de concorrência, mas como estamos sendo pessimistas, acabamos considerando que quando esse registro for lido provavelmente ele será alterado, então esse bloqueio é feito para escrita. O problema é que quando outro usuário for visualizar o mesmo registro, por estar usando a mesma tela que o primeiro, vamos querer um bloqueio de escrita para ele também. Mas como o registro já está bloqueado, o segundo usuário não consegue nem ler.

Isso fica pior ainda se o banco de dados utilizado não suportar bloqueio de linha. Nesse caso, por mais que solicitemos o bloqueio de uma única linha, a tabela inteira fica bloqueada, causando um problema enorme na aplicação. Em um sistema razoavelmente utilizado, em poucos minutos teríamos “engarrafamentos quilométricos” no nosso banco de dados.

A terceira opção é usarmos o *lock* otimista. Apesar do nome, não existe um bloqueio de registro nessa abordagem: em vez disso criamos uma propriedade, do tipo numérico ou data, e anotamos com `@Version`. Assim, a cada alteração do registro essa propriedade será atualizada, e dessa forma, o banco fica parecido com um sistema de controle de versão.

Caso dois usuários leiam o mesmo objeto na versão 1, quando o primeiro salvar a alteração esse objeto passa para a versão 2 no banco. Então, quando o segundo usuário for persistir suas alterações, a JPA vai perceber que ele está tentando salvar uma alteração baseada na versão 1, mas que já existe uma versão mais nova no banco. Ela lançará uma `OptimisticLockException` e a transação será marcada para *rollback*. Apesar de todo o conceito envolvido, o mapeamento é bem simples.

```
@Entity
public class Automovel {

    @Id @GeneratedValue
    private Integer id;

    private String descricao;

    @Column(nullable=false, updatable=false)
    private Date dataCadastro;

    @Version
    private Date ultimaAlteracao;
    ...
}
```

Com a configuração realizada, é responsabilidade da aplicação fazer o tratamento dos possíveis conflitos. Uma boa opção é apresentar ao usuário, em uma tela comparativa, a sua versão e a versão mais nova para que ele escolha como quer deixar as informações.

5.8 RELACIONAMENTOS MUITOS PARA UM COM @ManyToOne

Utilizamos a anotação `@ManyToOne` sempre que quisermos relacionar entidades. Para analisarmos como usamos esse tipo de relacionamento, considere a classe

Modelo de automóvel, que por sua vez tem uma relação com `Marca`. Assim podemos dizer que o `Modelo 458 Spider` é da `Marca Ferrari`.

```
@Entity
public class Modelo {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    private Marca marca;
}

@Entity
public class Marca {
    @Id @GeneratedValue
    private Long id;
}
```

O resultado desse mapeamento no banco de dados é o seguinte:

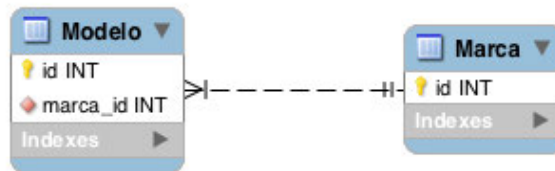


Figura 5.7: Tabela Modelo com chave estrangeira de Marca

Na tabela `Modelo` temos uma chave estrangeira chamada `marca_id` que faz referência para a tabela `Marca`. Esses nomes estão seguindo a nomenclatura *default* definida pela JPA.

Como já vimos, a tabela criada segue o mesmo nome da entidade e o mesmo vale para as colunas simples, que têm o mesmo nome dos atributos. A novidade aqui é a chave estrangeira, mas seu nome padrão também é bem simples. O nome `marca_id` é formado a partir da concatenação do nome da propriedade que estamos mapeando, que no caso é `marca`, seguido de “_”, e em seguida o nome da coluna que representa a chave primária da entidade `Marca`, que no caso é `id`.

No exemplo, a propriedade `marca` tem o mesmo nome da sua classe: `Marca`, exceto por iniciar em minúsculo.

Outro ponto importante a se observar é que é o nome da propriedade que é considerado para a especificação do nome da coluna da chave estrangeira no banco, e não da classe. Assim como a outra parte do nome da chave estrangeira é formada pelo nome da coluna da chave primária na tabela `Marca`, e não o nome da propriedade na classe `Marca`.

Vejamos o exemplo:

```
@Entity
@Table(name="T_MODELO")
public class Modelo {
    @Id @GeneratedValue
    private Integer id;

    @ManyToOne
    private Marca montadora;
}

@Entity
@Table(name="T_MARCA")
public class Marca {
    @Id @GeneratedValue @Column(name="pk")
    private Integer id;
}
```

Com esse novo exemplo, o novo nome da coluna que é chave estrangeira de `T_MARCA` é “`montadora_pk`”. Devemos observar que essa mesma regra para criação de chave estrangeira com nome *default* se aplica a todos os outros tipos de mapeamento.

O exemplo da classe `Marca` apresentado aqui é apenas para vermos um nome diferente de coluna de chave primária. Na seção 5.13 veremos como mapeá-la.

5.9 @ONEToOne

O mapeamento `@OneToOne` é bem parecido com o `@ManyToOne` que vimos anteriormente. A principal diferença é que na chave estrangeira teremos a restrição

unique. Ainda considerando a `Marca` do automóvel, podemos dizer que ela terá seus detalhes e criaremos uma nova entidade para isso, chamada `DetalheMarca`, que guardará as informações extras sobre ela. Como queremos que a `Marca` só esteja relacionada a um `DetalheMarca`, vamos indicar que essa relação é `@OneToOne`:

```
@Entity
public class DetalheMarca {
    @Id @GeneratedValue
    private Integer id;
    private Integer anoFundacao;
    private String paisOrigem;
    private String fundador;

    @OneToOne
    private Marca marca;
}
```

E logo após o DER que representa essa relação.

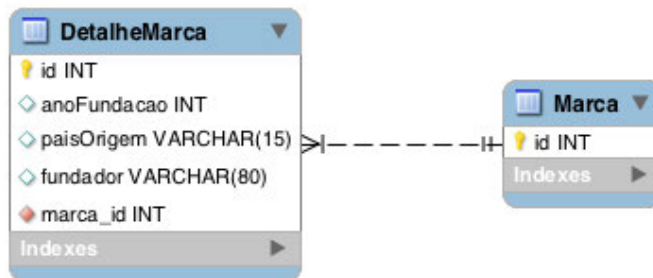


Figura 5.8: Tabela `DetalheMarca` com a chave estrangeira da `Marca`

Nesse mapeamento temos uma classe `DetalheMarca` cujas propriedades poderiam estar dentro da própria classe `Marca`, mas optamos por separar em tabelas e classes diferentes para uma melhor organização.

A vantagem desse tipo de abordagem é que evitamos classes e tabelas com muitos atributos e colunas. No entanto, podemos acabar gerando `joins` demais quando nossas consultas envolverem as duas entidades.

Indo além nesse exemplo, o que acontece se tivermos um objeto `Marca` e quisermos acessar seu fundador?

Na situação apresentada, temos o `DetalheMarca` mapeando a `Marca`, mas não o contrário. Temos um mapeamento unidirecional e gostaríamos de ter um relacionamento bidirecional.

@OneToOne unidirecional

Como acabamos de ver, um mapeamento unidirecional consiste em mapearmos apenas um dos lados do relacionamento. Esse lado terá em sua tabela um chave estrangeira referenciando a tabela do outro lado do relacionamento.

Anteriormente, a classe `DetalheMarca` mapeava a `Marca` e por isso a tabela `DetalheMarca` tinha uma chave estrangeira para a tabela `Marca`.

5.10 RELACIONAMENTOS BIDIRECIONAIS

Já sabemos que apenas `DetalheMarca` tem um acesso direto à `Marca`, mas gostaríamos que ambos os lados se enxergassem.

```
@Entity
public class DetalheMarca {
    @Id @GeneratedValue
    private Integer id;
    private Integer anoFundacao;
    private String paisOrigem;
    private String fundador;

    @OneToOne
    private Marca marca;
}

@Entity
public class Marca {
    @Id @GeneratedValue
    private Integer id;

    @OneToOne
    private DetalheMarca detalhe;
}
```

A seguir podemos ver que esse óbvio mapeamento resulta em algo inesperado.

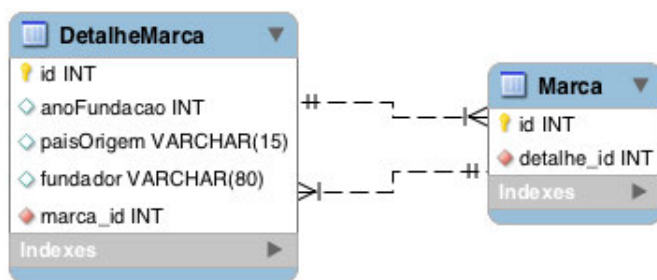


Figura 5.9: Mapeamento errado cria chave estrangeira nas duas tabelas

O que aconteceu aqui é que cada tabela possui sua própria chave estrangeira apontando para a outra, em vez de uma única em um dos lados. Quando há um relacionamento bidirecional, devemos escolher um dos lados para ser o dono e nesse caso somente a tabela do dono é que terá a chave estrangeira.

Por mais que ambos os lados tenham uma instância do outro lado, o desejável é que no banco de dados tenhamos apenas uma chave estrangeira. Como é um mapeamento `@OneToOne`, podemos escolher qualquer um dos lados. Para nosso modelo, vamos deixar a `Marca` como a dona. Com esse ajuste, o mapeamento e as tabelas ficam da seguinte forma.

```
@Entity
public class DetalheMarca {
    @Id @GeneratedValue
    private Integer id;
    ...
    @OneToOne(mappedBy="detalhe")
    private Marca marca;
    ...
}
```

```
@Entity
public class Marca {
    @Id @GeneratedValue
    ...
    @OneToOne
    private DetalheMarca detalhe;
    ...
}
```

Analisando o código, o lado que não é o dono indica quem tem esse papel através do atributo `mappedBy`. Vemos que na classe `DetalheMarca` nós dizemos que o dono do relacionamento é o outro lado, e que a informação que seria persistida em uma chave estrangeira “própria” agora vai ser mapeada pela chave estrangeira criada pela propriedade `detalhe` da classe `Marca`.

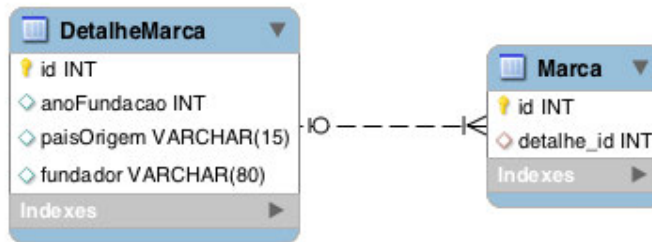


Figura 5.10: Agora quem tem a chave estrangeira é só a tabela `Marca`

5.11 O QUE TEM A VER O DONO DO RELACIONAMENTO COM A OPERAÇÃO EM CASCATA?

Vimos que é importante termos um dono para o relacionamento bidirecional para que não tenhamos duas chaves estrangeiras no banco de dados, o que causaria problema graves na aplicação. Agora veremos que, apesar de termos a possibilidade de usar operações em cascata, isso de forma alguma diminui a importância do papel do dono do relacionamento.

Podemos configurar o mapeamento de relacionamento com a opção de cascata como podemos ver a seguir.

```

@Entity
public class DetalheMarca {
    @Id @GeneratedValue
    private Integer id;

    @OneToOne(mappedBy="detalhe", cascade=CascadeType.ALL)
    private Marca marca;
}

@Entity

```

```
public class Marca {  
    @Id @GeneratedValue  
    private Integer id;  
  
    @OneToOne(cascade=CascadeType.ALL)  
    private DetalheMarca detalhe;  
}
```

Repare no atributo `cascade` com o valor `CascadeType.ALL` nos relacionamentos. Com isso, qualquer operação, como por exemplo `persist`, `remove`, `merge` que fizermos na `Marca` será feito também no `DetalheMarca`; por isso não precisamos chamar o `persist` nos dois lados.

Podemos especificar somente algumas operações, possibilitando propagar a operação de salvar mas não a de excluir. Mas como estamos trabalhando com `@OneToOne`, o mais comum será a opção `ALL`, já que ambos os lados representam uma mesma informação, que só foi dividida para melhorar a manutenibilidade do sistema.

Apesar da facilidade que essa funcionalidade nos oferece, é importante lembrarmos que continua existindo um comportamento diferente em cada um dos lados da relação. Como temos a `Marca` como a dona do relacionamento, precisamos obrigatoriamente alimentá-la com o `DetalheMarca`. Sem isso, a informação da chave estrangeira não será persistida.

A seguir veremos mais alguns cenários para não restar dúvidas sobre esse assunto tão importante.

Primeiro cenário: Fechando os dois relacionamentos e persistindo o lado fraco

```
Marca marca = new Marca("Ferrari");  
DetalheMarca detalhe = new DetalheMarca("Enzo Ferrari");  
  
marca.setDetalhe(detalhe);  
detalhe.setMarca(marca);  
entityManager.persist(detalhe);
```

Nesse código, nós relacionamos ambos os lados: `Marca` e `DetalheMarca`, mas acabamos salvando o lado que não é o dono do relacionamento. É necessário entendermos que o importante não é o lado que é passado para o método `persist` e sim que o dono conheça o outro lado.

Aqui mandamos salvar o `detalhe`, mas como configuramos o comportamento em cascata para a sua propriedade `marca`, é como se também tivéssemos mandado persistir do outro lado. E o melhor é que a JPA infere a ordem correta em que essas operações precisam ser executadas no banco. Então, por mais que tenhamos chamado o método `persist` do lado “fraco” o outro lado tem a informação necessária para persistir a chave estrangeira.

Segundo cenário: Persistindo lado forte

```
Marca marca = new Marca("Ferrari");
DetalheMarca detalhe = new DetalheMarca("Enzo Ferrari");

marca.setDetalhe(detalhe);
entityManager.persist(marca);
```

Nesse cenário salvamos a `marca`, e como indicamos que gostaríamos das operações em cascata, quando lermos o `DetalheMarca` do banco de dados, veremos que a informação da marca estará lá, já que a chave estrangeira será alimentada corretamente.

Terceiro cenário: Não fechando o relacionamento e persistindo o lado fraco

```
Marca marca = new Marca("Ferrari");
DetalheMarca detalhe = new DetalheMarca("Enzo Ferrari");

marca.setDetalhe(detalhe);
entityManager.persist(detalhe);
```

Agora temos um código muito parecido com o que vimos no primeiro cenário, sendo que a diferença é que agora associamos apenas o lado forte da relação.

Como o `detalhe` nesse exemplo não se relaciona com a `marca`, esta não será persistida, e consequentemente a relação entre `marca` e `detalhe` não existirá. A solução poderia ser exigirmos a presença de uma `Marca` para todo `DetalheMarca`; e de fato é o mais correto a se fazer nesse caso.

Quarto cenário: Novamente não fechando o relacionamento e persistindo o lado fraco

```
Marca marca = new Marca("Ferrari");
DetalheMarca detalhe = new DetalheMarca("Enzo Ferrari");

detalhe.setMarca(marca);
entityManager.persist(detalhe);
```

Nesse exemplo, ambos os lados serão persistidos mas a informação da chave estrangeira não será alimentada. Com isso, quando lermos tanto a `Marca` quanto o `DetalheMarca` do banco de dados, ambos virão com o outro lado nulo. Alimentar o `detalhe` com sua `marca` nesse caso serve apenas para a operação em cascata ser disparada, mas quando a `marca` for persistida, como ela não contém a informação do seu `detalhe`, a chave estrangeira não será alimentada.

Resumindo as operações em cascata

Uma prática muito comum para evitar esses problemas é fazer com que os *setters* dos relacionamentos definam os dois lados do relacionamento, assim não correríamos riscos de esquecer de preencher algum lado. Essa mudança é extremamente simples.

```
@Entity
public class DetalheMarca {
    @Id @GeneratedValue
    private Integer id;

    @OneToOne(mappedBy="detalhe", cascade=CascadeType.ALL)
    private Marca marca;

    public void setMarca(Marca marca) {
        this.marca = marca;
        marca.setDetalhe(this);
    }
}
```

```
@Entity
public class Marca {
    @Id @GeneratedValue
    private Integer id;

    @OneToOne(cascade=CascadeType.ALL)
    private DetalheMarca detalhe;
```

```
public void setDetalhe(DetalheMarca detalhe) {  
    this.detalhe = detalhe;  
    detalhe.setMarca(this);  
}  
}
```

Pronto, agora toda vez que alguém invocar algum dos *setters*, automaticamente os dois lados dos relacionamento estarão definidos.

5.12 ORGANIZE MELHOR SUAS ENTIDADES E PROMOVA REAPROVEITAMENTO COM @EMBEDDABLE E @EMBEDDED

Já vimos que a associação @OneToOne geralmente é usada para organizarmos melhor nossas classes e normalizar ainda mais as tabelas. Assim, em vez de termos uma tabela enorme, dividimos as informações com base em algum critério e as relacionamos com a informação principal através da chave estrangeira. Porém muitas vezes temos uma tabela já existente que contém todas as informações.

Podemos querer separar isso em objetos diferentes no nosso modelo OO, o que não é raro. Assim, mantemos nossas classes organizadas e, dependendo, até mesmo reaproveitamos os dados.

Outros exemplos em que costumamos encontrar tabelas grandes que quebramos em objetos menores são quando mapeamos uma pessoa ou funcionário que pode possuir endereço, documentos, filiação, e muitos outros detalhes que poderíamos querer separar em objetos específicos.

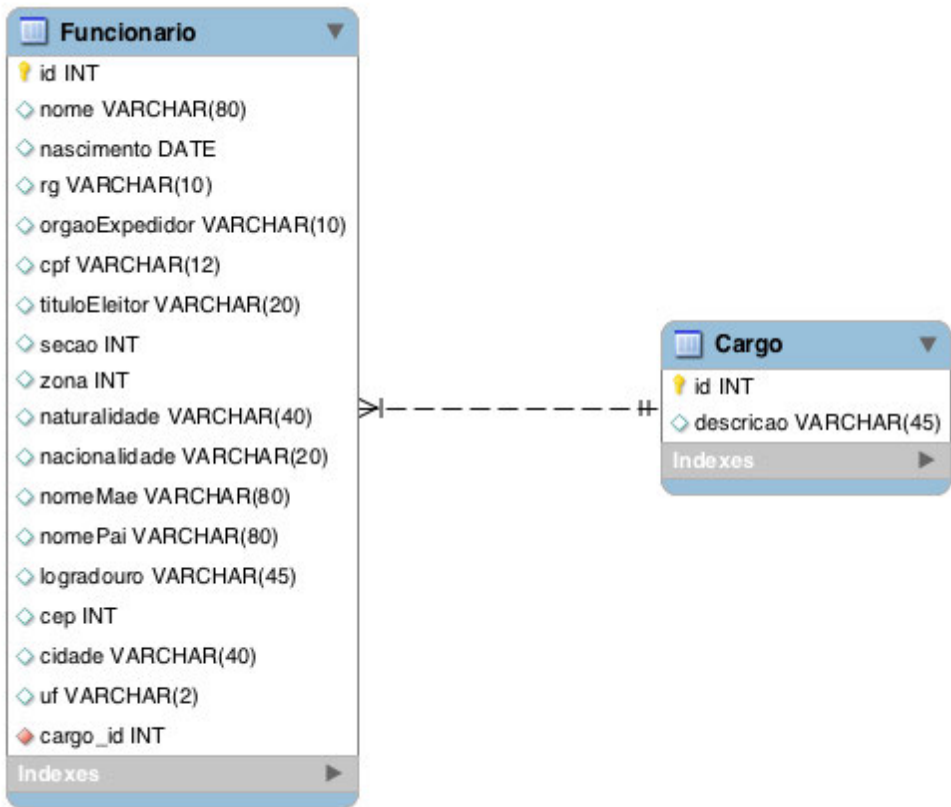


Figura 5.11: Tabela grande com campos de assuntos diversos

Agora imagine que poderíamos ter ainda colunas referentes à data de admissão e demissão, e várias informações adicionais em uma única tabela. Ela poderia ser representada em uma entidade com o seguinte código:

```
@Entity
public class Funcionario {
    @Id @GeneratedValue
    private Integer id;
    private String nome;
    private Date nascimento;
    private String rg;
    private String orgaoExpedidor;
    private String cpf;
```

```
private String tituloEleitor;
private Integer secao;
private Integer zona;
private String naturalidade;
private String nacionalidade;
private String nomeMae;
private String nomePai;
private String logradouro;
private String cep;
private String cidade;
private String uf;
```

```
@ManyToOne
private Cargo cargo;
}
```

```
@Entity
public class Cargo {
    @Id @GeneratedValue
    private Integer id;

    private String descricao;
}
```

A classe `Funcionario` já está grande, mas dependendo do sistema, poderia ser ainda maior. Uma forma de resolver esse problema sem alterar as tabelas no banco é usando `@Embedded` da seguinte maneira:

```
public class Funcionario {
    @Id @GeneratedValue
    private Integer id;

    private String nome;
    private Date nascimento;

    @Embedded
    private Documentacao documentacao;

    @Embedded
    private Endereco endereco;
```



```
@ManyToOne
private Cargo cargo;
}

@Entity
public class Cargo {
    @Id @GeneratedValue
    private Integer id;

    private String descricao;
}
```

Com isso, precisaríamos ter as classes `Documentacao` e `Endereco`, com seus respectivos atributos e indicarmos que elas não são entidades, mas sim, classes que precisam ter seus dados reaproveitados em outras entidades. Fazemos isso por meio da anotação `@Embeddable`.

```
@Embeddable
public class Documentacao {

    private String rg;
    private String orgaoExpedidor;
    private String cpf;
    private String tituloEleitor;
    private Integer secao;
    private Integer zona;
    private String naturalidade;
    private String nacionalidade;
    private String nomeMae;
    private String nomePai;

    // getters e setters se necessário
}
```

```
@Embeddable
public class Endereco {

    private String logradouro;
    private String cep;
    private String cidade;
    private String uf;
```

```
// getters e setters se necessário
}
```

Como pudemos ver, basta anotar a classe com `@Embeddable` para indicar que ela não é na verdade uma entidade, ou seja, que ela não representa uma tabela no banco, e sim que ela estará dentro de uma entidade, representando apenas uma parte de uma tabela.

Com isso, a classe `Funcionario` ficou muito mais limpa, e cada vez que se adicionar um atributo de um tipo que é `@Embeddable`, como a `Documentacao` e o `Endereco`, basta anotá-lo com `@Embedded`.

Agora temos que nos atentar para um detalhe: por mais que saibamos que na tabela está tudo junto, quando vamos consultar usando JPQL temos que respeitar a estrutura das nossas classes. É só recordarmos que, uma vez feito o mapeamento, esquecemos do banco e pensamos apenas com orientação a objetos. Para consultar todos os funcionários de Mato Grosso do Sul, fazemos a seguinte consulta:

```
select f from Funcionario as f where f.endereco.uf = 'MS'
```

Se tentarmos consultar pensando na tabela, teremos um exceção dizendo que não existe a propriedade `uf` na **classe** `Funcionario`.

```
select f from Funcionario as f where f.uf = 'MS'
```

Ou seja, **não estamos falando da tabela `Funcionario`, e sim da entidade `Funcionario`.**

5.13 RELACIONAMENTOS UM PARA MUITOS COM O @ONE-TO-MANY E @MANY-TO-ONE

No modelo da aplicação que estamos desenvolvendo, a `Marca` pode ter diversos `Modelos` associados a ela. Dessa forma, temos um relacionamento um para muitos entre ambas as entidades.

```
public class Marca {

    @Id @GeneratedValue
    private Integer id;
```

```
@OneToMany
private List<Modelo> modelos;

}
```

Para representarmos um relacionamento `OneToMany`, utilizamos a interface `java.util.List`, porém, seria possível utilizar qualquer outra `Collection`.

Já vimos o quanto é importante definirmos se um mapeamento é unidirecional ou bidirecional. Veremos agora como isso é especificado na relação `@OneToMany`.

@OneToMany unidirecional

Pensando na modelagem relacional, obviamente a relação em banco não pode ficar na tabela `Marca`, uma vez que não temos listas em bancos relacionais. A solução, nesse caso, é uma tabela de relacionamento chamada `Modelo_Marca`, que tem as chaves estrangeiras das duas tabelas em questão.

No entanto, talvez fosse mais natural termos na tabela `Modelo` uma chave estrangeira que apontasse para a tabela `Marca`, indicando quem está associado a ela.

@ManyToOne unidirecional

Vamos pensar nesse relacionamento de uma outra maneira. Queremos dizer que todo `Modelo` está de alguma maneira associado a uma `Marca`. No entanto, uma mesma marca pode possuir vários outros modelos, como vimos. Podemos dizer que o `Modelo` possui uma `Marca` da seguinte maneira:

```
public class Modelo {

    @Id @GeneratedValue
    private Integer id;

    @ManyToOne
    private Marca montadora;

}
```

Repare que agora a associação se inverteu, se comparamos com o exemplo que vimos há pouco, em que o relacionamento era com uma lista. Agora temos o relacionamento para um só objeto, a `marca`, por isso usamos o `@ManyToOne`.

Levando essa maneira de pensar para o lado relacional, seria equivalente a dizermos que na tabela `Modelo` temos algo que indique quem é sua `Marca`, o que seria a chave estrangeira. Dessa forma, teríamos na tabela `Modelo` uma coluna `marca_id`, que seria uma chave estrangeira referenciando a tabela `Marca`.

@OneToMany e @ManyToOne bidirecionais

Muitas das vezes, precisamos acessar pelos objetos os dados através das duas pontas do relacionamento. Ou seja, podemos ter um objeto do tipo `Marca` e querermos recuperar os modelos associados a eles; e também podemos ter um objeto do tipo `Modelo` e querermos recuperar a `Marca` associada a ela. Novamente, temos um cenário onde o relacionamento bidirecional poderia nos ajudar.

```
public class Marca {

    @Id @GeneratedValue
    private Integer id;
    ...
    @OneToMany(mappedBy="montadora")
    private List<Modelo> modelos;
    ...
}

public class Modelo {

    @Id @GeneratedValue
    private Integer id;
    ...
    @ManyToOne
    private Marca montadora;
    ...
}
```

Como já sabemos, o lado que não tem o `mappedBy` é o dono da relação, e portanto ele geralmente tem a chave estrangeira em sua tabela. Pronto, agora temos as tabelas estruturadas de uma maneira interessante e também conseguimos acessar as informações através das duas pontas do relacionamento.

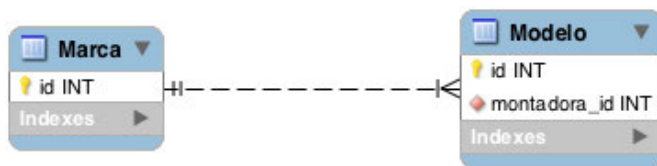


Figura 5.12: Modelo tem chave estrangeira para montadora, do tipo Marca

5.14 A IMPORTÂNCIA DO LAZY LOADING, O CARREGAMENTO PREGUIÇOSO

Assim que temos objetos relacionados, é necessário ter atenção redobrada em como usá-los. Agora que indicamos que a `Marca` tem uma lista contendo os `modelos`, podemos, a partir de qualquer objeto `Marca`, descobrir quem eles são.

O primeiro passo para isso é conseguir uma instância da `Marca`, que pode ser conseguida através do método `find` da `EntityManager`:

```
EntityManager em = JPAUtil.getEntityManager();
```

```
Marca m = em.find(Marca.class, 1);
```

Nesse instante, um SQL é executado para buscar a marca por meio de seu `id`. Repare que o relacionamento com os `modelos` foi ignorado.

Algumas pessoas podem achar estranho, porque como buscamos a `Marca`, ela já deveria estar com os `modelos` carregados. Mas e se essa `Marca` tivesse um milhão de `modelos`? Carregaríamos todos eles na memória? E mais importante ainda, até o momento não precisamos deles. Justamente por esse motivo, a JPA não busca a lista relacionada enquanto não a pedirmos. Esse comportamento é conhecido como *Lazy Loading*, ou carregamento preguiçoso.

Agora vamos invocar o método `getModelos` na `Marca`:

```
EntityManager em = JPAUtil.getEntityManager();
```

```
Marca m = em.find(Marca.class, 1);
List<Modelo> modelos = m.getModelos();
//só busca a lista quando ela é usada
Modelo modelo = modelos.get(0);
```

A partir do momento em que usamos os `modelos` retornados pelo método `getModelos()`, eles são buscados no banco de dados através de um novo `select`. Nesse exemplo, usamos a lista acessando seu primeiro elemento. Apenas recuperar a lista sem utilizá-la não faz buscar no banco.

Na JPA, todos os relacionamentos para uma `Collection`, ou seja, os relacionamentos `ToMany` são *lazy* por padrão. No entanto, essa característica pode ser modificada por um atributo `fetch` da anotação `@OneToMany`, trocando o comportamento de *lazy* para seu inverso, *eager*:

```
@Entity
public class Marca {

    @Id @GeneratedValue
    private Integer id;

    @OneToMany(fetch=FetchType.EAGER)
    private List<Modelo> modelos;
}
```

A partir do momento em que dizemos que o relacionamento é `EAGER`, quando buscarmos uma marca, ela já virá com todos os `modelos` carregados, através de um `select` que possui um `join` para trazer todas as informações de uma só vez.

O `EAGER` deve ser usado com muito cuidado, porque ele pode fazer com que muitos objetos sejam carregados indevidamente, e com que sua aplicação tenha menor performance e escalabilidade.

5.15 @LOB

Usamos `@Lob` para indicar que uma propriedade é um *large object*, geralmente usado para `String` muito grandes ou então para guardarmos tipos binários, como um arquivo anexo dentro do nosso objeto.

Como os dados que serão guardados nesse atributo podem ser muito grandes, como por exemplo uma imagem em alta definição ou um vídeo, é muito comum definir esse atributo como *lazy*, assim ele só será carregado quando efetivamente ser utilizado. Porém, como vimos na seção 5.5, só conseguimos especificar o carregamento sob demanda com a anotação `@Basic` se as anotações estiverem nos *getters*, e não na propriedade.

```
public class Automovel {

    private Long id;
    private byte[] fotografia;

    ...

    @Id @GeneratedValue
    public Long getId(){
        return id;
    }

    @Lob @Basic(fetch=FetchType.LAZY)
    public byte[] getFotografia(){
        return fotografia;
    }

    ...

}
```

5.16 @ELEMENTCOLLECTION

Os automóveis podem possuir `tags` que facilitem a pesquisa por eles. Assim, poderíamos marcá-los com características especiais deles. Como as `tags` são simples textos, como “automático”, “4 portas” e assim por diante, poderíamos mapeá-las como uma `String`, e, já que podemos ter várias `tags`, uma lista de `String`.

```
@Entity
public class Automovel {

    @Id @GeneratedValue
    private Integer id;

    private List<String> tags;

}
```

Nesse caso, teríamos um erro, pois a JPA não sabe como tratar essa lista de `String`. Sempre que tínhamos uma lista, precisávamos indicar que era um relacionamento. Porém, aqui não podemos simplesmente dizer que era um mapeamento `@OneToMany` unidirecional - uma vez que `String` não é uma `@Entity`.

A partir da versão 2 da JPA, entrou o mapeamento `@ElementCollection`, que permite associar uma entidade com uma lista de algum tipo básico, como a `String`.

```
@Entity
public class Automovel {

    @Id @GeneratedValue
    private Integer id;

    @ElementCollection @Column(length=20)
    private List<String> tags;

}
```

Dessa forma, o mapeamento gerará uma tabela de relacionamento chamada `Automovel_Tags` com a seguinte estrutura:

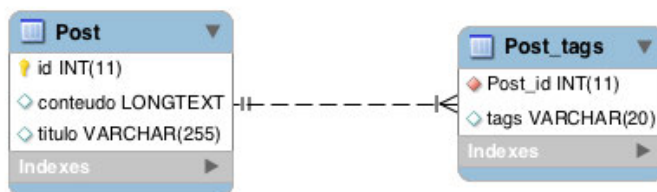


Figura 5.13: Tabela derivada do ‘relacionamento’ de Post com String

5.17 RELACIONAMENTOS MUITOS PARA MUITOS COM O @MANYTOMANY

Os automóveis vendidos na loja podem ter itens opcionais, como ar condicionado, limpador traseiro e assim por diante. Muitos desses opcionais estão presentes em diferentes carros, de modo que precisamos relacionar essas informações, o que caracteriza um relacionamento muitos para muitos.

Justamente para esse tipo de situação, a JPA possui a anotação `@ManyToMany`, que vamos usar para relacionar `Automovel` e `Opcional`:

```
@Entity
public class Automovel {
```



```

@Id @GeneratedValue
private Integer id;

@ManyToMany
private List<Opcional> opcionais;
}

@Entity
public class Opcional {

    @Id @GeneratedValue
    private Integer id;

    private String descricao;

    @ManyToMany(mappedBy="opcionais")
    private List<Automovel> automoveis;
}

```

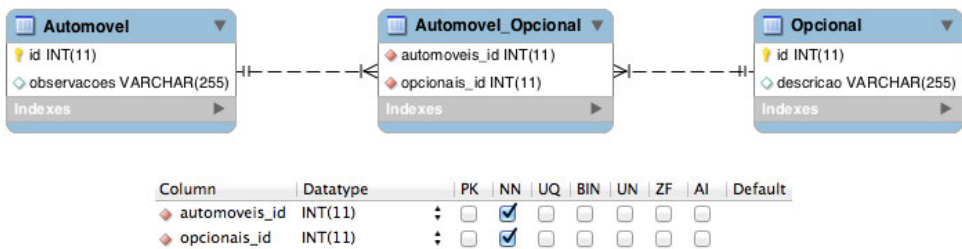


Figura 5.14: Relacionamento muitos para muitos

Como esse é um mapeamento bidirecional, temos que definir o lado que é o dono do relacionamento. Nesse caso, fizemos o dono ser o `Automovel`, dessa forma, ele não tem o `mappedBy`.

Vimos o que ocorre quando temos uma relação bidirecional e não especificamos o `mappedBy`: a JPA entende que são dois relacionamentos independentes. No caso do `@OneToOne` isso podia ser observado, já que eram criadas duas chaves estrangeiras, uma em cada tabela. Aqui no `@ManyToMany`, o resultado caso omitíssemos

o `mappedBy` seria parecido, mas em vez de duas chaves estrangeiras - o que não é possível por serem listas - teríamos duas tabelas de relacionamento, uma chamada `Automovel_Opcional` e outra chamada `Opcional_Automovel`. Pelo padrão de nomes da JPA, no nome da tabela de relacionamento aparece primeiro o nome do dono da relação. Por isso, na figura 5.14, a tabela de relacionamento se chama `Automovel_Opcional`.

O mesmo vale para o `@ManyToMany` em relação à atualização do relacionamento via seu dono. Como já vimos em outros mapeamentos, somente o dono do relacionamento atualiza a chave estrangeira, e nesse caso, a tabela de relacionamento. Sendo assim, de nada adianta adicionar um novo `Automovel` na lista de `automoveis` do objeto do tipo `Opcional` e salvar esse objeto. Precisamos, na verdade, adicionar o `Opcional` na lista `opcionais` do `Automovel` e então salvá-lo. Aí sim teremos a relação persistida no banco.

5.18 CUSTOMIZE AS COLUNAS DE RELACIONAMENTOS COM @JOINCOLUMN E @JOINCOLUMNS

Vimos diversas formas de relacionamento entre as entidades, e sempre que falamos nisso no mundo relacional, falamos de chaves estrangeiras. Até agora nos preocupamos apenas com o funcionamento dos relacionamentos, usando os nomes padrões que a JPA dá essas chaves.

Vimos que pode ser necessário o uso da anotação `@Column` para especificar um nome de coluna diferente quando queremos fugir do padrão de nomenclatura. No entanto, não conseguimos usá-la para mudar o nome das colunas das chaves estrangeiras. Para isso usamos as anotações `@JoinColumn` e `@JoinColumns`.

@JoinColumn no mapeamento de relacionamentos

Precisamos fazer com que a chave estrangeira da `Marca` associada ao `Modelo` tenha um nome diferente do padrão. Queremos que a coluna se chame `montadora_fk`.

```
@Entity
public class Modelo {

    @Id @GeneratedValue
    private Integer id;
```

```
@ManyToOne
@JoinColumn(name="montadora_fk")
private Marca montadora;

}
```

No `@JoinColumn`, a propriedade `name` se refere à coluna da tabela da entidade que contém a anotação. Ou seja, como a anotação está na entidade `Modelo`, a coluna `montadora_fk` está dentro da tabela `Modelo` e é chave estrangeira da coluna `id` dentro da tabela da entidade `Marca`.

Use `@JoinColumns` quando tiver chave estrangeira composta

Usamos a anotação `@JoinColumns` apenas para agrupar mais de uma `@JoinColumn`. Isso é necessário quando temos uma chave estrangeira composta.

Por exemplo, vamos considerar que `Aluno` tem como chave seu nome e o nome de sua mãe, e que cada `Prova` pertence a um `Aluno`. O mapeamento desse modelo ficaria da seguinte forma:

```
@Entity
public class Prova {

    @Id @GeneratedValue
    private Integer id;

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="aluno_nome", referencedColumnName="nome"),
        @JoinColumn(name="aluno_nome_mae",
                    referencedColumnName="nomeMae")
    })
    private Aluno aluno;
}
```

Nesse caso, como a chave é composta, precisamos indicar, por meio do atributo `referencedColumnName`, qual é a chave que estamos renomeando e a qual coluna da outra tabela ela se refere.

5.19 CONFIGURE AS TABELAS AUXILIARES COM @JOINTABLE

Como vimos quando falamos sobre os vários tipos de relacionamentos, alguns deles podem requerer uma tabela auxiliar, como foi o caso do `@ManyToMany` entre os automóveis que continham os itens opcionais. Naquele caso, a tabela auxiliar se chamava `Automovel_Opcional`.

Sempre que quisermos mudar as características de uma tabela auxiliar, podemos usar a anotação `@JoinTable`.

```
@Entity
public class Automovel {

    @Id @GeneratedValue
    private Integer id;

    @ManyToMany(
        @JoinTable(
            name="T_AUTOMOVEIS_OPCIONAIS",
            joinColumns=
                @JoinColumn(name="AUTO_ID"),
            inverseJoinColumns=
                @JoinColumn(name="OPCIONAL_ID")
        )
    private List<Opcional> opcionais;
}

@Entity
public class Opcional {

    @Id @GeneratedValue
    private Integer id;

    private String descricao;

    @ManyToMany(mappedBy="opcionais")
    private List<Automovel> automoveis;
}
```

A anotação `@JoinTable` é colocada na entidade dona do relacionamento e nos permite especificar o nome da tabela de relacionamento e as colunas que a formam. Através da propriedade `joinColumns` informamos as colunas que são chave da tabela dona do relacionamento. Já na propriedade `inverseJoinColumns` especificamos as colunas que são chave da outra tabela, que não é a dona do relacionamento.

5.20 CONCLUSÃO

Vimos várias maneiras de customizar os mapeamentos do Hibernate, de modo que já é possível ter uma flexibilidade maior e se adequar a diferentes modelos de dados.

CAPÍTULO 6

Consultas com a JPQL e os problemas comuns na integração com o JSF

No capítulo 3 fizemos o cadastro, exclusão e listagem dos automóveis, mas ainda não sabemos como recuperar dados de uma maneira mais complexa que um `"select a from Automovel a"`.

Neste capítulo, você vai aprender como funciona a busca de dados através da JPA sem abandonar o modelo orientado a objetos. Veremos também como lidar com a famosa e tão temida `LazyInitializationException`, além de questões fundamentais de infraestrutura para evitá-la.

Analisaremos como um tratamento inadequado da `LazyInitializationException` pode afetar nossos mapeamentos e consequentemente toda a performance da aplicação. Por fim, vamos ver casos em que nosso modelo pode não ser o mais adequado para a criação de um relatório e

mostrar como podemos fazê-lo sem maiores dificuldades.

6.1 FILTRE DADOS COM A JAVA PERSISTENCE QUERY LANGUAGE - JPQL

A *Java Persistence Query Language - JPQL* é uma linguagem de consulta, assim como a SQL, porém orientada a objetos. Isso significa que quando estivermos pesquisando dados, não consideramos nomes de tabelas ou colunas, e sim, entidades e seus atributos. Através dessa linguagem temos acesso a recursos que a SQL não nos oferece, como polimorfismo e até mesmo maneiras mais simples de buscarmos informações por meio de relacionamentos.

Vamos começar com um exemplo bem simples em SQL e em JPQL, e depois analisaremos as diferenças. Considere a entidade `Automovel`, mapeada de uma maneira que tenhamos uma tabela chamada `T_Automoveis` no banco de dados. Entre suas colunas, existe a `ano_modelo`.

```
@Entity
@Table(name="T_Automoveis")
public class Automovel {

    // id e outros atributos

    @Column(name="ano_modelo")
    private Integer anoModelo;

}
```

Em SQL, podemos fazer uma consulta que busque automóveis cujo ano seja maior ou igual a 2010:

```
select * from T_Automoveis where ano_modelo >= 2010
```

No SQL, utilizamos o nome da tabela e o nome da coluna. Contudo, para essa mesma pesquisa na JPQL, precisaremos usar apenas as informações da entidade:

```
select a from Automovel a where a.anoModelo >= 2010
```

Podemos notar uma certa semelhança entre o código em SQL e o que usa a JPQL. Na verdade, isso é desejável, já que SQL é uma linguagem bastante conhecida. Em

contrapartida, podemos anotar algumas diferenças, sendo que a primeira está na identificação da informação que vamos pesquisar: enquanto na SQL referenciamos o nome da tabela, na JPQL referenciamos o nome da entidade. O mesmo pode-se perceber em relação à propriedade `anoModelo`. Em vez de usarmos o nome da coluna, usamos o nome da propriedade da entidade.

Da mesma forma como fizemos uma consulta usando o operador `>=`, com JPQL podemos usar todos os operadores básicos que se usa na SQL: `=`, `>`, `>=`, `<`, `<=`, `<>`, `NOT`, `BETWEEN`, `LIKE`, `IN`, `IS NULL`, `IS EMPTY`, `MEMBER [OF]`, `EXISTS`.

Além dos operadores, a JPQL nos permite usar algumas funções:

6.2 COMO APLICAR FUNÇÕES NAS CONSULTAS

É comum termos que realizar algumas operações que manipulem alguns textos em nossas consultas, como por exemplo, considerar apenas um pedaço de uma `String`, realizar concatenação de textos, converter para maiúsculo ou minúsculo e assim por diante. Para esses casos, a JPQL possui diversas funções que podemos utilizar nas consultas. As principais são:

- `CONCAT(String, String...)`: Recebe uma lista com duas ou mais `Strings` e devolve uma só, concatenando-as;
- `SUBSTRING(String, int start [, int length])`: Recebe um texto, sua posição inicial e, opcionalmente, o comprimento do texto que será devolvido, e retorna só a parte solicitada;
- `TRIM([[LEADING|TRAILING|BOTH] [char] FROM] String)`: Apesar de geralmente usarmos apenas `TRIM('um texto ')`, temos várias combinações:

```
TRIM(' UM TEXTO ') retorna 'UM TEXTO'
TRIM(LEADING FROM ' UM TEXTO ') retorna 'UM TEXTO '
TRIM(TRAILING FROM ' UM TEXTO ') retorna ' UM TEXTO'
TRIM(BOTH FROM ' UM TEXTO ') retorna 'UM TEXTO'
TRIM(LEADING 'A' FROM 'ARARA') retorna 'RARA'
TRIM(TRAILING 'A' FROM 'ARARA') retorna 'ARAR'
TRIM('A' FROM 'ARARA') retorna 'RAR'
```

Caso não informemos um dos três valores (LEADING ou TRAILING ou BOTH) o padrão será BOTH. E caso não informemos que `char` queremos limpar, será seguido o padrão, que é limpar o espaço;

- `LOWER(String)`: Devolve o texto em minúsculo;
- `UPPER(String)`: Devolve o texto em maiúsculo;
- `LENGTH(String)`: Devolve o comprimento do texto;
- `LOCATE(String original, String substring [, int start])`: Devolve a posição em que a `substring` é encontrada na `original`, e opcionalmente há um terceiro parâmetro indicando em qual posição a busca deve começar;

Em outros momentos, podemos utilizar funções que trabalham com valores numéricos:

- `ABS(int)`: Devolve o valor absoluto (sem o sinal) de um determinado número;
- `SQRT(int)`: Devolve a raiz quadrada de um número;
- `MOD(int, int)`: Retorna o resto da divisão do primeiro número pelo segundo. Por exemplo, `MOD(5, 2)` é 1;
- `SIZE(collection)`: Retorna o tamanho de uma coleção: `SIZE(marca.modelos)`;
- `INDEX(obj)`: Retorna a posição de um determinado elemento quando ele estiver em uma lista ordenada:

```
select candidato from Concurso concurso
  join concurso.candidatosAprovados candidato
 where INDEX(candidato) < 5
```

Essa consulta devolve os cinco primeiros candidatos aprovados no concurso.

E por fim, podemos também manipular datas por meio de funções:

- `CURRENT_DATE`: Devolve a data atual do banco de dados;
- `CURRENT_TIME`: Devolve a hora atual do banco de dados;
- `CURRENT_TIMESTAMP`: Devolve a data e hora atual do banco de dados.

6.3 ONDE ESTÃO MEUS JOINS?

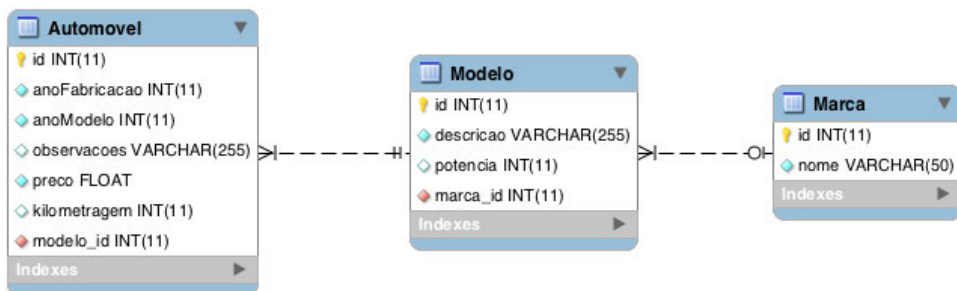


Figura 6.1: DER entre as tabelas Automovel, Modelo e Marca

Estamos com nossos modelos de maneira que reflitam a estrutura de tabelas mostrada na figura 6.1. Quando é preciso buscar automóveis e filtrá-los pelo nome da marca, por exemplo “Ferrari”, temos que escrever um SQL parecido com:

```
select * from Automovel auto
  left outer join Modelo modelo on auto.modelo_id = modelo.id
  left outer join Marca marca on modelo.marca_id = marca.id
where marca.nome = 'Ferrari'
```

Agora levando o pensamento para o mundo orientado a objetos, temos a seguinte estrutura de classes representando o automóvel, marcas e modelos:

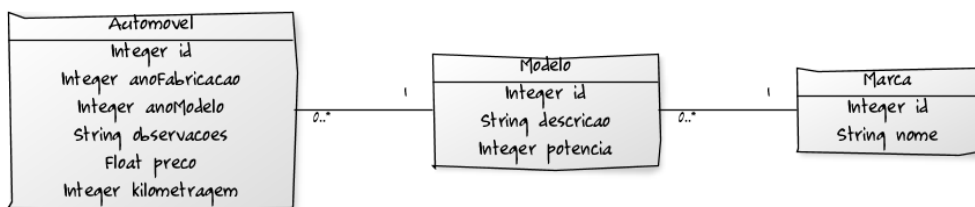


Figura 6.2: Diagrama com as classes Automovel, Modelo e Marca

Com isso, podemos realizar a consulta para buscar automóveis cuja marca seja “Ferrari”, porém através da JPQL.

```
select a from Automovel a where a.modelo.marca.nome = 'Ferrari'
```

Quando temos um caminho “navegável” de objetos simples no nosso modelo não precisamos de qualquer `join`. A JPA abstrai essa necessidade, e a própria implementação que estivermos usando fará a conversão para os *joins* adequados. Por isso, dizemos que a JPQL é uma linguagem de consulta orientada a objetos: se nós, através de nossos objetos, conseguimos acessar outros objetos, então a JPQL também consegue.

Isso significa que os *joins* estão abolidos do nosso dicionário? A resposta é não. Para trabalharmos com coleções de objetos precisamos escrevê-lo.

Ainda usando os mesmos modelos, vamos considerar que precisamos buscar marcas que tenham algum modelo com “911” em sua descrição.

Em SQL teríamos a seguinte consulta:

```
select * from Marca marca
  left outer join Modelo modelo on marca.id = modelo.marca_id
 where modelo.descricao like '%911%'
```

Agora, como estamos pesquisando dentro da lista, precisamos do `join` também na JPQL:

```
select marca from Marca marca
  join marca.modelos modelo
 where modelo.descricao like '%911%'
```

6.4 EXECUTE SUAS CONSULTAS E USE PARÂMETROS

Nos exemplos que vimos até aqui, sempre deixamos os parâmetros na própria consulta por questões didáticas, mas em um código real raramente temos parâmetros fixos. Veremos agora como passá-los para nossa *query*.

Para efetivamente enviar a consulta escrita na linguagem JPQL para JPA executar, nós precisamos de um objeto do tipo `javax.persistence.Query` que conseguimos através do método `createQuery` da `EntityManager`. A partir dele, podemos alimentar os parâmetros que são definidos por `:nome_do_parametro` (dois pontos mais o nome do parâmetro).

```
String jpql =
    "select a from Automovel a where a.modelo.marca.nome = :nomeMarca";
```

```
Query query = entityManager.createQuery(jpql, Automovel.class);
query.setParameter("nomeMarca", "Ferrari");
```

```
List<Automovel> automoveis = query.getResultList();
```

Podemos perceber que chamamos o parâmetro de `nomeMarca`, mas ao utilizá-lo na consulta, devemos colocar o `:` antes, para identificá-lo como um parâmetro. Além disso, indicamos para o método `createQuery`, qual é o tipo de resultado que queremos na nossa consulta, que no caso é um objeto do tipo `Automovel`. Com isso, a `query` sabe que precisará devolver uma lista de objetos desse tipo.

Podemos também passar um objeto de uma entidade como parâmetro, e não apenas textos, números ou datas. Um exemplo disso é o que podemos ver a seguir.

```
String jpql = "select a from Automovel a where a.modelo.marca = :marca";
```

```
Marca marca = ...
```

```
Query query = entityManager.createQuery(jpql, Automovel.class);
```

```
query.setParameter("marca", marca);
```

```
List<Automovel> automoveis = query.getResultList();
```

Quando passamos um objeto de uma entidade para a consulta, a chave é utilizada para fazer a busca, ou seja, a `marca` será filtrada pelo seu `id`.

6.5 UTILIZE FUNÇÕES DE AGREGAÇÃO

Podemos calcular a média dos anos de modelo dos automóveis que temos cadastrados. Para realizar esse cálculo, é comum utilizarmos uma função que faça todo o trabalho para nós, uma vez dada uma propriedade. Essas funções são conhecidas como funções de agregação, e a JPQL as disponibiliza para usarmos:

- `AVG (property)`: Devolve a média de valores numéricos;
- `MAX (property)`: Devolve o valor máximo entre valores comparáveis (números, datas, strings);
- `MIN (property)`: Devolve o valor mínimo entre valores comparáveis (números, datas, strings);
- `SUM (property)`: Devolve a soma de valores numéricos;
- `COUNT (property)`: Devolve a quantidade de elementos.

Então, para calcularmos a média dos `anoModelo`, teríamos a consulta:

```
select AVG(a.anoModelo) from Automovel a
```

Precisaríamos apenas executar essa consulta, através da `javax.persistence.Query`:

```
String jpql = "select AVG(a.anoModelo) from Automovel a";
```

```
Query query = entityManager.createQuery(jpql, Double.class);
```

```
Double media = query.getSingleResult();
```

Repare que, para a agregação que devolve apenas um valor, indicamos no momento de chamar o `createQuery` que o retorno é `Double` - que é o tipo resultante da média, e não mais um `Automovel`, como fizemos antes. Além disso, na hora de recuperar o resultado, não teremos vários itens, e sim um só, que é o próprio valor. Por isso, chamamos o método `getSingleResult` e não o `getResultList`.

6.6 FAÇA SUB-CONSULTAS COM A JPQL

Com a JPQL, podemos criar *subselects* nas nossas consultas. A seguir, um exemplo que pesquisa automóveis acima da média de idade:

```
select a from Automovel a
    where a.anoModelo >
        (select AVG(auto.anoModelo) from Automovel auto)
```

Nesse *subselect* definimos um outro *alias* para `Automovel`, já que as variáveis do `select` de fora estão disponíveis no *subselect*.

Podemos realizar ainda consultas mais elaboradas. Agora precisamos buscar todas as `Marcas` que tenham pelo menos um `Automovel` que custe pelo menos R\$ 1.000.000 (um milhão de reais).

```
select marca from Marca marca where EXISTS (
    select a from Automovel a where
        a.modelo.marca = marca and a.preco >= 1000000
)
```

Repare no uso do `EXISTS` para verificar que a sub-consulta devolve algum resultado para aquela `marca`.

6.7 AGRUPAMENTOS E HAVING

Para terminar nossa passagem pelas opções de pesquisa, vamos analisar um caso de agrupamento com a restrição de grupo.

```
select a.marca, COUNT(a) from Automovel a
      GROUP BY a.marca
      HAVING COUNT(a) > 10
```

Nesse exemplo, agrupamos os automóveis por `marca` e fizemos a consulta devolver a marca e sua quantidade de automóveis. Por fim, restringimos a pesquisa a marcas com pelo menos 10 automóveis.

Mas... como fazemos para recuperar a `marca` e o resultado do `count`, se quando chamamos o método `createQuery`, temos que dizer o tipo da informação que a consulta devolve? **Não temos um tipo que represente a marca e a quantidade.**

6.8 CONSULTAS COMPLEXAS... RESULTADOS COMPLEXOS?

Na maioria dos exemplos que vimos até agora, quase sempre lidamos com o objeto completo:

```
select a from Automovel a
```

Mas em alguns exemplos que vimos, e provavelmente em muitos momentos do nosso dia a dia vamos ver mais, há necessidade de listarmos apenas parte do objeto:

```
select a.descricao, a.preco from Automovel a
```

Nesses casos a construção da pesquisa é simples. Mas como fica o retorno dessa `Query` no código java? É isso que veremos a partir de agora.

```
String jpql =
    "select a.modelo.marca.nome, a.descricao, a.preco from Automovel a";

Query query = entityManager.createQuery(jpql);

List<Object[]> result = query.getResultList();

for(Object[] row : result){
    String nomeMarca = (String) row[0];
    String descricao = (String) row[1];
    Float preco = (Float) row[2];
}
```

Como é possível vermos no exemplo, quando invocamos o método `getResultList` da `Query`, o retorno que temos é do tipo `List`. Mas como não estamos devolvendo um `Automovel` completo, não teremos um `List<Automovel>`. Em vez disso, estamos devolvendo várias propriedades diferentes, de tipos diferentes e de objetos diferentes. Com tantas possibilidades, não há como especificar esse resultado mais do que com um `array` de `Object`, em que cada propriedade da projeção aparece na mesma ordem dentro do `array`.

Existe, porém, uma funcionalidade que nos permite lidar com projeções de forma mais “agradável” do que manipulando um `Object[]`. É a `select new`.

6.9 USE O SELECT NEW E ESQUEÇA OS ARRAYS DE OBJECT

Apesar de em boa parte do sistema lidarmos com consultas que devolvem objetos completos, em algumas situações, não temos como fugir das projeções. Já vimos como tratá-las, mas não queremos manipular `Object[]` dentro do nosso código, já que seria extremamente trabalhoso.

Analisando o problema que temos, verificamos que precisamos de um objeto, que não faz parte do nosso modelo, mas que precisa existir para transferir o resultado da consulta de uma forma mais estruturada. Para isso, vamos criar uma classe que represente essas informações.

```
public class ResumoAutomovel {
    private String marca;
    private String modelo;
    private String descricao;
    private Float preco;

    public ResumoAutomovel(String marca, String descricao,
                           Float preco) {

        //copia os parâmetros para as propriedades do objeto
    }

    //getters e setters se necessário
}
```

Agora que temos esse objeto, podemos pedir para a JPA usá-lo para representar nossa consulta. Para fazermos isso, basta instanciar o objeto no `select`.


```
String jpql = "select new facesmotors.ResumoAutomovel" +  
    "(a.modelo.marca.nome, a.descricao, a.preco) from Automovel a";  
Query query = entityManager.createQuery(jpql);  
List<ResumoAutomovel> result = query.getResultList();
```

A consulta com `select new` irá chamar, para cada linha retornada, o construtor da classe informada exatamente com os mesmos parâmetros que constam na consulta.

6.10 ORGANIZE SUAS CONSULTAS COM NAMED QUERIES

Agora que já vimos como criar nossas consultas e como lidamos com elas no código Java, vamos ver um recurso que nos fornece uma maneira diferente de organizá-las. Esse recurso se chama *named query*.

```
@NamedQuery(name="Automovel.listarTodos",  
    query="select a from Automovel a")  
@Entity  
public class Automovel {  
    // atributos e métodos  
}
```

As *named queries* têm esse nome porque são consultas com um nome único. No exemplo, demos o nome `Automovel.listarTodos` à consulta `select a from Automovel a`, que simplesmente lista todos os automóveis. Nós costumamos colocar a consulta em cima da entidade que é retornada por ela. Nesse caso, como nossa consulta retorna uma lista de `Automovel`, colocamo-la nessa entidade.

Um ponto importante é que o nome da *query* deve ser único, por isso geralmente costumamos usar o nome da entidade no nome da *query* para evitar conflitos. Sem isso, teríamos que colocar nomes como `listarTodosAutomoveis`. Obviamente isso é apenas uma sugestão de convenção.

Em um primeiro momento, a *named query* pode parecer não ajudar muito, até porque a consulta que estamos analisando é muito simples, mas vamos evoluir mais esse exemplo.

```
@NamedQuery(name="Automovel.listarTodos",  
    query="select a from Automovel a where a.ativo = true")  
@Entity  
public class Automovel {
```

```
// outros atributos e métodos

    private boolean ativo;
}
```

Agora fizemos uma pequena modificação na nossa classe e na consulta. Vamos considerar que os automóveis pararam de ser excluídos fisicamente do banco, e agora possuem uma propriedade que marca se logicamente o objeto foi excluído ou não. Se em vez de concentrar a consulta de automóveis em um único lugar nós a tivéssemos espalhado pela aplicação, teríamos que buscar todos os locais onde ela era feita e mudar. Senão, automóveis “fantasmas” começariam a aparecer para os usuários. E olha que aquela *query* inicial parecia tão inofensiva que achamos que nem valeria a pena fazer isso.

Até aí você pode se perguntar: “Mas se eu encapsulasse todas as consultas em métodos de DAO, eu não teria o mesmo resultado?”. A resposta é: “Sim, teria!”. Fica a cargo do desenvolver escolher onde se prefere colocar o código da consulta.

Uma vantagem mais visível é que as *named queries* são checadas enquanto o contexto da JPA se inicia. Então se fizermos mais uma mudança no nosso modelo, para, por exemplo, guardar a data da exclusão em vez de somente um atributo, mas esquecermos de alterar a *query*, teremos um erro:

```
//esquecemos de mudar a consulta
@NamedQuery(name="Automovel.listarTodos",
            query="select a from Automovel a where a.ativo = true")
@Entity
public class Automovel {

    @Temporal
    private Date dataExclusao;
}
```

Erro!

```
ERROR: HHH000177: Error in named query: Automovel.listarTodos
org.hibernate.QueryException: could not resolve property: ativo
of: facesmotors.entities.Automovel
[select a from facesmotors.entities.Automovel a where a.ativo = true]
```

Então agora vamos arrumar nossa *named query*.

```
@NamedQuery(name="Automovel.listarTodos",
            query="select a from Automovel a " +
                "where a.dataExclusao is null")

@Entity
public class Automovel {

    @Temporal
    private Date dataExclusao;
}
```

Uma questão que acaba surgindo é se não existe uma forma *type safe* de fazermos nossas consultas. Ou seja, em vez de depender de textos que não são validados pelo compilador, por que não escrever consultas que deem erro de compilação em casos como o do exemplo que acabamos de ver?

A resposta é que esse suporte existe. Foi adicionado na versão 2.0 da JPA e chama-se `CriteriaBuilder`. Através dessa classe, podemos criar *queries* programaticamente e o compilador gera uma espécie de classe auxiliar para cada entidade. Então fazemos nossas consultas usando essas classes. No entanto, essa API é extremamente complexa e torna muito custosa a escrita de consultas relativamente simples. Por esse motivo, sua adoção no mercado é ainda quase nenhuma.

6.11 EXECUTE AS NAMED QUERIES

Já vimos como executar uma *query* simples dentro de uma classe Java. Agora veremos como praticamente não muda nada com o uso das *named queries*.

```
Query query = em.createNamedQuery("Automovel.listarTodos",
                                Automovel.class);
List<Automovel> automoveis = query.getResultList();
```

A única diferença é que, em vez de usarmos o método `createQuery`, usamos o `createNamedQuery`. Este, em vez de receber a JPQL, recebe somente seu nome, já que ela já está definida na anotação em cima da entidade. Fora isso nada muda, o retorno do método continua sendo uma `Query`. Para passar parâmetros ou executá-la devemos fazer exatamente como fazíamos com a *query* normal.

Nesse exemplo podemos notar outro detalhe: como o nome da *query* é uma `String` é uma prática comum definir uma constante para que não erremos na digitação e também para podermos usar recursos de completação de código ao digitar o nome da *query*.

```

@NamedQuery(name=Automovel.BUSCAR_TODOS,
            query="select a from Automovel a " +
                " where a.dataExclusao is null")

@Entity
public class Automovel {

    // outros atributos e métodos

    public static final String BUSCAR_TODOS = "Automovel.listarTodos";

    private Date dataExclusao;
}

```

E o código que realiza a query:

```

Query query = em.createNamedQuery(Automovel.BUSCAR_TODOS,
                                   Automovel.class);
List<Automovel> automoveis = query.getResultList();

```

Agora definimos uma constante e podemos usar o completador de código da IDE para ver as consultas disponíveis na classe `Automovel`.

6.12 RELACIONAMENTOS LAZY, N+1 QUERY E JOIN FETCH

No capítulo 5 foram mostrados os tipos de relacionamentos entre as entidades. Uma das mais importantes funcionalidades que foi mostrada é o relacionamento *lazy*, que por padrão ocorre em todos os relacionamentos `ToMany`. Ou seja, toda vez que nossa entidade tem um relacionamento que é uma lista, esse relacionamento será *lazy*. Agora se o relacionamento for com um único objeto, ou seja, um relacionamento `ToOne`, ele será *eager* por padrão. Mas vimos que nesses casos é possível alterar o padrão para que também seja *lazy*.

Um relacionamento *lazy* pode evitar o carregamento desnecessário de milhares de registros, como vimos na seção 5.14. Assim, se fôssemos listar vinte `Marcas` que possuem, cada uma, dez `Modelos`, evitamos de carregar duzentos objetos quando não usamos os `Modelos` ao listar as `Marcas`. Mas e se, por acaso, em uma tela mais complexa for necessário listar cada item de uma lista que é *lazy*? Aí o que era uma vantagem passa a dar mais trabalho para otimizar.

Vamos considerar o seguinte modelo:

```
@Entity
public class Marca {

    @Id @GeneratedValue
    private Integer id;

    private String nome;

    @OneToMany(mappedBy="montadora") //LAZY por padrão
    private List<Modelo> modelos;

}
```

```
@Entity
public class Modelo {

    @Id @GeneratedValue
    private Integer id;

    private String descricao;

    @ManyToOne
    private Marca montadora;

}
```

Como não alteramos o padrão, o relacionamento de `Marca` com `Modelo` é *lazy*. Então, caso tenhamos a seguinte listagem de `Marcas`, não teremos nenhum carregamento indesejado de objetos.

```
<h:dataTable value="#{marcaBean.marcas}" var="marca">
    <h:column>
        <h:facet name="header">Nome da Marca</h:facet>
        #{marca.nome}
    </h:column>
</h:dataTable>
```

Consideraremos aqui que `#{marcaBean.marcas}` simplesmente retorna uma lista de `Marca`, similar com o que vimos no capítulo 3.

```
@ManagedBean
public class MarcaBean {
```

```

private List<Marca> marcas;

@PostConstruct
public void carregaMarcas(){
    EntityManager em = JpaUtil.getEntityManager();
    marcas = em.createQuery("select m from Marca m", Marca.class)
                .getResultList();
    em.close();
}

public List<Marca> getMarcas(){
    return marcas;
}
}

```

Agora, estamos usando um recurso novo, que é o `@PostConstruct`. Ele indica um método que será invocado quando o JSF precisar desse `ManagedBean`. Dessa forma, também garantimos que a consulta será realizada somente uma vez para cada requisição.

Mas agora precisaremos de uma modificação que muda todo o cenário: vamos listar os Modelos de cada Marca:

```

<h:dataTable value="#{marcaBean.marcas}" var="marca">
    <h:column>
        <h:facet name="header">Nome da Marca</h:facet>
        #{marca.nome}
    </h:column>
    <h:column>
        <h:facet name="header">Modelos Cadastrados</h:facet>
        <ul>
            <ui:repeat value="#{marca.modelos}" var="modelo">
                <li>#{modelo.descricao}</li>
            </ui:repeat>
        </ul>
    </h:column>
</h:dataTable>

```

A Tag `<ui:repeat>`, vem do Facelets e tem como papel iterar sobre uma Collection, fazendo o papel de um `foreach`.

Executando novamente nossa tela teremos uma `LazyInitializationException`. Essa, na verdade, é uma exceção do Hibernate, que é a implementação de JPA que estamos utilizando. Mas independentemente da implementação escolhida, teremos uma exceção.

O motivo do problema é o fechamento do `EntityManager` dentro do método `carregaMarcas` do `MarcaBean`. Ou seja, no momento em que a consulta precisa ser realizada para buscar os modelos no banco de dados, a conexão não está mais aberta. No entanto, se não fecharmos a `EntityManager`, teríamos uma conexão aberta e não liberada a cada requisição para essa página, ou seja, impraticável.

Nesse ponto, se não soubéssemos das vantagens do uso do *lazy*, poderíamos simplesmente mudar o relacionamento para *eager* e resolver o problema. Mas não podemos esquecer que mexer no mapeamento é algo muito sério, já que vale para o sistema todo, em todas as situações. É só imaginarmos que uma listagem de `Marcas` pode ser usada em um combo. E nesse caso certamente estaríamos carregando diversos `Modelos` sem necessidade.

Para resolver essa questão de ter que liberar recursos, no caso a `EntityManager`, mas ao mesmo tempo não mudar o mapeamento e ainda deixar a aplicação funcionando, foi criado um padrão chamado “Open Entity Manager in View”.

6.13 EVITE A LAZYINITIALIZATIONEXCEPTION COM O OPENENTITYMANAGERINVIEW

A `LazyInitializationException` ocorre quando tentamos acessar propriedades *lazy* que ainda não foram inicializadas, mas a `EntityManager` que trouxe o objeto do banco já foi fechada.

A solução para o problema é fechar a `EntityManager` no momento correto. Se analisarmos, percebemos que o erro acontecia porque, durante a renderização da tela, enquanto a `h:dataTable` iterava sobre os elementos, foi que ocorreu a exceção. Isso quer dizer que podemos (e devemos) sim fechar a `EntityManager`, só temos que tomar o cuidado de fazer isso depois que a processamento de toda a tela terminar.

Existem diversas soluções para esse problema, sendo que uma das mais comuns é por meio da implementação de um `javax.servlet.Filter`.

Como já vimos também, para realizar qualquer escrita no banco via JPA precisamos de transações. Se você estiver desenvolvendo sua aplicação para poder ser

executada em um *Servlet Container*, como por exemplo Jetty ou Tomcat, é necessário tratar as transações diretamente, mas não precisamos deixar esse tratamento dentro de cada método que se salva no banco. Em vez disso, deixamos esse código no *Filter*. Se estivéssemos utilizando um *Application Server* como JBoss ou Glassfish, poderíamos deixar o tratamento de transações, e a criação e fechamento da *EntityManager* a cargo do servidor.

Dessa forma, bastaria que implementássemos um filtro que abrisse a *EntityManager* e a transação, e em seguida delegasse o tratamento para o framework; e no final, quando tudo já estivesse processado, fizesse o `commit` ou o `rollback` da transação e fechasse a *EntityManager*.

```
@WebFilter(urlPatterns="/*")
public class OpenSessionAndTransactionInView implements Filter {

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

        // inicia a transação antes de processar o request
        EntityManager em = JpaUtil.getEntityManager();
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();

            // processa a requisição
            chain.doFilter(request, response);

            // faz commit
            tx.commit();
        } catch (Exception e) { // ou em caso de erro faz o rollback
            if(tx != null && tx.isActive()){
                tx.rollback();
            }
        }
        finally {
            em.close();
        }
    }
}
```



```
    }

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {

    }

    @Override
    public void destroy() {
        JpaUtil.closeEntityManagerFactory();
    }

}
```

Agora com o filtro implementado, se a página for acessada, não haverá mais `LazyInitializationException`. No entanto, quando tivermos muitas marcas e modelos associados, perceberemos que muitas consultas estão sendo feitas no banco de dados. Uma para cada `Marca` que existe no banco. Onde está o problema agora?

6.14 O PROBLEMA DAS N+1 CONSULTAS E COMO RESOLVÊ-LO

Vamos entender melhor o que está acontecendo. As várias consultas ocorrem porque fazemos uma busca por uma lista de `Marca`, que - até o momento em que é executada - dentro do método `carregaMarcas()`, não sabe que os `modelos` serão usados.

Considere que temos 10 marcas cadastradas no banco de dados e que para cada uma delas temos 20 `modelos` associados. A cada iteração da nossa `h:dataTable`, uma instância diferente de `Marca` é usada e cada uma possui uma lista de `Modelo` que ainda não foi carregada do banco. Porém, nessa mesma iteração, na segunda coluna do `dataTable`, nós usamos essa lista. Então a JPA precisa carregá-la do banco, usando uma nova consulta. Esse procedimento é repetido para cada uma das dez instâncias de `Marca`, ou seja, serão dez consultas adicionais.

Tivemos no início uma primeira consulta que buscou as `Marcas` no banco e retornou 10 objetos. Para cada uma das 10 marcas, tivemos uma consulta extra para buscar seus modelos, e por isso dizemos que tivemos N+1 consultas. N é o número de objetos retornados por 1 consulta original. Nesse caso, tivemos 11 consultas. A primeira para buscar as marcas, e uma para buscar os modelos associados a cada

uma das 10 marcas.

Para resolver esse problema, a opção mais “certeira” é tratar o relacionamento como se fosse *eager* somente nesse caso. Como não podemos mexer no mapeamento porque afetaria outras áreas da aplicação, mudamos na nossa consulta. Sabemos que, com a JPA, para buscar as `Marcas` precisamos fazer apenas o seguinte:

```
select m from Marca m
```

Como `Marca` possui um relacionamento com `modelos` esse carregamento é automático. Só precisaríamos fazer um `join` com `Modelo` se quiséssemos fazer alguma busca como essa:

```
select marca from Marca marca
    join marca.modelos modelo
    where modelo.cilindradas > 500
```

Mas agora, precisamos de um `join` somente para carregar a lista `marca.modelos`. Para esse fim temos o `join fetch`. Usando essa instrução fazemos uma consulta como se fosse `fetch=EAGER` no relacionamento. Nossa consulta então ficaria assim:

```
select marca from Marca marca join fetch marca.modelos
```

Agora conseguimos fazer com que um relacionamento que é *lazy* se comporte como se fosse *eager* somente nessa consulta. Mas atenção, o contrário não é possível. Não tem como definirmos um relacionamento como *eager* e fazer uma consulta *lazy* para carregar um simples combo, por exemplo. Por isso, geralmente deixamos o mapeamento *lazy* e, caso precisemos, forçamos o carregamento somente em uma consulta.

6.15 FOI BASTANTE, MAS NÃO ACABOU...

Agora que você já sabe como realizar consultas com a JPA e também já consegue realizar as operações básicas de cadastro, alteração e exclusão, você já deve estar pronto para encarar a grande maioria dos sistemas que envolvam essa tecnologia. Claro, a tecnologia não acaba por aí, ainda vamos ver muito mais. Da mesma forma, ainda temos bastante a ver sobre o JSF. E é justamente daí que vamos continuar.

CAPÍTULO 7

Entenda o JSF e crie aplicações web

Vimos no capítulo 3 como construir telas utilizando JSF e fizemos formulários e listagens. Neste capítulo, teremos como objetivo o entendimento de como o JSF funciona por dentro. Na seção 7.11 veremos como construir telas usando componentes ricos e componentes customizados.

Pelo fato do JSF ser construído usando um paradigma diferente da maioria dos frameworks Web, durante o processo de aprendizagem é comum que tenhamos algumas dificuldades com essa nova maneira de criar aplicações web. Então, primeiro vamos entender como esse mundo novo funciona, e aí sim estaremos prontos para praticar.

7.1 SE PREPARE PARA UM MUNDO DIFERENTE, BASEADO EM COMPONENTES

Quando você começou a programar, seu primeiro paradigma foi a orientação a objetos ou ela lhe foi apresentada depois que você já tinha feito diversos sistemas? Você

iniciou desenvolvendo aplicações Desktop e depois precisou ir para a web? Quem passou por uma dessas mudanças: do procedural para o OO ou do Desktop para a Web vai entender o quão difícil é passar por uma mudança de paradigma.

Introduzi esses exemplos porque assim como existe mudança de paradigma entre a programação Desktop e a Web, existe mudança também ao desenvolver Web com uma ferramenta baseada em ação e uma baseada em componentes. Frameworks como **VRaptor**, **SpringMVC** e **Ruby on Rails** são baseados em ação, já **JSF**, **Tapestry** e **Wicket** são baseados em componentes.

Apesar de termos diferenças no funcionamento do lado servidor, ou seja, desde a submissão das informações do usuário na tela até processarmos a lógica da aplicação, a principal diferença do JSF está na camada de visualização. Pelo fato de ele ter componentes próprios e utilizá-los, assim como fizemos no capítulo 3 com as tags `h:form`, `h:inputText` e outras, o resultado final é um HTML gerado e não um escrito por nós.

Quando o JSF surgiu, a ideia era fazer como no WebForms do .Net, onde teríamos ferramentas de clicar e arrastar que montariam a tela para nós. Apesar de, no início, terem surgido algumas assim, a verdade é que o desenvolvedor Java não possui essa cultura. A consequência é que hoje em dia essa ideia praticamente inexistente. Escrevemos nós mesmos o código das páginas, em arquivos `.xhtml`.

7.2 A WEB STATELESS CONTRA A WEB STATEFUL

Quando desenvolvemos com um framework baseado em ação, como o clássico *Struts*, é comum termos tags que nos auxiliam no desenvolvimento da *view*. Isso passa a falsa impressão de que ele também está ligado à tela assim como o JSF, no entanto, se você quiser ignorar essas tags, fazendo tudo na mão, você pode.

Quando vamos submeter uma tela da nossa aplicação baseada em ação, pouco importa o estado da *view*, como ela foi escrita, tampouco se em vez de ser um JSP ela for um HTML estático. O que importa é que será montada uma requisição que terá como parâmetros os campos contidos no formulário que você escreveu. Frameworks baseado em ações não conhecem a tela, conhecem o `request` e as informações que estão sendo passadas nele.

Em contrapartida, quando trabalhamos com JSF, ele é quem cria a tela. O processo não é nada complicado, é até simples de entender, só que geralmente já saímos fazendo nosso *Hello World* e não paramos para ver como as coisas funcionam. Com JSF, quando chamamos nossa `pagina.xhtml`, em vez de executarmos uma Servlet,

o JSF vai ler esse arquivo e montar em memória uma árvore de componentes, conforme podemos ver na figura 7.1. Depois que essa árvore é montada, ela é passada para um renderizador.

Dessa forma, a árvore de componentes representa apenas a estrutura da página, e o JSF a utilizará para escrever o HTML a cada nó que passar.

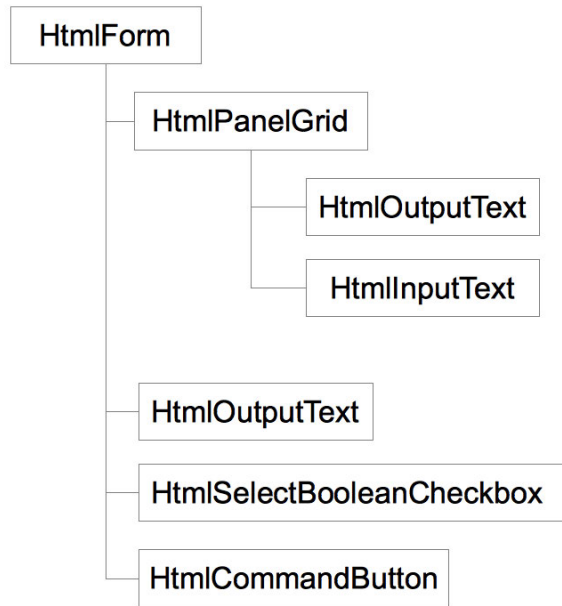


Figura 7.1: Árvore de componentes em memória

Para criar a estrutura da imagem anterior, bastaria um simples código para montar um formulário:

```
<h:form>

    <h:panelGrid columns="2">
        Marca: <h:inputText value="#{marcaBean.marca.nome}"
                        required="true"/>
    </h:panelGrid>

<br/>
```

```
<h:selectBooleanCheckbox value="#{marcaBean.continuarInserindo}"/>
Continuar Inserindo?
<br/>
<h:commandButton value="Salvar" action="#{marcaBean.salvar}"/>

</h:form>
```

Mas o que isso tem a ver com stateful mesmo? A diferença mais significativa do JSF em relação à Servlet (gerada automaticamente baseada em um JSP) não é na geração do HTML, isto é, na ida do servidor para o cliente, e sim quando o cliente submete o formulário apresentado. Enquanto nos frameworks baseados em ações tanto faz como a tela foi criada, o JSF guarda na memória qual foi a árvore usada para gerar aquela tela.

Uma vez que o JSF conseguiu a árvore de componentes, ele vai comparar cada atributo da requisição com os campos que estavam disponíveis para o usuário. Afinal, como foi o JSF quem gerou a tela, ele sabe exatamente quais campos estavam disponíveis e, em caso de seleções como combos, radios e selects, sabe também quais opções estavam lá.

Tendo todas as informações na mão, o JSF valida se os dados enviados são compatíveis com os disponíveis, e caso não sejam, ele indica que a requisição é inválida. É exatamente pelo fato de o JSF conhecer todo o estado da nossa tela, e guardar isso através das requisições, que o consideramos um framework stateful.

7.3 O CICLO DE VIDA DAS REQUISIÇÕES NO JSF

Entender o ciclo de vida do JSF é basicamente entender como ele funciona, e é uma tarefa obrigatória para qualquer pessoa que vai utilizá-lo.

Todo o ciclo de vida do JSF é composto por 6 fases, nas quais diversas tarefas são realizadas. Mas não se assuste, veremos que é mais simples do que pensávamos.

7.4 FASE 1 - CRIAR OU RESTAURAR A ÁRVORE DE COMPONENTES DA TELA (RESTORE VIEW)

Quando uma requisição chega ao JSF, a primeira tarefa realizada por ele é construir ou restaurar a árvore de componentes correspondente ao arquivo XHTML lido. Por exemplo, quando acessamos a página `cadastraAutomoveis.xhtml` da nossa aplicação, o JSF busca por padrão o arquivo de mesmo nome para construir a tela.

Apesar do nome *Restore View*, quando a tela é acessada pela primeira vez a árvore de componentes será criada. Como já foi explicado na seção 7.2, o JSF lê o `xhtml` e cria em memória a árvore de componentes.

Para conseguirmos enxergar o que acontece, considere o código a seguir:

```
<h:form>
    <h:inputText value="#{pessoa.nome}"/>
    <h:inputText value="#{pessoa.idade}">
        <f:validateRequired/>
    </h:inputText>
</h:form>
```

E a classe `Pessoa`:

```
public class Pessoa {
    private String nome;
    private Integer idade;
    //getters e setters
}
```

O JSF irá criar uma instância de `javax.faces.component.html.HtmlForm`, uma classe dele mesmo, que representa através de código Java um `h:form`, e colocará abaixo dela duas instâncias de `javax.faces.component.html.HtmlInputText`, que representam o `h:inputText`. No segundo `HtmlInputText` será chamado o método `addValidator` passando como parâmetro uma instância de `javax.faces.validator.RequiredValidator`.

Considerando que usuário esteja requisitando a página pela primeira vez, após a criação da árvore de componentes, o *request* vai direto para a fase 6, na qual será renderizada a resposta para o usuário.

Caso o usuário já esteja utilizando a aplicação, a árvore, em vez de ser criada, será recuperada da sessão do usuário. O JSF faz um cache de árvores, guardando na sessão do usuário o estado das telas acessadas por último. Outra alternativa é solicitar que o JSF armazene o estado no cliente. Nesse caso, a árvore será serializada e enviada em um campo oculto dentro do HTML gerado, e quando o usuário submeter novamente ela será reconstruída a partir do valor desse campo oculto.

Essa configuração é feita no `web.xml` como podemos ver a seguir.

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
```

```
<param-value>client</param-value> <!-- valor padrão é server -->
</context-param>
```

Depois que a árvore é restaurada, seja buscando na sessão ou via deserialização, a requisição do usuário segue para a fase 2.

7.5 FASE 2 - APLICAR VALORES DA REQUISIÇÃO NA ÁRVORE DE COMPONENTES (APPLY REQUEST VALUES)

Aqui o JSF irá buscar os valores informados pelo usuário e colocá-los nos seus respectivos componentes. Considerando que o usuário tenha entrado com o nome “João” e a idade “28”, o pseudocódigo a seguir seria executado nessa fase.

```
HtmlInputText nome = ... // JSF já conhece o componente
nome.setSubmittedValue("Fulano");
```

```
HtmlInputText idade = ... // JSF já conhece o componente
idade.setSubmittedValue("28");
```

O importante aqui é entendermos que o JSF chama o método `setSubmittedValue` passando o valor que o usuário digitou, sem se preocupar se ele é válido ou não. Por mais que o usuário tivesse informado “ABC” na idade, esse valor seria passado para esse método e o componente teria esse valor na sua propriedade `submittedValue`.

7.6 FASE 3 - CONVERTER E VALIDAR (VALIDATE)

Essa sem dúvida é a fase mais conhecida, já que, se você não entender como funciona o JSF, certamente mais cedo ou mais tarde terá algum erro de conversão ou de validação. Essa fase é dividida em partes, como o próprio título da seção informa: converte e valida.

Em nosso exemplo, a conversão será transformar a `String idade = "28"` em `Integer idade = 28`. Caso o usuário informasse algo como `"28a"`, teríamos um erro de conversão.

O JSF já possui conversores prontos para os tipos básicos como `Integer`, `Long`, `java.util.Date`, entre outros. Mas e quando precisarmos converter um código de funcionário para um objeto da nossa aplicação do tipo `Funcionario`? Precisaremos registrar no JSF um conversor customizado, como veremos na seção 7.36, mas o processo de conversão do JSF é o mesmo.

O JSF descobre qual a classe do objeto que está vinculada com o `value` do componente, que no caso da idade é do tipo `Integer`, e então busca no seu contexto um conversor para esse tipo.

Depois de convertido, o valor informado pelo usuário será validado. No caso do componente que referencia idade temos o `RequiredValidator` agindo em cima de um `Integer`, e não em cima de uma `String`. Da mesma forma, se fosse um validador de intervalo, `LongRangeValidator`, o mesmo validaria um número, e não uma `String`. Só consideramos erro de validação se a conversão aconteceu com sucesso, senão teremos erro de conversão.

7.7 FASE 4 - ATUALIZAR O MODELO (UPDATE MODEL)

Agora que já temos informações válidas, é hora de colocar esses valores dentro do nosso modelo. No nosso exemplo, o modelo é o objeto do tipo `Pessoa`, e a execução da fase 4 pode ser entendida pelo pseudocódigo a seguir.

```
HtmlInputText inputNome = ... // já convertido e validado
HtmlInputText inputIdade = ... // já convertido e validado

Pessoa pessoa = jsfAvaliaExpressionLanguage("#{pessoa}");

pessoa.setNome(inputNome.getValue());
pessoa.setIdade(inputIdade.getValue());
```

Como as ELs ligadas aos componentes eram `#{pessoa.nome}` e `#{pessoa.idade}`, podemos perceber um padrão. Como nessa fase o JSF está inserindo valores no modelo, e não recuperando, ele recupera o objeto correspondente ao penúltimo nível da EL e chama o método `set` do último nível. Se tivéssemos o objeto `Funcionario` ligado à EL `#{cadastroBean.cadastro.funcionario}`, o JSF recuperaria o objeto `cadastro` que é propriedade do `#{cadastroBean}` e chamaria o método `setFuncionario` do objeto `cadastro`.

Depois de terminada essa fase, temos nosso modelo com os valores corretos. É o que poderíamos chamar de “mundo perfeito”, onde o usuário entra com `String` e o JSF nos entrega objetos Java com seus tipos corretos e valores validados.

7.8 FASE 5 - INVOCAR AÇÃO DA APLICAÇÃO (INVOKE APPLICATION)

Nessa fase acontece a lógica da aplicação. Aqui dentro não precisamos e não devemos ficar buscando objetos baseados em algum *id*, uma vez que isso é responsabilidade dos conversores. Igualmente, não devemos ficar validando se uma senha respeita o tamanho mínimo de caracteres e possui letras e números, já que isso deve ser feito pelo validador. Se você já viu um projeto em JSF no qual a ação inicia com `ifs`, a probabilidade dos recursos do JSF estarem sendo subutilizados é muito grande.

Apesar da possibilidade de construirmos validadores customizados, costumamos deixar as validações **de negócio** dentro da fase 5, mas somente essas deviam estar aqui. Vamos considerar, por exemplo, um sistema de universidade, em que para efetuar uma matrícula precisamos verificar primeiro se não existe mensalidade em aberto, ou se já não existe um acordo sendo cumprido em relação às possíveis mensalidades em aberto. Nesse caso poderíamos ter um trecho de código como o seguinte.

```
public void efetuaMatricula(PropostaMatricula proposta){
    try{
        servicoFinanceiro.validaSituacaoAluno(proposta.getAluno());

        // efetuaMatricula
    }
    catch(MensalidadesEmAbertoException e){
        // exibe mensagem para o usuário e aborta o a matrícula
    }
}
```

E agora que realizou a ação, terminou, certo? Errado! Ainda tem um último detalhe importantíssimo por acontecer.

7.9 FASE 6 - RENDERIZAR A RESPOSTA (RENDER RESPONSE)

Chegamos à última fase do ciclo de vida do JSF. Podemos chegar até ela depois que tudo ocorreu bem nas outras fases, como por exemplo depois que salvamos um registro com sucesso, ou depois de um erro em alguma fase intermediária.

Se ocorrer um erro de conversão ou validação, as fases 4 e 5 são puladas e vamos direto para a fase 6 para mostrar novamente o formulário para o usuário para que ele

possa corrigir, já com as devidas mensagens de erro. Ou ainda podemos chegar aqui logo depois da fase 1, quando o usuário pede a página pela primeira vez. Nesse caso o JSF monta a árvore e já manda a fase 6, já que, na primeira requisição à aplicação, não haverá formulário sendo submetido e nem ação para ser invocada.

Nessa fase acontece a geração do HTML a partir da árvore de componentes do JSF. Vamos considerar a renderização do mesmo trecho de código que vimos na fase 1:

```
<h:form>
    <h:inputText value="#{pessoa.nome}"/>
    <h:inputText value="#{pessoa.idade}"/>
    <f:validateRequired/>
</h:inputText>
</h:form>
```

E a classe `Pessoa`:

```
public class Pessoa {
    private String nome;
    private Integer idade;
    //getters e setters
}
```

A cada tag que o JSF passa, ele chama o renderizador correspondente. Existem renderizadores para cada componente HTML básico, mas bibliotecas de componentes como Richfaces e Primefaces podem substituí-los. Uma utilidade da substituição dos renderizadores é que essas bibliotecas podem incluir, por exemplo, regras de CSS próprias delas, sem forçar o desenvolvedor a colocá-las em todos os atributos. Assim como as bibliotecas, nós podemos também gerar nossa própria HTML. Seja através da criação de novos componentes ou implementando novos renderizadores.

Para cada nó da árvore, o JSF manda renderizar o início da tag e depois seus componentes filhos. Terminados os filhos, ou não existindo, é feito o fechamento da tag. No caso dos componentes que possuem *Expression Language* ligada a eles, esta é avaliada para obter o objeto Java correspondente.

Como está sendo gerada uma saída, ao avaliar a expressão `#{pessoa.nome}`, uma instância de `pessoa` será obtida e é nela que o método `getNome` será chamado.

Assim que o JSF tem o objeto Java, ele recupera novamente o conversor para o tipo adequado da informação, para que gere a `String` correspondente ao objeto,

que será exibida na tela. Caso não exista um conversor, o JSF apenas chama o método `toString` do objeto.

Uma visão geral do ciclo de vida do JSF pode ser vista na figura 7.2.

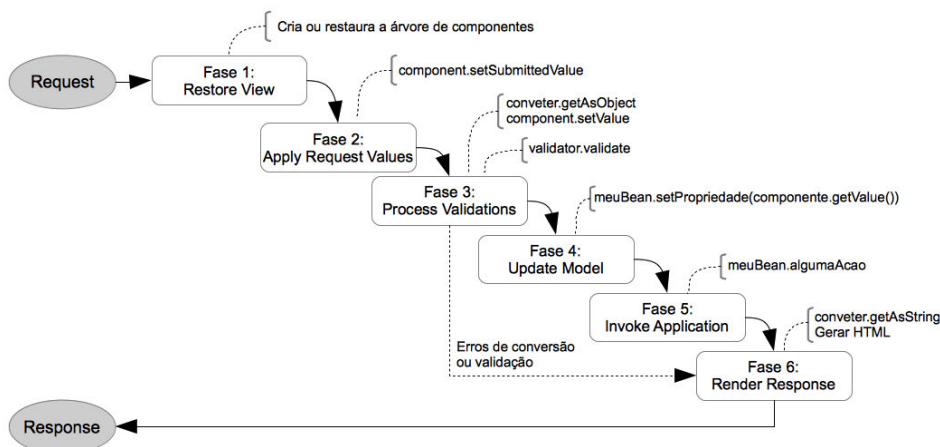


Figura 7.2: Ciclo de vida do JSF

7.10 AJA SOBRE AS FASES DO JSF COM OS PHASELISTENERS

Vimos as seis fases do JSF e a responsabilidade de cada uma. Agora veremos como solicitar ao JSF que nos notifique a cada mudança de fase, através de um `PhaseListener`.

Utilizamos um `PhaseListener` praticamente nas mesmas situações em que usaríamos um `javax.servlet.Filter`. Mas em vez de envolver a requisição inteira, seremos notificados antes e depois de cada fase, o que faz com que tenhamos um acesso mais granular. Com isso, podemos ir a um ponto específico da requisição, caso nos interesse.

Um exemplo da utilização do `PhaseListener` é para autenticação:

```

public class AutenticacaoPhaseListener implements PhaseListener {

    private static final String RESTRICTION_PATTERN = "~/restrito/.*";

    public PhaseId getPhaseId() {

```

```
        return PhaseId.RESTORE_VIEW;
    }

    public void beforePhase(PhaseEvent event) {
    }

    public void afterPhase(PhaseEvent event) {
        FacesContext context = event.getFacesContext();
        String viewId = context.getViewRoot().getViewId();
        boolean urlProtegida = Pattern.
            matches(RESTRICTION_PATTERN, viewId);

        Object usuario = context.getExternalContext()
            .getSessionMap().get("usuarioLogado");

        if(urlProtegida && usuario == null){
            NavigationHandler navigator = context
                .getApplication()
                .getNavigationHandler();

            navigator.handleNavigation(context, null, "login");
        }
    }
}
```

Esse é apenas um exemplo de código de autenticação, mas o mais importante é analisarmos o funcionamento do `PhaseListener`. O primeiro método, `getPhaseId()` devolve qual fase do JSF nosso *listener* irá escutar. Ou podemos devolver `PhaseId.ANY_PHASE` para indicar que queremos ser notificados antes e depois de todas as fases.

Diferente de um `javax.servlet.Filter`, não temos um único método que envolve toda a fase. Em vez disso, temos um método que é chamado antes do processamento da fase, `beforePhase`, e outro que é chamado depois, `afterPhase`. No nosso exemplo, utilizamos somente o segundo método, porque antes da fase *Restore View* ainda não é possível recuperar qual tela está sendo requisitada.

Para que o código do `PhaseListener` funcione, precisamos de uma regra de navegação global que saiba ir para a tela de login, a partir do *outcome* `login`. Também devemos registrá-lo no `faces-config.xml`, que acabaria ficando com um conteúdo parecido com o seguinte:

```

<navigation-rule>
    <navigation-case>
        <from-outcome>login</from-outcome>
        <to-view-id>/login.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
<lifecycle>
    <phase-listener>
        facesmotors.AutenticacaoPhaseListener
    </phase-listener>
</lifecycle>

```

É muito importante que a tela de login não esteja contida na expressão regular que restringe as páginas da aplicação, senão teremos um *loop* de redirecionamentos.

7.11 CONHEÇA OS COMPONENTES DO JSF

Uma questão interessante em relação a todos os componentes do JSF é que, apesar de gerarem HTML, temos duas propriedades presentes em praticamente todos os componentes, o que nos permite customizar o visual da tela a partir de folha de estilos. Essas propriedades são `style` e `styleClass`. A primeira aceita conteúdo CSS, que será colocado dentro da propriedade `style` do HTML gerado. A segunda renderiza a propriedade `class` no HTML gerado e acaba sendo muitas vezes mais usada, porque é uma boa prática colocar classes de CSS no HTML, e não o estilo propriamente dito.

```
<h:inputText style="padding: 10px;" styleClass="valor_numerico" />
```

São diversos componentes disponíveis no JSF, alguns mais simples, outros mais complexos. Vamos aprendê-los.

7.12 H:FORM

```

<h:form>

</h:form>

```

A tag `h:form` geralmente é usada sem nenhuma propriedade, já que no JSF não precisamos especificar nenhuma `action`, como é normal fazer em frameworks

baseado em ações. Afinal, a informação do que será feito quando a página for submetida está no próprio componente que fará a submissão das informações, como o `h:commandButton`.

Quando especificamos alguma propriedade, essa geralmente é o `id`. Precisamos disso quando vamos manipular diretamente os elementos, como por exemplo para trabalharmos com AJAX.

Outra característica importante com relação a `id` de componentes é quando temos outros componentes dentro do formulário.

```
<h:form id="formulario">
  Idade: <h:inputText id="campo_idade"/>
</h:form>
```

Com esse código, o HTML gerado terá o formulário com `id formulario`, porém o `inputText` terá `id formulario:campo_idade`. Repare que o `id` do formulário foi adicionado ao início do componente filho, o `inputText`. Esse comportamento é o que chamamos de *naming container* e podemos desabilitá-lo por meio do atributo `prependId`, deixando-o como `false`.

```
<h:form id="formulario" prependId="false">
  Idade: <h:inputText id="campo_idade"/>
</h:form>
```

Pronto, dessa vez, no HTML gerado, o `inputText` terá um `id` somente com o valor `campo_idade`. O resultado do código no navegador será o seguinte:

Idade:

Figura 7.3: Resultado da tag `h:inputText`

7.13 H:INPUTTEXT E H:INPUTTEXTAREA

```
<h:inputText value="#{managedBean.objeto.descricao}"/>
```

```
Descrição: <h:inputTextarea value="#{managedBean.objeto.descricao}"/>
```

Esses dois componentes possuem uma apresentação visual diferente mas são praticamente iguais. A diferença entre eles é que o `h:inputTextarea` possui as

propriedades `rows`, que especifica a quantidade de linhas, e `cols`, que especifica a quantidade de colunas que o campo irá apresentar. Já o `h:inputText` tem as propriedades `size`, que serve para especificar o tamanho do campo baseado na quantidade de caracteres, algo parecido com o `cols` do `h:inputTextarea`; e a propriedade `maxlength`, que especifica a quantidade máxima de caracteres que o campo aceita.

Depois que vimos as diferenças, vamos tratar ambos apenas como “input”. No JSF, por mais que tenhamos diversas forma de apresentar os elementos na tela, quase todos os componentes são inputs. Um combo, um select, um checkbox ou radiobutton são formas diferentes de apresentar inputs ao usuário.

A principal propriedade de qualquer input é a `value`. A ela, nós ligamos uma *Expression Language* que, depois de conversão e validação, permitirá ao JSF colocar o valor que o usuário digitou no campo, diretamente dentro da propriedade especificada.



Figura 7.4: Resultado do tag `h:inputTextarea`

7.14 H:INPUTSECRET

Senha: `<h:inputSecret value="#{managedBean.usuario.senha}"/>`

O `h:inputSecret` gera o HTML equivalente a um `input type="password"`, mostrando o conteúdo digitado de uma maneira que não se consiga visualizar o conteúdo, normalmente mostrando * (asteriscos) no lugar do texto.



Figura 7.5: Resultado da tag `h:inputSecret`

7.15 H:INPUTHIDDEN

```
<h:inputHidden value="#{managedBean.objeto.id}"/>
```

Esse componente se comporta quase que exatamente como um `h:inputText`, com a diferença de ficar oculto do usuário. Dessa forma, não existe um resultado visual para ilustrarmos.

7.16 H:SELECTONEMENU, F:SELECTITEM E F:SELECTITEMS

Marca:

```
<h:selectOneMenu value="#{modeloBean.modelo.marca}">
    <f:selectItem itemLabel="-- Selecione --" noSelectionOption="true"/>
    <f:selectItems value="#{marcaBean.marcas}" var="marca"
        itemValue="#{marca}" itemLabel="#{marca.nome}"/>
</h:selectOneMenu>
```

Apesar de parecer mais complexo, um `h:selectOneMenu` nada mais é do que um input que já traz para o usuário opções pré-definidas. Por exemplo, o usuário precisa escolher qual é a `marca` do carro que ele quer pesquisar. Em vez de deixarmos que ele digite essa informação e possivelmente cometa erro de digitação, podemos listar as opções disponíveis para ele, assim é possível escolher.

Essas opções podem ser uma combinação de itens estáticos, como o do `f:selectItem`, com dinâmicos, como o do `f:selectItems`. Geralmente o primeiro é usado apenas para uma mensagem instruindo o usuário a selecionar um valor. Sem isso, a primeira opção do combo viria selecionada, e não teríamos como saber se o usuário escolheu aquela opção ou simplesmente se esqueceu de selecionar.

A listagem dinâmica geralmente vem de algum *Managed Bean* que possua essa listagem. No exemplo, `marcaBean` já se possui naturalmente essa listagem, então buscamos de lá. O importante a notarmos é a estrutura do `f:selectItems`, que uma vez entendida ficará simples e bem prático usar qualquer tipo de select.

O `f:selectItems` possui uma propriedade `value` que liga o componente com a lista de valores que ele vai listar. Com isso, definimos uma variável para podermos acessar de dentro das propriedades `itemValue` e `itemLabel`.

O `itemValue` do `f:selectItems` especifica qual valor será atribuído ao `value` do `h:selectOneMenu`. Repare que no `value` do combo temos a `Marca` do `Modelo` que está sendo editado, e o `itemValue` especifica que ao selecionar um item do combo o valor atribuído será uma `Marca`. Já a propriedade `itemLabel` serve para especificarmos o que aparecerá para o usuário.



Figura 7.6: Resultado da tag h:selectOneMenu

7.17 H:SELECTONERADIO

Marca:

```
<h:selectOneRadio value="#{modeloBean.modelo.marca}">
  <f:selectItems value="#{marcaBean.marcas}" var="marca"
    itemValue="#{marca}" itemLabel="#{marca.nome}" />
</h:selectOneRadio>
```

Um `h:selectOneRadio` é praticamente igual a um `h:selectOneMenu`. Primeiro como podemos perceber, os dois iniciam seu nome com `selectOne`, ou seja, nos dois casos o usuário poderá selecionar apenas um elemento, porém a exibição será por *radio button*.

Outra diferença está em como cada componente é apresentado para o usuário. No caso do `h:selectOneRadio`, podemos especificar um pouco mais como será essa apresentação através da propriedade `layout`, que por padrão tem o valor `lineDirection`. Com esse valor as opções serão apresentadas uma na frente da outra. Ou podemos especificar `pageDirection`, e nesse caso as opções serão apresentadas uma abaixo da outra.



Figura 7.7: Resultado da tag h:selectOneRadio com layout lineDirection



Figura 7.8: Resultado da tag `h:selectOneRadio` com `layout pageDirection`

7.18 H:SELECTONELISTBOX

Marca:

```
<h:selectOneListbox value="#{modeloBean.modelo.marca}" size="5">
    <f:selectItems value="#{marcaBean.marcas}" var="marca"
        itemValue="#{marca}" itemLabel="#{marca.nome}" />
</h:selectOneListbox>
```

Outro componente da família dos `selectOne`: a diferença desse componente para um combo (`h:selectOneMenu`) é que o combo mostra apenas uma linha, e precisamos clicar para abrir as opções. Já o `h:selectOneListbox` já traz as opções visíveis, bastando o usuário selecionar o que desejar. Caso queiramos limitar o tamanho da lista, que por padrão mostra todos os elementos, podemos utilizar a propriedade `size`. Então, caso tenhamos mais elementos do que o que pode ser apresentado, será exibida uma barra de rolagem no componente.



Figura 7.9: Resultado da tag `h:selectOneListbox`

7.19 H:SELECTMANYMENU E H:SELECTMANYLISTBOX

```
<h:selectManyMenu value="#{automovelBean.automovel.acessorios}">
    <f:selectItems value="#{acessorioBean.acessorios}" var="acessorio"
        itemValue="#{acessorio}"
        itemLabel="#{acessorio.descricao}" />
</h:selectManyMenu>
```

Marca:

```
<h:selectManyListbox value="#{automovelBean.automovel.acessorios}"
    size="5">
    <f:selectItems value="#{acessorioBean.acessorios}" var="acessorio"
        itemValue="#{acessorio}"
        itemLabel="#{acessorio.descricao}" />
</h:selectManyListbox>
```

Apesar de estarmos entrando na família dos `selectMany`, a dinâmica dos componentes continua a mesma. Agora o `value` do componente guarda uma lista e não mais um único objeto. O uso do `f:selectItems` no entanto, continua inalterado. Assim, o `itemValue` continua representando um objeto selecionado de tipo compatível com o `value` do `select` em que ele está inserido, e agora ele é de um tipo compatível com os elementos do `value`.

Em nosso exemplo, o `value` do `select` é uma lista de acessórios e cada elemento do `f:selectItems` é um acessório que pode ser inserido nessa lista.

Com relação aos componentes `h:selectManyMenu` e `h:selectManyListbox`, a diferença entre eles é apenas a presença da propriedade `size` no componente `h:selectManyListbox`, enquanto o componente `h:selectManyMenu` tem tamanho fixado em uma única linha.



Figura 7.10: Resultado da tag `h:selectManyListbox`

7.20 H:SELECTMANYCHECKBOX

Acessórios:

```
<h:selectManyCheckbox value="#{automovelBean.automovel.acessorios}">
    <f:selectItems value="#{acessorioBean.acessorios}" var="acessorio"
        itemValue="#{acessorio}"
        itemLabel="#{acessorio.descricao}" />
</h:selectManyCheckbox>
```

É a versão “many” do componente `h:selectOneRadio`, mostrando *checkboxes* para a seleção do usuário. Ele também possui uma propriedade chamada `layout` que por padrão tem valor `lineDirection` e mostra os itens selecionáveis um ao lado do outro. Podemos também mudar o valor da propriedade para `pageDirection` e mostrá-los um abaixo do outro. Excetuando-se isso, o comportamento do componente é o mesmo dos componentes `selectMany` vistos anteriormente.

Acessórios:
☐ Teto solar ☐ Roda de liga leve

Figura 7.11: Resultado da tag `h:selectManyCheckbox` com layout `lineDirection`

Acessórios:
☐ Teto solar
☐ Roda de liga leve

Figura 7.12: Resultado da tag `h:selectManyCheckbox` com layout `pageDirection`

7.21 H:SELECTBOOLEANCHECKBOX

Usuário inativo:

```
<h:selectBooleanCheckbox value="#{usuarioBean.usuario.inativo}"/>
```

Não podemos confundir esse componente com o `h:selectManyCheckbox`. Apesar de ambos serem *checkboxes*, são completamente diferentes. Como seu próprio nome sugere, usamos o `h:selectBooleanCheckbox` ligado com propriedades booleanas. Como podemos ver no exemplo, usamos esse componente para alterar propriedades dos nossos objetos de modelo, como mudar o status de um usuário enquanto o editamos.

Usuário inativo: ☒

Figura 7.13: Resultado da tag h:selectBooleanCheckbox

Podemos também usar o componente para, além de alterar a propriedade de um objeto, ativar ou não a exibição de outros componentes.

```
<h:selectBooleanCheckbox value="#{compraBean.compra.temDesconto}">
    <f:ajax render="desconto"/>
</h:selectBooleanCheckbox> Compra com desconto
<h:panelGroup id="desconto">
    <h:panelGroup rendered="#{compraBean.compra.temDesconto}">
        % desconto <h:inputText value="#{compraBean.compra.desconto}"/>
    </h:panelGroup>
</h:panelGroup>
```

7.22 NOVIDADE DO JSF 2.2: H:INPUTFILE

```
<h:form enctype="multipart/form-data">
    Imagem: <h:inputFile value="#{automovelBean.uploadedFile}"/>

    <h:commandButton value="Salvar" action="#{automovelBean.salvar}"/>
</h:form>
```

E o código Java para receber o *upload* do arquivo.

```
public void class AutomovelBean {

    private javax.servlet.http.Part uploadedFile;
    private Automovel automovel;

    // getters e setters

    public void salvar(){
        try {
            InputStream is = uploadedFile.getInputStream();
            byte[] bytes = IOUtils.toByteArray(is);
            automovel.setImagem(bytes);
            em.persist(automovel);
        }
    }
}
```

```

    } catch (IOException e) {
        // tratar exceção
    }
}
}

```

Esse é um componente novo do JSF 2.2. Até a versão anterior, para realizar *upload* de arquivo era necessário utilizar componente de bibliotecas como o Primefaces.

Como podemos ver no exemplo, ligamos o `value` do componente `h:inputFile` diretamente no objeto do tipo `javax.servlet.http.Part`, novo na API de Servlet 3.0. A partir desse objeto conseguimos acessar o `InputStream` do arquivo enviado pelo usuário.

Recuperamos então o `byte[]` a partir do `InputStream` e o guardamos na propriedade `imagem` do `Automovel`, que é armazenada no banco de dados como um `@Lob` da JPA.

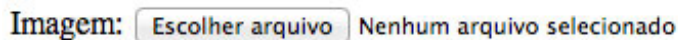


Figura 7.14: Resultado da tag `h:inputFile`

7.23 H:PANELGRID

```

<h:panelGrid columns="2">
    Marca:      <h:outputText value="#{automovel.modelo.marca}"/>
    Modelo:     <h:outputText value="#{automovel.modelo}"/>
    Ano Fabr.:  <h:outputText value="#{automovel.anoFabricacao}"/>
    Ano Modelo: <h:outputText value="#{automovel.anoModelo}"/>
    Km:        <h:outputText value="#{automovel.kilometragem}"/>
</h:panelGrid>

```

Um `h:panelGrid` renderiza uma tabela cujo número de colunas é definido pela propriedade `columns`. No exemplo, informamos que queríamos duas colunas, então a cada dois componentes o `h:panelGrid` cria uma nova linha. A vantagem em usar o componente em vez de fazer uma tabela é que podemos parametrizar o número de colunas, podendo por exemplo em vez de duas, mostrar quatro colunas sem mudanças no código.

Assim como qualquer componente `container`, ou seja, que envolve outros componentes, o `h:panelGrid` nos permite manipular todos os componentes que estão dentro dele como um só. Se quisermos por exemplo renderizar via AJAX todo o conteúdo do `container`, não precisamos passar o `id` de cada componente, basta passar o `id` do `container`. Vamos ver tudo isso com mais calma na seção 9.4, onde falaremos sobre como fazer requisições assíncronas com o JSF.

7.24 H: PANELGROUP

```
<h:selectBooleanCheckbox value="#{compraBean.compra.temDesconto}">
    <f:ajax render="desconto"/>
</h:selectBooleanCheckbox> Compra com desconto

<h:panelGroup id="desconto">
    <h:panelGroup rendered="#{compraBean.compra.temDesconto}">
        % desconto<h:inputText value="#{compraBean.compra.desconto}"/>
    </h:panelGroup>
</h:panelGroup>
```

O `h:panelGroup` é um *container* que serve somente para agrupar outros componentes. Assim, podemos renderizar ou não e atualizar todo o conteúdo a partir de um só componente.

O exemplo apresentado é o mesmo que vimos quando falávamos do componente `h:selectBooleanCheckbox`. Nele podemos ver como o `h:panelGroup` é usado para ser atualizado via AJAX através do seu `id` com valor `"desconto"`. Vemos também como podemos condicionar a renderização de todo o conteúdo de outro `h:panelGroup` através da propriedade `rendered`.

7.25 H: OUTPUTTEXT

```
Marca:    <h:outputText value="#{automovel.modelo.marca}"/>
Modelo:   <h:outputText value="#{automovel.modelo}"/>
```

Nem sempre utilizamos o componente `h:outputText` para escrever alguma saída na nossa aplicação, já que podemos escrever diretamente o texto que desejamos, mesmo que nele haja conteúdo dinâmico. Para isso, basta colocar a expressão que quisermos. Por exemplo, poderíamos reescrever o exemplo acima da seguinte forma:


```
Marca:      #{automovel.modelo.marca}  
Modelo:     #{automovel.modelo}
```

O funcionamento é o mesmo. Colocar o texto diretamente não faz o JSF trabalhar errado ou se perder. Mas enquanto no primeiro exemplo tínhamos quatro componentes (as duas *labels* são componentes implícitos), nesse último temos apenas um. Como já vimos o funcionamento do `h:panelGrid`, sabemos que em alguns momentos o número de componentes influencia no resultado final, e esse é um motivo para o uso do `h:outputText`.

Outro motivo para usarmos o `h:outputText` é a possibilidade de usá-lo em conjunto com conversores. Como os textos comuns (outputs implícitos) não são tags, não temos como colocar um conversor nele. E podemos usar conversores que apresentem corretamente valores monetários ou percentuais, por exemplo.

Marca: Ferrari
Modelo: F-40

Figura 7.15: Resultado da tag `h:outputText`

```
Preço:  
<h:outputText value="#{automovel.preco}">  
    <f:convertNumber type="currency" />  
</h:outputText>
```

Usando esse conversor, baseado na configuração do seu browser, o JSF irá apresentar o número armazenado na propriedade `preco` em Reais ou, dependendo da localização, poderia apresentar em Dólares. Por exemplo, o `Float 1000000.0` seria apresentado assim: “R\$ 1.000.000,00”.

Outro uso para o componente é a possibilidade de especificarmos classes de CSS assim como podemos fazer em qualquer outro componente JSF. Sem o componente, precisaríamos, por exemplo, envolver manualmente o texto em um HTML `span`.

7.26 H:OUTPUTLABEL

```
<h:outputLabel value="Ano de Fabricação:" for="anoFabricacao"/>  
<h:inputText value="#{automovel.anoFabricacao}" id="anoFabricacao"/>
```

```
<h:outputLabel value="Ano do Modelo:" for="anoModelo"/>
<h:inputText value="#{automovel.anoModelo}" id="anoModelo"/>
```

No entanto, ao usarmos o `h:outputLabel` temos uma semântica correta com relação aos *labels* dos campos, já que, em vez de um simples texto, a tag `h:outputLabel` do JSF renderiza a tag `label` do HTML. Seu atributo `for` referencia o `id` do `input` ao qual ele se refere.

7.27 H:OUTPUTFORMAT

```
<h:outputFormat value="Eu moro em {0}, {1}" >
    <f:param value="Campo Grande" />
    <f:param value="MS" />
</h:outputFormat>
```

Usamos o componente `h:outputFormat` quando queremos montar textos parametrizados.

Seus valores podem vir de uma *Expression Language*. Vamos analisar um exemplo um pouco mais complexo.

```
<h:outputFormat value="#{preferenciasBean.bemVindo}" >
    <f:param value="#{session.usuarioLogado.nome}" />
    <f:param value="#{session.usuarioLogado.tratamento}" />
    <f:param value="#{session.usuarioLogado.cargo}" />
    <f:param value="#{session.localidadeSelecionada}" />
</h:outputFormat>
```

7.28 H:OUTPUTSCRIPT E H:OUTPUTSTYLESHEET

```
<h:outputStylesheet library="css" name="facesmotors.css"/>
<h:outputScript library="scripts" name="facesmotors-commons.js"
    target="head"/>
```

Esses dois componentes utilizam o mecanismo padrão do JSF de lidar com recursos como javascripts, folhas de estilo CSS, imagens e até mesmo para a criação de componentes customizados. Varemos mais detalhes sobre esse suporte na seção [7.34](#).

A adição desses componente permite que também adicionemos nossos recursos, seja CSS ou javascript, em qualquer lugar da página, e no HTML renderizado eles

estarão onde quisermos. Por exemplo, os arquivos CSS relacionados com o componente `h:outputStylesheet` estarão sempre no `head` do HTML.

Já o `h:outputScript` possui a propriedade `target` que nos permite indicar onde queremos que o script seja carregado. Se não especificarmos nada, ele será carregado no mesmo local onde o `h:outputScript` foi colocado na página, no entanto, podemos determinar os valores: `head`, `body` e `form` para indicar onde queremos que ele fique.

7.29 H:GRAPHICIMAGE

```
<h:graphicImage url="www.servidor.com/imagem.jpg"/>
<h:graphicImage library="images" name="imagem.jpg"/>
```

Usamos `h:graphicImage` para exibir imagens na nossa aplicação. Basicamente temos dois modos, como pudemos ver no código anterior. O primeiro é usado para exibir uma imagem baseada em sua `url`, e a segunda usamos para exibir imagens de dentro da nossa aplicação.

O primeiro modo apenas renderiza a tag `img` do HTML com a propriedade `src` com o mesmo valor passado na `url` do `h:graphicImage`.

O segundo modo é mais interessante e usa o mesmo suporte a recursos que as tags `h:outputScript` e `h:outputStylesheet`. A propriedade `library` especifica o diretório e a `name` é o nome do arquivo dentro da pasta. A vantagem desse recurso é que não precisamos ficar nos preocupando em montar o *link* relativo da imagem e podemos inclusive referenciar imagens que estão dentro de arquivos `jar`. Veremos mais sobre o suporte a recursos na seção 7.34.

7.30 H:DATATABLE

```
<h:dataTable value="#{automovelBean.automoveis}" var="auto"
    rowClasses="table-linha-par,table-linha-impar" border="1">

    <h:column>
        <f:facet name="header">Marca</f:facet>
        #{auto.modelo.marca}
    </h:column>
    <h:column>
        <f:facet name="header">Modelo</f:facet>
        #{auto.modelo}
```

```

</h:column>
<h:column>
  <f:facet name="header">Ano Fabricação</f:facet>
  #{auto.anoFabricacao}
</h:column>
<h:column>
  <f:facet name="header">Ano Modelo</f:facet>
  #{auto.anoModelo}
</h:column>
</h:dataTable>

```

Utilizamos `h:dataTable` quando desejamos mostrar dados tabulares. Temos duas propriedades principais, `value` e `var`, bem parecido com o componente `f:selectItems`. Na propriedade `value` ligamos a lista que queremos iterar, e na `var` definimos um nome de variável que usaremos para referenciar cada objeto da lista.

Dentro da `h:dataTable` nós definimos as colunas usando a tag `h:column`, e dentro desta podemos definir um cabeçalho pelo `f:facet` “header”, e um rodapé com o `f:facet` “footer”.

Temos também propriedades como `headerClass` e `footerClass` para especificarmos as classes CSS do cabeçalho e do rodapé, respectivamente. Além dessas, como podemos ver no exemplo, temos ainda a propriedade `rowClasses` que permite fazer *zebramento* das linhas, utilizando um estilo CSS para cada linha da tabela; e temos a propriedade `columnClasses` que faz o mesmo para as colunas.

Marca	Modelo	Ano Fabricação	Ano Modelo
Volkswagem	Fusca	1966	1966
Fiat	Punto Sporting	2008	2009
Ferrari	459 Italia	2012	2013
Audi	R8	2010	2010

Figura 7.16: Resultado da tag `h:dataTable`

7.31 UI:REPEAT

```
<ui:repeat value="#{automovelBean.automoveis}" var="auto"
           varStatus="status">
    <motors:automovel automovel="#{auto}"/>
</ui:repeat>
```

A tag `ui:repeat` também é usada para iterar elementos, mas em vez de renderizar uma tabela, ela simplesmente percorre a lista, deixando-nos livres para escolher o tipo de saída que desejamos.

Além das tradicionais propriedades `value` e `var` que funcionam igual as da `h:dataTable`, a `ui:repeat` nos permite, com a propriedade `varStatus`, definir uma variável relacionada ao status da iteração. No nosso exemplo, definimos a variável `status`, e através dela podemos fazer, dentre outras coisas, as seguintes checagens:

- `#{status.first}`: é booleano e indica se é o primeiro elemento da lista;
- `#{status.last}`: é booleano e indica se é o último elemento da lista;
- `#{status.index}`: do tipo inteiro que representa o índice da iteração. Se fosse um `for` tradicional, seria o `int i`;
- `#{status.even}`: é booleano e indica se o `index` é par;
- `#{status.odd}`: é booleano e indica se o `index` é ímpar.

Para utilizar essa tag, é necessário importar os componentes disponíveis do Faceslets. Para isso, é só adicionar no começo do seu `xhtml` o seguinte:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">

</html>
```

7.32 H:COMMANDBUTTON E H:COMMANDLINK

```
<h:dataTable value="#{automovelBean.automoveis}" var="automovel">
```

```
<h:commandLink value="Editar" action="editar">
    <f:setPropertyActionListener value="#{automovel}"
        target="#{automovelBean.automovel}"/>
</h:commandLink>

</h:dataTable>

<h:commandButton value="Salvar"
    action="#{automovelBean.salvar(automovel)}/>
```

[Editar](#)

Figura 7.17: Resultado da tag h:commandLink

Salvar

Figura 7.18: Resultado da tag h:commandButton

A diferença básica entre `h:commandButtons` e `h:commandLinks` é basicamente a forma como são renderizados. Fora isso, o que se aplica a um se aplica também ao outro. No exemplo acima, poderíamos trocar os componentes de lugar sem influenciar no funcionamento. Apesar do `h:commandLink` renderizar um link, ele fará o *submit* do formulário em que estiverem.

Ambos os componentes `command` possuem uma propriedade `action` que em última instância é uma `String`. Essa `String` pode ser literal como no exemplo do link, ou pode ser o retorno do método invocado do *Managed Bean*. A `String` resultante é usada para efetuar a navegação, que veremos na seção 7.41.

No `h:commandLink`, simplesmente colocamos o objeto de contexto da `h:dataTable` na propriedade `automovel` do `automovelBean`. Na prática, ele pegará o `Automovel` que estava sendo apresentado na linha do link que foi acionado e colocará onde o `target` do `f:setPropertyActionListener` aponta. Então, efetuará a regra de navegação baseada na `String` “editar”.

Já no exemplo do botão, ligamos o atributo `action` com a execução de um método. Após efetuar a regra de negócio, essa `action` vai devolver algo que será

usado para a tomada de decisão da navegação, da mesma forma como no caso do link. A diferença é que o retorno pode ser uma `String`, `null`, um método `void` ou ainda retornar um `Object` qualquer, que terá seu método `toString` chamado para a obtenção do resultado para a navegação.

7.33 A DIFERENÇA ENTRE ACTION E ACTIONLISTENER

```
<h:commandButton id="botaoSalvar" value="Salvar"
    actionListener="#{automovelBean.listener}"
    action="#{automovelBean.salvar(automovel)}"/>
```

Apesar de aparentemente parecidos, `action` e `actionListener` são conceitualmente bem diferentes. Enquanto o primeiro é usado para executar a lógica da aplicação, o segundo serve para observarmos eventos de tela. Seria como ficar escutando as ações do usuário, mas sem objetivo de negócio.

Como está apresentado no código, um `actionListener` deve ser um método público e `void` do objeto `#{automovelBean}`, mas temos a opção de não termos nenhum parâmetro ou de receber um `javax.faces.event.ActionEvent`. Esse último caso geralmente é mais interessante, porque a partir desse objeto temos acesso ao componente que originou a ação, que no exemplo é o botão rotulado “Salvar”.

```
public void listener(ActionEvent event){
    UIComponent source = event.getComponent();
    System.out.println("Ação executada no componente "
        + source.getId());
}
```

Ao executar a ação de salvar, a `actionListener` escreveria no console o `id` do componente, que no caso é `botaoSalvar`. Existe também uma outra forma de associar uma `ActionListener` com a ação, através da tag `f:actionListener`.

```
<h:commandButton id="botaoSalvar" value="Salvar"
    action="#{automovelBean.salvar(automovel)}">
    <f:actionListener type="facesmotors.LoggerActionListener"/>
    <f:actionListener type="..."/>
</h:commandButton>
```

Nesse caso, estamos vinculando nossa ação com classes que implementam a interface `ActionListener`. Uma mesma ação pode ter vários *listeners*, mas só conseguimos fazer isso com a tag `f:actionListener`, já que a propriedade

`actionListener` do `h:commandButton` e `h:commandLink` só permite vincular um método. Se especificarmos mais de um *listener*, eles serão executados na ordem que informarmos.

A seguir temos um exemplo de `ActionListener`.

```
public class LoggerActionListener implements ActionListener{

    @Override
    public void processAction(ActionEvent event)
        throws AbortProcessingException {

        UIComponent source = event.getComponent();
        System.out.println("Ação executada no componente "
            + source.getId());

    }

}
```

Note a possibilidade de abortarmos a execução da ação principal, que no caso é salvar o `Automovel`, bastando para isso lançarmos uma `AbortProcessingException`. Isso torna possível, por exemplo, verificar se o usuário tem ou não permissão para executar uma determinada operação.

Agora depois de vermos as `ActionListeners`, podemos compreender melhor o funcionamento da tag `f:setPropertyActionListener`. Ela adiciona na nossa ação uma `ActionListener` que copia um objeto de um local (`value`) para outro (`target`).

7.34 PADRONIZAÇÃO NO CARREGAMENTO DE RECURSOS

No desenvolvimento web em geral precisamos lidar com carregamento de recursos como imagens, arquivos javascript, CSS e outros. Geralmente guardamos todos eles em alguma pasta e os *linkamos* quando necessário. Mas o que acontece quando precisamos componentizar algo? Por exemplo, quando vamos deixar pronto dentro de um `jar`, uma tela de login padrão, que além dos campos de usuário e senha possui uma imagem, como a carregamos?

Até pouco tempo atrás, para lidarmos com recursos dentro de arquivos `jar`, era necessário o uso de uma *Servlet* que fizesse o trabalho por nós. Com o suporte a recursos do JSF, podemos usar a mesma maneira de carregar recursos de dentro

da aplicação ou de dentro de um jar. Podemos ainda versionar um determinado recurso, como a biblioteca do jQuery, por exemplo.

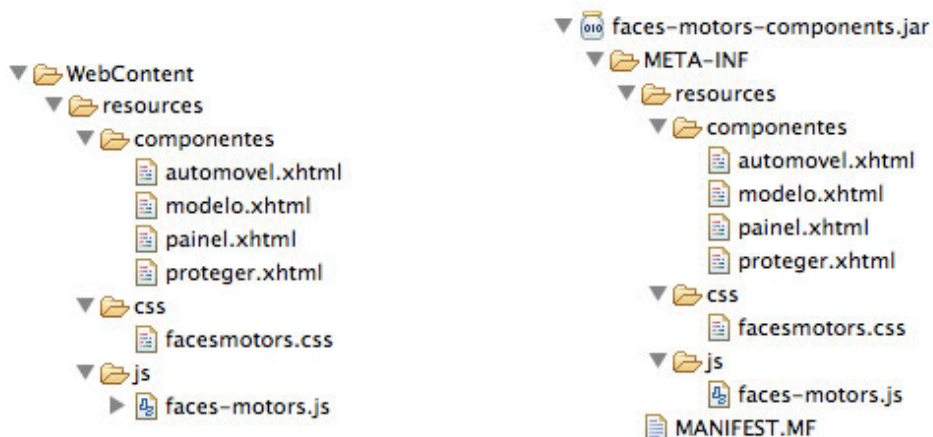


Figura 7.19: A estrutura dos recursos dentro do projeto ou no jar é igual

A base de tudo é a pasta `resources`, que pode tanto estar na raiz do contexto web quanto na pasta `META-INF` dentro de um jar ou no *classpath* da aplicação.

Temos basicamente três componentes básicos do JSF que lidam com recursos que são `h:outputScript`, `h:outputStylesheet` e `h:graphicImage`.

Em comum nesses três componentes temos basicamente duas propriedades: `library` e `name`.

```
<h:outputScript library="scripts" name="facesmotors.js"/>
```

Para o JSF, `library` é a pasta abaixo da pasta `resources`. Conforme a imagem que acabamos de ver, temos basicamente três bibliotecas na nossa aplicação: `images`, `scripts` e `css`. Já a propriedade `name` serve para especificar o nome do recurso que está dentro da `library`.

Essa é a maneira mais simples e natural de se trabalhar com os recursos, no entanto, podemos ir além e trabalharmos de uma maneira mais complexa. Para entendermos o que podemos fazer, vamos dar uma olhada no nome que podemos ter para os recursos:

```
[localePrefix/] [libraryName/] [libraryVersion/] resourceName [/resourceName]
```

Podemos ter desde a versão internacionalizada de um recurso (`localePrefix`), até especificarmos versões, tanto da biblioteca como do recurso.

Então, se estivéssemos desenvolvendo uma biblioteca de componentes chamada “facesmotors”, poderíamos fazer algo parecido com isso:



Figura 7.20: Jars com versões diferentes do mesmo script

Agora, temos duas versões de uma mesma biblioteca, mas basta especificarmos que versão queremos carregar ao usar qualquer um dos componentes com suporte a recursos.

```
<h:outputScript library="facesmotors/1_0"
  name="facesmotors-scripts.js"/>
```

```
<h:outputScript library="facesmotors/1_1"
  name="facesmotors-scripts.js"/>
```

As versões são especificadas usando `_` como separador dos números. Além disso, podemos usar uma *Expression Language* para carregar recursos mesmo onde não estivermos usando algum componente com esse tipo de suporte:

```
<script type="text/javascript"
  src="#{resource['facesmotors/1_1:facesmotors-scripts.js']}"/>
</script>
```

7.35 ENTENDA OS CONVERSORES NATIVOS, O `f:convertDateTime` E O `f:convertNumber`

O JSF possui conversores nativos para transformar as `Strings` recebidas na requisição da aplicação para os tipos básicos como `Integer`, `Float` e alguns outros. Como seu funcionamento é trivial, ou seja, é basicamente uma simples conversão para `String`, essa seção tem como objetivo tratar os conversores que já existem, mas que, diferentemente dos conversores para `Integer` ou `Float`, precisam de algum tipo de configuração extra.

Basicamente temos dois conversores nessa situação: `f:convertDateTime` e `f:convertNumber`. Datas e números podem ser informados ou apresentados de diversas formas, e por isso os usamos, para poder indicar o formato que queremos usar em cada situação.

`f:convertDateTime`

```
<h:inputText value="#{managedBean.objeto.data}">
  <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText>

<h:outputText value="#{managedBean.objeto.data}">
  <f:convertDateTime dateStyle="full" type="both"/>
</h:outputText>
```

O uso mais comum desse conversor é a partir da sua propriedade `pattern`, na qual inserimos o padrão da data que queremos informar ou exibir. Mas caso não queiramos deixar o padrão fixo, podemos utilizar as propriedades a seguir:

`dateStyle`

Pode assumir os seguintes valores com os respectivos resultados:

- `default` (valor padrão): 21/12/2012
- `short`: 21/12/12
- `medium`: 21/12/2012
- `long`: 21 de Dezembro de 2012
- `full`: Sexta-feira, 21 de Dezembro de 2012

type

A propriedade `type` permite configurar se queremos trabalhar somente com data, hora, ou ambos:

- `date` (valor padrão): 21/12/2012
- `time`: 23:59:59
- `both`: 21/12/2012 23:59:59

locale

Podemos informar a localidade. Deve ser uma instância de `java.util.Locale` ou então uma `String`, como “en” ou “pt”.

Caso não informemos o `locale`, o JSF usará o devolvido por `FacesContext.getViewRoot().getLocale()`, que, se não for alterado de alguma forma, será a localidade padrão do sistema (browser) do usuário.

timeZone

Uma característica do JSF que não agrada muito com relação ao `f:convertDateTime` é que ele usa o *time zone* GMT por padrão, e não o horário do sistema. Para mudar isso teríamos que informar `timeZone="GMT-3"`, no caso do horário oficial do Brasil, mas aí teríamos algo fixo. Para resolver esse problema podemos colocar o seguinte parâmetro no `web.xml`.

```
<context-param>
  <param-name>
    javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE
  </param-name>
  <param-value>true</param-value>
</context-param>
```

Com isso, em vez do converter usar por padrão o *time zone* GMT, será usado o do sistema.

f:convertNumber

```
<h:outputText value="#{automovel.preco}">
  <f:convertNumber type="currency"/>
</h:outputText>
<h:outputText value="#{automovel.kilometragem}">
  <f:convertNumber type="number"/>
</h:outputText>
```

Apesar do JSF converter automaticamente números mais simples, quando trabalhamos com números mais “complexos”, por exemplo, representação de dinheiro ou número com separador de milhar, acabamos usando o `f:convertNumber`.

Esse conversor possui muitas propriedades que auxiliam na formatação correta dos números. Podemos dizer o mínimo e o máximo de casas decimais que devem ser usadas (`maxFractionDigits` e `minFractionDigits`), qual o símbolo de dinheiro será utilizado (`currencyCode` ou `currencySymbol`) para exibir R\$ ou US\$, por exemplo, ou ainda um `pattern` qualquer. Podemos também desabilitar a separação de milhar colocando `groupingUsed="false"`.

Também é comum especificarmos o valor da propriedade `type`, na qual indicamos se o valor é “number” (valor padrão), “currency” (indicando dinheiro) ou “percent”.

Outra propriedade importante é a `locale`, que por padrão pega a localidade do browser do usuário. Assim, os caracteres de separação de milhar e decimal (ponto ou vírgula) são usados de acordo com o valor dado para essa propriedade.

7.36 CONVERSORES CUSTOMIZADOS

Como visto na seção 7.3, o JSF é quem gera o HTML que será apresentado para o usuário, mais especificamente na sexta fase do seu ciclo de vida. Caso a tela já venha com algum valor preenchido, por exemplo ao editar uma `Pessoa`, esse objeto precisa ser transformado para `String` para que sua informação possa ser guardada no HTML. Esse processo nos parece simples, já que todo objeto Java possui um método `toString`.

O interessante acontece depois que o usuário edita o formulário e o envia novamente para o servidor. Então aquela `String` que representa um objeto do nosso domínio precisa voltar a ser um objeto, por exemplo, do tipo `Pessoa`. Nesse ponto lembramos que o Java não tem um método `fromString` que faça o inverso do `toString`. Justamente por isso, temos os conversores.

Conversores são implementações da interface `javax.faces.convert.Converter` e possuem dois métodos que seriam os correspondentes ao `toString` e um hipotético `fromString` do Java. Sendo assim, a interface `Converter` possui os métodos `getAsString`, que recebe um `Object` e devolve a `String`, e um método `getAsObject` que recebe a `String` e devolve o `Object`. Simples assim.

Implemente conversores customizados

Agora que compreendemos a motivação para os conversores, vamos implementar um usando a JPA. Considere um `h:inputText`, definido da seguinte forma:

```
<h:inputText value="#{automovelBean.automovel}" />
```

`automovelBean.automovel` mapeia diretamente para um objeto do tipo `Automovel`, então, vamos criar um conversor que, quando o campo for submetido, ao mostrar a informação do automóvel, em vez de mostrar o `toString` do objeto, mostre algo melhor.

Vamos fazer o nosso `Converter` para `Automovel`.

```
@FacesConverter(forClass=Automovel.class)
public class AutomovelConverter implements Converter {

}
```

O primeiro detalhe que podemos observar é que além de implementar a interface `Converter`, um conversor precisa ser registrado no contexto do JSF. Isso pode ser feito via XML ou via a anotação `@FacesConverter`, que é o que estamos usando.

A partir da propriedade `forClass` registramos nosso conversor para atuar toda vez que o JSF precisar converter uma `String` para a classe `Automovel`. O primeiro método dessa interface que será implementado é o `getAsString`.

```
public String getAsString(FacesContext context, UIComponent component,
                          Object object) {

    Automovel automovel = (Automovel) object;
    if(automovel == null || automovel.getId() == null) return null;

    return String.valueOf(automovel.getId());
}
```

Na implementação, simplesmente devolvemos a propriedade `id` do `Automovel`. Ou seja, sempre que quisermos mostrar na tela um `Automovel`, o que será exibido é o seu `id`.

```
public Object getAsObject(FacesContext context, UIComponent component,
                          String string){

    if(string == null || string.isEmpty()) return null;
    Integer id = Integer.valueOf(string);
```

```
Automovel automovel = entityManager.find(Automovel.class, id);  
return automovel;  
}
```

Aqui nós temos o método que completa a ação do converter. A partir da `String` devolvida pelo método `getAsString`, recuperamos o objeto que originalmente estava ligado ao nosso modelo. Caso a `String` passada seja nula ou em branco, devolveremos uma referência nula.

Uma vez que obtemos o valor submetido, convertemos o mesmo para o tipo da chave da nossa classe, no caso `Integer`, e em seguida buscamos esse objeto no banco de dados através do método `find` ou mesmo `getReference` da interface `EntityManager`.

Criar conversores é algo bastante útil quando se está trabalhando com componentes de lista, como o `selectOneMenu`, por exemplo. Assim, é possível indicar os valores dos `selectItems` desse componente como sendo objetos, e não pelos ids.

7.37 CONHEÇA OS VALIDADORES NATIVOS

Seguindo o ciclo de vida do JSF, logo após a conversão, vem a validação. Por isso, agora vamos ver quais as formas de validação o JSF traz prontas para usarmos. Como exemplo, vamos considerar um trecho de tela como o seguinte:

```
<h:panelGrid columns="2">  
  Ano de Fabricação:  
    <h:inputText value="#{auto.anoFabricacao}">  
      <f:validateLongRange minimum="1950" maximum="2012"/>  
    </h:inputText>  
  
  Ano do Modelo:  
    <h:inputText value="#{auto.anoModelo}">  
      <f:validateLongRange minimum="1950" maximum="2013"/>  
    </h:inputText>  
  
  Preço: <h:inputText value="#{auto.preco}" />  
  
  Kilometragem: <h:inputText value="#{auto.kilometragem}" />
```

Observações: `<h:inputTextarea value="#{auto.observacoes}" />`
`</h:panelGrid>`

Tanto a propriedade `minimum` quanto a `maximum` aceitam *Expression Language*, caso necessário. Apesar de estarmos colocando esses validadores no XHTML, como já aprendemos sobre o ciclo de vida, sabemos que isso vai virar uma informação para o JSF validar nossos dados do lado servidor.

Temos várias opções de validadores disponíveis, e todos eles têm algumas propriedades em comum:

- `disabled`: que aceita uma expressão e podemos usar para desligar condicionalmente um validador. Por exemplo, podemos ter uma regra de negócio que nos faça deixar ligado um validador apenas se o status do objeto sendo salvo for diferente de “rascunho”; ou ainda deixar que o usuário administrador possa enviar dados sem nenhuma restrição.

```
<h:inputText value="#{auto.anoFabricacao}">
  <f:validateLongRange
    minimum="1950"
    maximum="2012"
    disabled="#{currentUser.role == 'admin'}"/>
</h:inputText>
```

- `for`: essa propriedade é usada quando trabalhamos com *composite components*. Veremos como criar esses componentes e então usaremos essa propriedade na seção 9.6.
- `binding`: assim como qualquer outro componente JSF, podemos usar a propriedade `binding` para ligar o objeto da árvore do JSF com uma propriedade do *Managed Bean* e assim temos como manipulá-lo de forma programática.

f:validateLongRange e f:validateDoubleRange

São usados para validar números inteiros e reais. Podemos notar que não temos validadores para `Integer` e `Float`, já que esses são apenas versões com menor capacidade de armazenamento que `Long` e `Double`.

Propriedades:

- **minimum**: aceita *Expression Language*, e indica o mínimo valor que o número em questão pode ter.

- **maximum:** aceita *Expression Language*, e indica o máximo valor que o número em questão pode ter.

f:validateLength

É bem parecido com os validadores `f:validateLongRange` e `f:validateDoubleRange`, a diferença é que enquanto estes trabalham com valores numéricos e seus intervalos, o `f:validateLength` trabalha com comprimento máximo e mínimo de uma `String`. Podemos, por exemplo, usá-lo para validar o comprimento mínimo de uma senha.

Propriedades:

- **minimum:** aceita *Expression Language*, e indica o comprimento mínimo que a `String` em questão pode ter.
- **maximum:** aceita *Expression Language*, e indica o comprimento máximo que a `String` em questão pode ter.

f:validateRequired

Valida a obrigatoriedade ou não do preenchimento do valor de um componente de entrada de dados. Na prática, gera o mesmo resultado que a propriedade `required` do componente.

```
<h:inputText value="#{auto.preco}">
  <f:validateRequired disabled="#{currentUser.role == 'admin'}"/>
</h:inputText>
```

Ou com o mesmo resultado:

```
<h:inputText value="#{auto.preco}"
  required="#{currentUser.role != 'admin'}"/>
```

f:validateRegex

Esse nos permite criar uma validação baseada em expressão regular. Apesar de termos validadores prontos em projetos como o Caelum Stella, disponível em <http://stella.caelum.com.br/>, podemos utilizar expressões para validar CEP, e-mail, e outros formatos. É importante salientar que esse validador não gera nenhuma máscara para o campo. Ele apenas valida um determinado padrão do lado servidor.

Esse validador possui apenas uma propriedade:

- **pattern:** podemos colocar qualquer expressão que desejarmos. Por exemplo, uma expressão para validar um login que possua entre 6 e 18 caracteres, aceitando somente letras minúsculas.

```
<h:inputText value="#{user.login}">
  <f:validateRegex pattern="[a-z]{6,18}" />
</h:inputText>
```

Nesse caso, fizemos uma expressão regular que indica que a senha só pode ter letras de `a` até `z` e deve estar entre 6 e 18 caracteres.

f:validateBean

Na seção 8.4 veremos mais detalhes da integração entre JSF e *Bean Validation*. Mas em resumo, essa tag é usada para especificar os grupos que desejamos validar. Isso porque, apesar de a nossa classe conter as anotações da *Bean Validation*, em uma tela como um wizard por exemplo, vamos querer validar apenas parte das propriedades em cada momento.

Propriedade:

- **validationGroups:** informamos o grupo de propriedades que desejamos validar. Caso queiramos informar mais de um grupo, informamos todos eles separados por vírgula.

Podemos desabilitar a validação via *bean validation* em determinadas propriedades como a seguir.

```
<h:panelGrid columns="2">
  Ano de Fabricação:
  <h:inputText value="#{auto.anoFabricacao}">
    <f:validateBean disabled="true" />
  </h:inputText>

  Ano do Modelo:
  <h:inputText value="#{auto.anoModelo}">
    <f:validateBean disabled="${auto.status == 'rascunho'}" />
  </h:inputText>

  Preço:
  <h:inputText value="#{auto.preco}" />
```

```
Kilometragem:
<h:inputText value="#{auto.kilometragem}" />

Observações:
<h:inputTextarea value="#{auto.observacoes}" />
</h:panelGrid>
```

Mas geralmente faríamos isso em situações como a mencionada acima, na qual temos um wizard ou alguma tela de cadastro não tão trivial em que aqueles valores, apesar de obrigatórios, não precisam ser validados naquele momento, provavelmente porque serão providos depois. Nesses casos, uma possibilidade é trabalharmos com grupos e usar algo parecido com o seguinte:

```
<f:validateBean validationGroups="cadastroBasico">
  <h:panelGrid columns="2">
    Ano de Fabricação:
    <h:inputText value="#{auto.anoFabricacao}" />

    Ano do Modelo:
    <h:inputText value="#{auto.anoModelo}" />

    Preço:
    <h:inputText value="#{auto.preco}" />

    Kilometragem:
    <h:inputText value="#{auto.kilometragem}" />

    Observações:
    <h:inputTextarea value="#{auto.observacoes}" />
  </h:panelGrid>
</f:validateBean>
```

Agora sim, considerando que essa tela corresponde a um cadastro básico, campos que são obrigatórios mas que não são parte desse grupo não seriam validados agora.

Outro ponto interessante nos exemplos apresentados é que podemos tanto colocar a tag dentro de um determinado componente, como em volta de todos os componentes que gostaríamos que tivessem o mesmo tratamento. Muito menos repetitivo.

7.38 E QUANDO OS VALIDADORES NATIVOS NÃO FAZEM O QUE EU QUERO? CRIE SEUS VALIDADORES

Já vimos várias formas de validar os valores que o usuário submete. Mas muitas vezes, temos validações que são específicas do nosso domínio, como verificar se um CPF digitado é efetivamente válido.

Considere que o usuário pode fazer o `Pedido` de um `Produto` pelo `codigo` desse `Produto`. Vamos supor que, nesse cenário, o usuário possa fazer “pedidos populares”, nos quais é dado um desconto de 20% (vinte por cento) no preço do produto, desde que o valor unitário não ultrapasse R\$ 200,00. Para conseguirmos fazer isso, vamos ter o seguinte código:

```
@Entity
public class Produto {
    @Id @GeneratedValue
    private Integer id;

    @Column(unique=true)
    private String codigo;
    private Float valor;
}

@Entity
public class Pedido {
    @Id @GeneratedValue
    private Integer id;

    @ManyToOne
    private Produto produto;
    private Integer quantidade;
    private boolean pedidoPopular;
}
```

E o `xhtml` mostrando o formulário:

```
<h:form>
    <h:panelGrid columns="2">
        Pedido popular?
        <h:selectBooleanCheckbox
```

```

        value="#{pedidoBean.pedido.pedidoPopular}">
    <f:ajax/>
</h:selectBooleanCheckbox>

Produto:
<h:inputText id="produto" value="#{pedidoBean.pedido.produto}"
            converter="produtoConverter">

    <f:validator validatorId="produtoPedidoPopularValidator"
        disabled="#{!pedidoBean.pedido.pedidoPopular}" />

</h:inputText>

Quantidade:
<h:inputText value="#{pedidoBean.pedido.quantidade}" />

<h:panelGrid/>
</h:form>

```

O que percebemos de novidade nesse código é a presença da tag `f:validator` no componente `h:inputText` de `id="produto"`. Nesse validador, cujo código veremos a seguir, checamos se a restrição de negócio de valor máximo de R\$ 200,00 está ou não sendo respeitada. Utilizamos também a propriedade `disabled` para somente aplicar o validador caso a opção de “pedido popular” não esteja selecionada. Além disso, indicamos que o `validatorId` é `produtoPedidoPopularValidator`, que é uma referência para um `Validator` que vamos criar.

Como estamos trabalhando com validadores, sabemos que eles se executam na terceira fase do ciclo de vida do JSF, já a atualização do nosso modelo acontece só na quarta fase. Sendo assim, o valor da expressão `#{!pedidoBean.pedido.pedidoPopular}` só será atualizada na quarta fase, ou seja, depois do momento em que o conversor se executaria.

Geralmente em casos assim, o resultado acaba sendo o inverso do esperado, já que, como o valor de `pedidoPopular` originalmente é falso, então o conversor não será desativado na primeira requisição. Se o usuário selecionar “pedido popular”, o validador só será desativado na próxima requisição. Logo, teríamos sempre o resultado esperado - validador habilitado ou desabilitado corretamente - com uma requisição de atraso.

Por esse motivo, no `xhtml` colocamos a tag `f:ajax` sem nenhuma proprie-

dade especificada. Isso somente executa um pequeno *request*, atualizando o valor do próprio componente no modelo e não trazendo nada como resposta. Dessa forma, antecipamos a atualização da propriedade `pedidoPopular` para que o validador seja desabilitado ou não no momento correto.

Veremos na seção 9.4 mais detalhes sobre o uso da tag `f:ajax` e na seção 10.7 como utilizar AJAX de modo eficiente para não comprometer a performance da aplicação.

A seguir temos a definição do nosso validador:

```
@FacesValidator("produtoPedidoPopularValidator")
public class ProdutoPedidoPopularValidator implements Validator{

}
```

Podemos notar que o validador funciona de forma bem parecida com um conversor customizado. Ele implementa a interface `javax.faces.validator.Validator` e é registrado com um id através da anotação `@javax.faces.validator.FacesValidator`.

Ao criar um validador customizado, temos a implementar um único método, que deve lançar uma `ValidatorException`, caso o valor seja inválido.

```
public void validate(FacesContext context, UIComponent component,
                    Object value) throws ValidatorException {
    // objeto já vem convertido
    Produto produto = (Produto) value;
    if(produto.getValor() > 200.0){
        FacesMessage message =
            new FacesMessage("Pedido popular: máximo R$ 200");
        message.setSeverity(FacesMessage.SEVERITY_ERROR);
        throw new ValidatorException(message);
    }
}
```

A implementação é simples: precisamos apenas lançar uma `ValidatorException` que recebe uma ou mais `FacesMessage`, e que pode opcionalmente também receber uma `Throwable`. Caso o validador execute sem lançar exceção, significa que o valor é válido.

No caso de regras muito específicas, é possível deixá-las direto no *Managed Bean*. Para isso, bastaria implementar um método que recebe os três parâmetros que os

validadores recebem, `FacesContext`, `UIComponent` e `Object`, dando um nome arbitrário a ele.

```
@ManagedBean
public class PedidoBean {

    private Pedido pedido;
    ...
    public void validaProdutoPedidoPopular(FacesContext context,
        UIComponent component, Object value)
        throws ValidatorException {

        //objeto vem convertido
        Produto produto = (Produto) value;
        if(pedido.isPedidoPopular() && produto.getValor() > 200.0){
            FacesMessage message =
                new FacesMessage("Pedido popular: máximo R$ 200");
            message.setSeverity(FacesMessage.SEVERITY_ERROR);
            throw new ValidatorException(message);
        }
    }
}
```

E por fim, referenciar esse validador em um componente, como o `inputText`:

```
<h:inputText id="produto"
    value="#{pedidoBean.pedido.produto}"
    converter="produtoConverter"
    validator="#{pedidoBean.validaProdutoPedidoPopular}"/>
```

7.39 NOVIDADE: JSF COM GET E BOOKMARKABLE URLS

Nessa seção veremos alguns componentes que nos possibilitam usar *bookmarkable URLs*, ou seja, URLs que podemos colocar em nosso *bookmark*, ou favoritos.

Agora que já vimos como funcionam os componentes básicos do JSF e também já entendemos como funcionam os conversores e validadores, poderemos tirar melhor proveito desse suporte que foi adicionado no JSF versão 2.0 e aprimorado na versão 2.2.

7.40 H:BUTTON E H:LINK

```
<h:dataTable value="#{automovelBean.automoveis}" var="automovel">

    <h:button value="Editar" outcome="editar">
        <f:param name="automovel" value="#{automovel.id}"/>
    </h:button>

    <h:link value="Editar" outcome="editar" includeViewParams="true">
        <f:param name="automovel" value="#{automovel.id}"/>
    </h:link>

</h:dataTable>
```

Novamente, os componentes que renderizam botão e link têm o mesmo comportamento, mas no caso do `h:button` e `h:link`, eles são elementos que não submetem o formulário onde estão. Por isso mesmo nem precisam estar dentro de um formulário. Esses componentes geram requisições do tipo `GET`, e não `POST` como os componentes `command`.

Algo muito importante a notarmos é a propriedade `outcome`. Como não temos uma maneira de executar uma ação e assim descobrir o resultado para a navegação, chamado `outcome`, temos que deixar explícito qual é esse resultado. Baseado nesse `outcome`, o JSF executará a regra de navegação para descobrir qual link deverá montar.

Depois que o JSF sabe qual página será alvo do link pela regra de navegação, caso a propriedade `includeViewParams` esteja com valor igual a `true`, ele identifica se a página alvo possui `f:viewParam`; caso possua, ele tenta gerar link ou botão com os parâmetros necessários na tela seguinte.

A tela que será exibida, por exemplo a `editarAutomovel.xhtml`, poderá ter um código como o seguinte:

```
<f:metadata>
    <f:viewParam name="automovel" value="#{automovelBean.automovel}"/>
    <f:viewParam name="usuario" value="#{session.currentUser}"/>
</f:metadata>
```

Sendo assim, considerando que tenhamos um conversor para usuário que o transforme para `String` devolvendo seu `username`, e que o `username` do usuário logado é “admin”, o link gerado pelo `h:link` seria algo parecido com o

seguinte: <http://servidor.com.br/faces-motors/editarAutomovel.xhtml?automovel=90&usuario=admin>

A tag `f:metadata`, apenas delimita uma área para a definição de metadados sobre a página, como os parâmetros. Ainda veremos sobre `f:viewParam`, mas sua principal finalidade é explicitar as informações que a tela precisa para funcionar. Usando `includeViewParams=true`, mesmo que essas propriedades não estejam informadas via um `f:param` no `h:link`, o JSF renderiza o link usando essas informações para que o possamos salvar nos *bookmarks*.

f:viewParam

```
<f:metadata>
    <f:viewParam name="automovel" value="#{automovelBean.automovel}"/>
    <f:viewParam name="usuario" value="#{session.currentUser}"/>
</f:metadata>
```

Os parâmetros da *view* servem para que possamos trabalhar corretamente com requisições GET no JSF. Por exemplo, podemos montar um link em outra aplicação, feita com outra tecnologia, e enviar para nossa aplicação em JSF, ou podemos simplesmente acessar um link salvo nos nossos *bookmarks*. Esse link irá conter as informações que serão carregadas no nosso modelo através de cada componente `f:viewParam`.

Podemos pensar no `f:viewParam` como se fosse um `h:inputText`, mas, em vez de entrarmos com os dados em um formulário, fazemos isso via URL. Podemos colocar conversores, validadores e tudo mais que temos disponível com os *inputs* normais.

Para entendermos melhor, considere que a URL em questão é a mesma apresentada no exemplo anterior.

<http://servidor.com.br/faces-motors/editarAutomovel.xhtml?automovel=90&usuario=admin>

Assumindo que tanto `Automovel` quanto `Usuario` possuem conversores adequadamente implementados, os objetos serão atribuídos ao que é apontado pelo `value` do `f:viewParam` e a tela executará normalmente.

f:viewAction

```
<f:metadata>
    <f:viewParam name="modelo"
```

```
        value="#{automovelBuscaBean.modelo}"/>
    <f:viewParam name="anoMinimo"
        value="#{automovelBuscaBean.anoMinimo}"/>
    <f:viewParam name="anoMaximo"
        value="#{automovelBuscaBean.anoMaximo}"/>

    <f:viewAction action="#{automovelBuscaBean.buscarAutomoveis}"/>
</f:metadata>
```

Esse componente é novo na versão 2.2 do JSF e serve para auxiliar no processo de construção de uma tela a partir de uma URL que pode vir de outra aplicação ou dos nossos *bookmarks*. Por padrão, essa `action` vai executar como se fosse o pressionar de um `h:commandButton` ou `h:commandLink`. Ele executa na quinta fase do ciclo de vida do JSF e só na primeira vez que abrimos a tela, não executando nas submissões para a mesma página, que chamamos de “postback”. Caso essa ação precise ser chamada todas as vezes, precisaríamos habilitar isso por meio da propriedade `onPostback=true`.

Quando essa ação se executar, os valores associados ao `f:viewParam` já estarão carregados e faremos apenas o carregamento de outras informações necessárias para a página. Como efetuar a busca segundo os parâmetros informados, por exemplo.

7.41 REGRAS DE NAVEGAÇÃO

Ao gravarmos uma `Marca` o ideal é que sejamos enviados para a tela de listagem. O mecanismo tradicional para fazer o redirecionamento, presente no JSF desde sua primeira versão, é o uso de resultados lógicos a cada ação, e depois vincular cada um desses com a próxima página a ser exibida. Como estamos falando de navegação entre páginas, o caminho completo das páginas de origem e destino precisa ser conhecido.

Nesse exemplo temos duas páginas: `editar.xhtml` e `listar.xhtml`, ambas dentro da pasta `marca`, como pode ser visto na figura a seguir. Vamos analisar o uso de resultados lógicos, chamados de *outcome*, para navegar entre essas páginas.

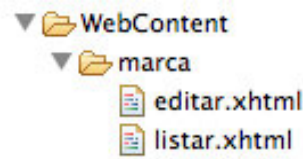


Figura 7.21: Estrutura da pasta ‘marca’

```

@ManagedBean @ViewScoped
public class MarcaBean {
    private Marca marca = new Marca();

    public String salvar(){
        try {
            EntityManager em = JpaUtil.getEntityManager();
            em.persist(marca);
            return "sucesso";
        }
        catch(Exception e){
            return "falha";
        }
    }

    // getters e setters...
}

```

Apenas retornamos um resultado lógico, uma `String` “sucesso” ou “falha”. Em qualquer projeto JSF, é possível ter um arquivo de configuração, o `faces-config.xml`, em que nós configuramos para qual página cada resultado desse deve enviar o usuário.

```

<faces-config ...>
    <navigation-rule>
        <from-view-id>/marca/editar.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>sucesso</from-outcome>
            <to-view-id>/marca/listar.xhtml</to-view-id>
            <redirect/>
        </navigation-case>
        <navigation-case>
            <from-action>#{marcaBean.excluir}</from-action>

```

```
<from-outcome>sucesso</from-outcome>
<to-view-id>/home.xhtml</to-view-id>
<redirect/>
</navigation-case>
<navigation-case>
  <from-outcome>falha</from-outcome>
  <to-view-id>/marca/editar.xhtml</to-view-id>
</navigation-case>
</navigation-rule>
</faces-config>
```

A estrutura básica é assim: uma `<navigation-rule>` possui opcionalmente uma `<from-view-id>`, na qual especificamos que essa regra se aplica às navegações que partirem dessa página. Podemos usar “*” (asterisco) para indicar uma parte do caminho da página, ou então deixar somente o asterisco para indicar que é uma regra global, que se aplica a qualquer página do sistema.

Após especificarmos a página de origem da regra, definimos `navigation-cases`, que funcionam com um `switch case`, no qual a variável analisada é o `outcome` e opcionalmente a própria ação que foi executada. Como podemos ver, no `faces-config.xml` podemos usar o mesmo `outcome` em mais de uma `action`. Considerando que a `action` executada foi `salvar()`, a próxima página a ser exibida será `/marca/listar.xhtml`. Por mais que exista outro caso de navegação que aceite o mesmo `outcome`, essa outra só é válida caso a ação processada seja `#{marcaBean.excluir}`.

Em ambos os casos, vemos a presença da tag `<redirect/>`, que serve para indicar que, em vez de um `forward`, o JSF deverá fazer um `redirect` para a próxima página.

No final, temos o caso que considera o resultado `falha`. Nessa regra, nós colocamos a próxima página igual à original, ou seja, em caso de erro, não mudaremos de página. No entanto, podemos remover esse último caso sem prejuízo, já que, mesmo que o JSF não encontre nenhum caso de navegação configurado para o `outcome` `falha`, não ocorrerá nenhum erro. Isso porque o comportamento padrão é ficar na mesma página quando não existe nenhum caso de navegação compatível.

Navegações implícitas

Apesar da navegação através de um resultado lógico ser bem flexível no caso de quisermos mudar a página que será exibida depois, no cotidiano do desenvolvi-

mento de aplicações não é muito comum isso acontecer. Na prática, muitas vezes tínhamos que fazer uma configuração que dificilmente mudaria, além de termos um arquivo XML com imensas regras de navegação, apesar de existirem diversas ferramentas visuais para construí-las.

Na versão 2.0, foi acrescentado ao JSF o suporte de navegação implícita. Retornado o *outcome*, é buscada uma regra de navegação no `faces-config.xml` e caso nenhuma seja encontrada, é buscado no mesmo diretório da *view* atual, uma página com o nome igual ao *outcome* retornado. Vamos tomar como exemplo a mesma estrutura da figura 7.21.

Como dificilmente teremos uma página chamada `sucesso.xhtml`, vamos alterar o código do método `salvar` da classe `MarcaBean`.

```
public String salvar(){
    EntityManager em = JpaUtil.getEntityManager();
    em.persist(marca);
    return "listar";
}
```

Usando a navegação implícita, podemos deixar de usar as `navigation-rules` no `faces-config.xml`, então vamos considerar que apagamos as regras vistas anteriormente.

Agora quando o método `salvar` executar com sucesso, como nenhuma regra para o *outcome* “listar” está definida, automaticamente o JSF exibirá a *view* `listar.xhtml` que estiver na mesma pasta da *view* atual.

Por fim, é importante ficar atento ao fato de que, na navegação implícita, assim como através do `faces-config.xml`, por padrão o JSF realiza apenas um *forward* para a *view* que será exibida. Vimos que podemos mudar isso no `faces-config.xml` por meio da tag `<redirect />`. No caso da navegação implícita, basta indicar a necessidade do *redirect* na *String* que está sendo retornada, com `?faces-redirect=true`. Assim, o método `salvar` que redireciona para o `listar.xhtml` ficaria:

```
public String salvar(){
    EntityManager em = JpaUtil.getEntityManager();
    em.persist(marca);
    return "listar?faces-redirect=true";
}
```

Além disso, quando o método é `void` e não temos nenhuma navegação definida, significa que a tela a ser renderizada é a mesma tela de origem.

7.42 ENTENDA OS ESCOPOS E SAIBA COMO E QUANDO TRABALHAR COM CADA UM

Desde a época em que programávamos as nossas Servlets, temos contato com três escopos: `request`, `session` e `application`. Esses mesmos escopos eram os únicos disponíveis até a última versão da série 1.X do JSF.

O resultado disso era simples: funcionalidades simples eram construídas utilizando escopo `request`, mas qualquer outra um pouco mais complexa nos obrigava a usar um escopo maior, como o `session`.

O objetivo dessa seção é entendermos quando usar cada escopo, sem utilizar algo maior do que precisamos só para evitar algum problema.

Request Scope

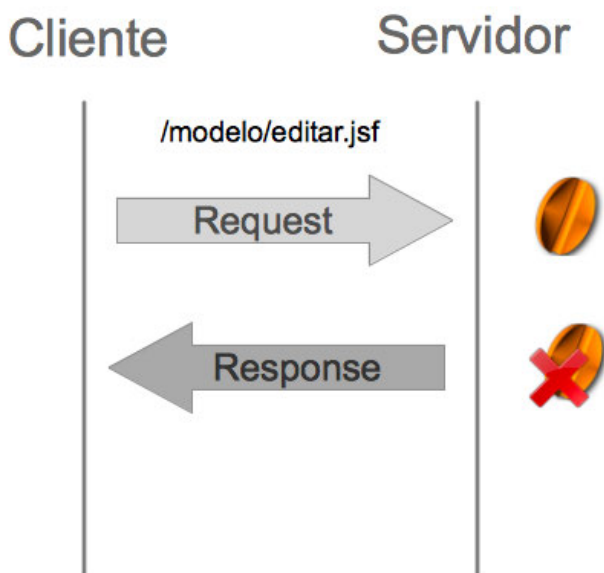


Figura 7.22: Escopo request

```
@ManagedBean
@RequestScoped
public class MeuBean {
```

```
}
```

Objetos armazenados no escopo `request` sobrevivem por uma passada no ciclo de vida do JSF, ou seja, desde que a requisição do usuário chega ao servidor (fase 1) até que a resposta seja enviada a esse usuário (fase 6). Por ser um escopo que tem uma vida curta, seria o preferível para utilizarmos, afinal, quanto menos tempo nossos objetos viverem, mais cedo a memória do servidor será liberada e com isso nossa aplicação tende a escalar melhor.

Como quase sempre na programação não há receita de bolo a seguir, apesar de ser uma boa prática usarmos sempre o menor escopo possível, isso não significa que usaremos sempre o escopo `request`. Esse escopo é apropriado quando não precisamos memorizar dados entre as requisições dos usuários. Se guardamos em memória várias ações do usuário, como buscar vários funcionários para vinculá-los a um departamento, e somente no final salvamos as informações, é bem provável que `request` não seja a melhor opção. Agora, se ao editarmos os dados de uma única pessoa tivermos apenas campos que não precisam ser calculados com base em outros campos, então provavelmente poderemos utilizar esse escopo.

O escopo `request` é o padrão dos *Managed Beans*, dessa forma, ao não anotar sua classe, esse escopo será utilizado.

Session Scope

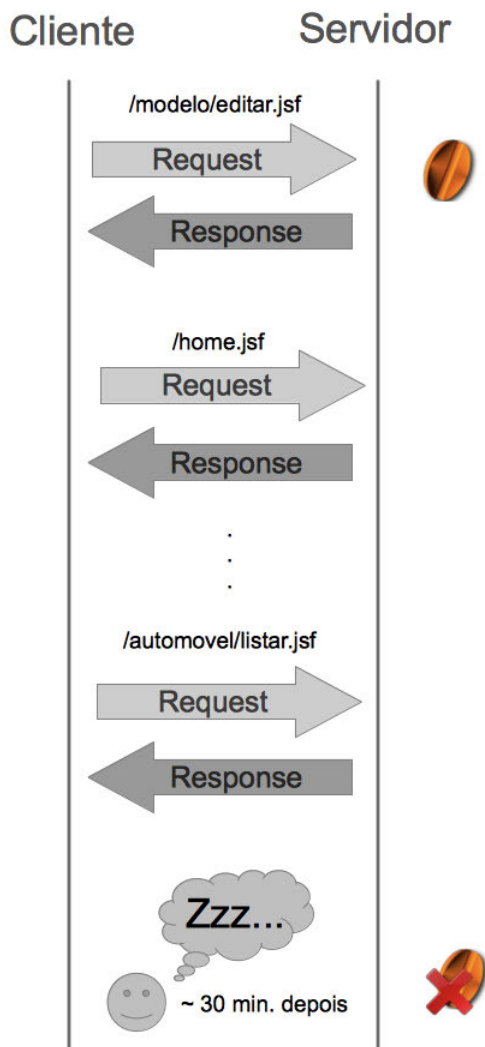


Figura 7.23: Escopo session


```
@ManagedBean
@SessionScoped
public class MeuBean {

}
```

Nesse escopo, tudo que armazenarmos ficará disponível enquanto a sessão do usuário estiver ativa. A sessão do usuário é a ligação do navegador com o servidor Java, então se o mesmo usuário abrir dois navegadores diferentes apontando para a mesma aplicação, ele terá duas sessões diferentes.

Geralmente esse escopo é usado para guardar informações que precisam permanecer por muito tempo, mas que são referentes a um único usuário. Um exemplo, além das informações do próprio usuário logado, é armazenar informação sobre a última operação realizada por ele, para proporcionar uma forma fácil de voltar a ela. Considere que um usuário acabou de editar um `Automovel` e foi para outra parte da aplicação. Depois ele volta à tela de edição de automóveis e tem a opção de reabrir o último `Automovel` editado por ele.

Application Scope

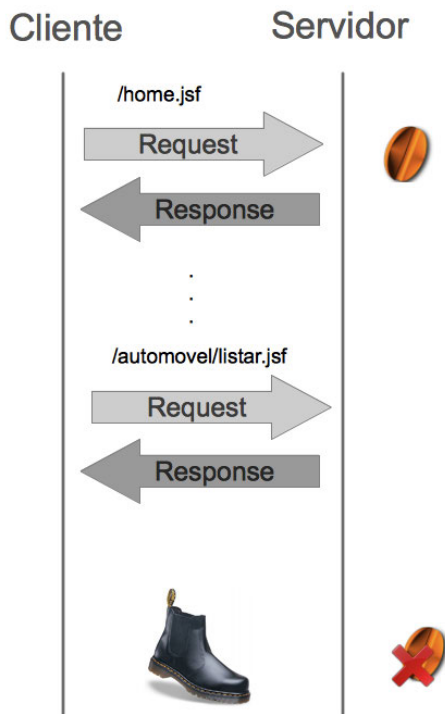


Figura 7.24: Escopo application

```
@ManagedBean
@ApplicationScoped
public class MeuBean {

}
```

Tudo que é armazenado no escopo de aplicação permanece enquanto a aplicação estiver executando, e é compartilhada entre todos os usuários. Nesse escopo é possível guardar informações como os usuários que estão online na aplicação e disponibilizar alguma funcionalidade de comunicação entre eles. É comum também a

realização de cache manual de alguns valores nesse escopo, como por exemplo uma listagem de estados e municípios.

View Scope

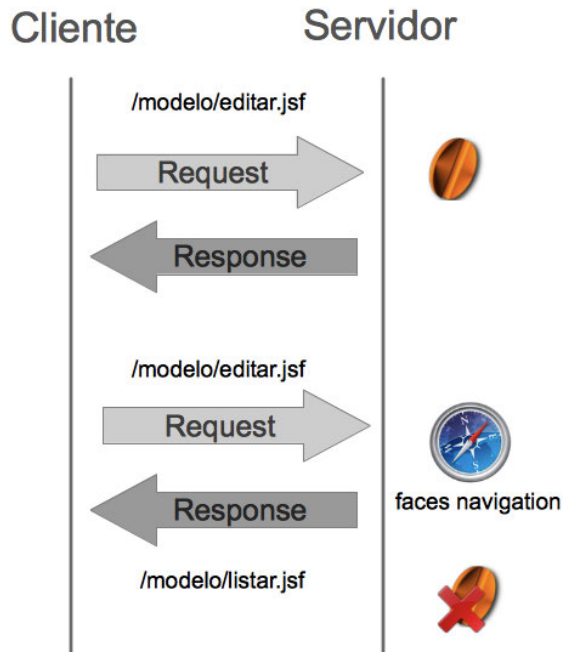


Figura 7.25: Escopo view

```
@ManagedBean
@ViewScoped
public class MeuBean {

}
```

A versão 2 do JSF acrescentou um escopo que desse suporte ao modelo *stateful* do framework, que não onerasse tanto a escalabilidade da aplicação, como o caso do escopo de sessão. Com isso, surgiu o `@ViewScoped`.

Esse escopo consiste em manter os dados contidos nele por quantas requisições forem feitas, mas desde que sejam todas para a mesma *view*. No momento em que trocamos de página o escopo é zerado. Isso é muito bom, porque evita que acumulemos objetos que ficam vivos por muito tempo, como no escopo de sessão, mas ao mesmo tempo permite ações feitas em sequência, como combos em cascata, que nesse escopo funcionam perfeitamente.

Apesar de muito útil, é necessário atentar para alguns detalhes importantes desse escopo. O principal deles é que o objeto no escopo *view* só é removido da memória se a mudança de página for feita via um *POST*. Se simplesmente acessarmos um link que aponta para outra página, os dados da tela anterior continuarão na memória. Esse comportamento permite abrirmos um link em outra aba sem perder os dados da página original. Mas se não lembrarmos de que somente com navegação via *POST* o escopo é destruído, podemos ter objetos desnecessários na memória.

7.43 A CURVA DE APRENDIZADO DO JSF

São muitos conceitos novos caso você nunca tenha usado o JSF antes, e com isso, a curva de aprendizado no começo tende a ser um pouco mais complicada. Isso é comum. Nesse capítulo você aprendeu diversas características que fazem do JSF um framework único entre os disponíveis em Java e espero que você esteja pronto para aplicar todos esses conceitos em seus projetos.

Mas ainda não é tudo, temos mais por ver. agora que já aprendemos as características, vamos poder nos elevar a um nível em que usaremos as técnicas avançadas e várias outras boas práticas.

CAPÍTULO 8

Validações simplificadas na JPA e no JSF com a Bean Validation

Neste capítulo será apresentada uma breve visão sobre a *Bean Validation* e como usá-la integrada a outras especificações que são alvo deste livro.

No capítulo 7 vimos o funcionamento do JSF, e como ele trata questões de validação. Vimos como é simples adicionar uma validação de campo obrigatório ou um intervalo permitido para um valor numérico.

Como validações são requisitos de qualquer aplicação, passou-se a adotá-las no modelo e não na apresentação ou controle do MVC. Foi então que se popularizou, impulsionado pelo sucesso do Hibernate, o Hibernate Validator, que acabou inspirando uma especificação chamada *Bean Validation* (JSR-303). Com ela, nós colocamos as restrições via anotação nas nossas entidades, então tanto o JSF quanto a JPA usam essas informações para validar os dados.

8.1 TRABALHE COM OS VALIDADORES E CRIE SUA PRÓPRIA VALIDAÇÃO

Assim como a JPA, a Bean Validation é uma especificação que define uma série de anotações para usarmos em nossas classes. Porém, em vez de mapeamento objeto relacional, definimos no nosso modelo as regras de validação de dados. Por exemplo, na entidade `Automovel` podemos especificar as seguintes restrições.

```
@Entity
public class Automovel {

    @Min(1900)
    @MaxAnoAtualMais(message="O máximo do ano de fabricação é {0}")
    private Integer anoFabricacao;

    @Min(1900)
    @MaxAnoAtualMais(value=1, message="O máximo do ano do modelo é {0}")
    private Integer anoModelo;

    @NotNull
    private Float preco;
}
```

`@Min` e `@NotNull` são validadores nativos da Bean Validation. Mas para considerar dinamicamente o ano atual, criamos um validador customizado `@MaxAnoAtualMais`, no qual passamos a informação de até quantos anos à frente do atual é considerado um valor aceito.

Para implementar essa validação, precisamos da definição da anotação `MaxAnoAtualMais`:

```
@Documented
@Constraint(validatedBy = MaxAnoAtualMaisValidator.class)
@Target( { METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
public @interface MaxAnoAtualMais {

    String message() default "O valor máximo para esse campo é {0}";

    int value() default 0;
}
```

```
Class<?>[] groups() default {};  
Class<? extends Payload>[] payload() default {};  
}
```

E também a classe onde estará toda a regra de validação:

```
public class MaxAnoAtualMaisValidator implements  
    ConstraintValidator<MaxAnoAtualMais, Integer> {  
  
    private int maxValue;  
    @Override  
    public void initialize(MaxAnoAtualMais annotation) {  
        int anosAdicionais = annotation.value();  
        int anoAtual = Calendar.getInstance().get(Calendar.YEAR);  
        maxValue = anoAtual + anosAdicionais;  
    }  
  
    @Override  
    public boolean isValid(Integer value,  
        ConstraintValidatorContext context) {  
        if(value > maxValue){  
            String template = context.  
                getDefaultConstraintMessageTemplate();  
  
            String message = MessageFormat.format(template, maxValue);  
  
            context.buildConstraintViolationWithTemplate(message)  
                .addConstraintViolation()  
                .disableDefaultConstraintViolation();  
  
            return false;  
        }  
        return true;  
    }  
}
```

No método `initialize` do validador, nós recebemos a anotação que define a validação. No nosso exemplo, temos duas ocorrências da anotação `@MaxAnoAtualMais`, uma com zero e outra com um ano adicional. Recuperamos

esse valor e somamos com o ano atual, assim sabemos qual é o máximo que o validador irá permitir.

No método `isValid`, recebemos o valor informado pelo usuário e retornamos `true` se o valor for válido ou `false` caso contrário. O restante do corpo do método foi usado apenas para montarmos uma mensagem mais completa para o usuário.

Recuperamos a mensagem de erro configurada na anotação via `context.getDefaultConstraintMessageTemplate()`. No caso da validação do ano de fabricação, o valor de `template` é "O Valor máximo do ano de fabricação é {0}". Com isso em mãos, geramos a mensagem usando a classe `MessageFormat` que troca `{0}` pelo valor de `maxValue`. Com a mensagem pronta, nós reconfiguramos a mensagem de erro com o restante do código.

8.2 ORGANIZE GRUPOS DE VALIDAÇÃO

Além de especificarmos os validadores para cada propriedade da nossa entidade, podemos também agrupá-los. Para um cadastro muito longo, é possível ter um grupo que represente a validação mínima para se salvar o registro como rascunho, e um grupo mais completo que represente a validação como um todo. Um exemplo disso é o e-mail, podemos salvar um e-mail como rascunho sem ter um destinatário, mas para enviá-lo o destinatário é obrigatório.

Se não especificarmos nada, por padrão, todas as validações ficam em um grupo chamado `Default`. E também, sempre antes de persistir, a JPA valida tudo que estiver associado a esse grupo. Com isso, todos os validadores da nossa entidade, mesmo que não obrigatórios para salvar a entidade como rascunho, serão disparados no momento em que tentarmos persistir esse objeto.

Para especificar um grupo, usamos uma classe qualquer, geralmente criada apenas para representá-lo.

```
package facesmotors.validation.groups;
public interface ValidacaoMinima { }
```

E agora especificaremos o grupo `ValidacaoMinima` na validação da propriedade `preco`. Isso significa que mesmo quando salvarmos um objeto em modo "rascunho", ou seja, em uma versão incompleta, pelo menos as propriedades que estiverem nesse grupo deverão ser validadas.


```
@Entity
public class Automovel {

    @Min(1900)
    @MaxAnoAtualMais(message="O máximo do ano de fabricação é {0}")
    private Integer anoFabricacao;

    @Min(1900)
    @MaxAnoAtualMais(value=1, message="O máximo do ano do modelo é {0}")
    private Integer anoModelo;

    @NotNull(groups={ValidacaoMinima.class, Default.class})
    private Float preco;

}
```

8.3 A INTEGRAÇÃO ENTRE BEAN VALIDATION E JPA

Integrar JPA e Bean Validation requer esforço quase zero. Basta que haja um *Validation Provider*, que é uma implementação de Bean Validation no ambiente, que a JPA automaticamente passará a usá-la.

No entanto podemos especificar uma configuração para desabilitar a integração entre JPA e Bean Validation:

```
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
    <validation-mode>NONE</validation-mode>
    <properties>
        ...
    </properties>
</persistence-unit>
```

Na tag `validation-mode`, os valores possíveis são:

- **NONE**: desabilita a integração entre as especificações;
- **CALLBACK**: explicitamente liga a integração, lançando uma exceção caso um provedor não esteja disponível. Pode ser interessante para assegurar que a aplicação não irá executar sem as validações previstas;
- **AUTO**: habilita a validação se o *provider* for encontrado.

Como vimos na seção anterior, a JPA por padrão valida o grupo `Default` ao inserir e ao realizar a atualização dos registros. Podemos, porém, alterar esse comportamento para validar apenas o grupo `ValidacaoMinima`. Vamos adicionar ao arquivo `persistence.xml` as seguintes propriedades:

```
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
  <validation-mode>AUTO</validation-mode>
  <properties>
    ...
    <property name="javax.persistence.validation.group.pre-persist"
      value="facesmotors.validation.groups.ValidacaoMinima" />
    <property name="javax.persistence.validation.group.pre-update"
      value="facesmotors.validation.groups.ValidacaoMinima" />
  </properties>
</persistence-unit>
```

Além das duas propriedades, ainda temos a `javax.persistence.validation.group.pre-remove` que podemos usar para especificar o grupo padrão a ser considerado na exclusão das informações. Em cada propriedade dessa, podemos passar mais de um grupo, usando a vírgula como separador do nome das classes.

8.4 A INTEGRAÇÃO ENTRE BEAN VALIDATION E JSF

A integração *Bean Validation* e JSF também é automática desde que uma implementação esteja disponível no ambiente da aplicação. Temos apenas a opção de desabilitar essa integração por meio de uma simples configuração no `web.xml`

```
<context-param>
  <param-name>javax.faces.validator.DISABLE_BEAN_VALIDATOR</param-name>
  <param-value>true</param-value>
</context-param>
```

Desde que não desabilitemos a integração, não precisamos fazer nada para usar as anotações como validadores do JSF. De toda forma, temos a tag `f:validateBean` que, apesar de parecer, não serve para habilitar a validação do JSF via Bean Validation. Usamos essa tag para desabilitar a validação ou então especificar um grupo diferente de validação.

```

<f:validateBean
    validationGroups="facesmotors.validation.groups.ValidacaoMinima">

    <h:panelGrid columns="2">
        Ano de Fabricação: <h:inputText value="#{auto.anoFabricacao}" />
        Ano do Modelo: <h:inputText value="#{auto.anoModelo}" />
        Preço: <h:inputText value="#{auto.preco}" />
        Kilometragem: <h:inputText value="#{auto.kilometragem}" />
        Observações: <h:inputTextarea value="#{auto.observacoes}" />
    </h:panelGrid>

</f:validateBean>

```

Nesse exemplo, especificamos que todos os componentes dentro da tag `f:validateBean` terão validados apenas o grupo `ValidacaoMinima`.

Em relação ao JSF temos mais um detalhe que envolve a validação de campos cujos valores são do tipo `String`, como por exemplo a propriedade `observacoes`. Por padrão, quando desenvolvemos para a web com Java, todo *input* cujo valor seja uma `String` não retorna valor `null`, e sim uma `String` em branco. Porém, a especificação *Bean Validation* não possui um validador que verifica se o valor é diferente de “em branco”.

```

@Entity
public class Automovel {
    ...
    @NotNull
    private String observacoes;
    ...
}

```

Em vez de criarmos um novo validador customizado ou usar algum específico da implementação de *Bean Validation*, que existe no Hibernate Validator - a anotação `NotEmpty` -, vamos somente fazer uma configuração no `web.xml` para que `String` em branco sejam submetidas com valor `null`.

```

<context-param>
    <param-name>
        javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
    </param-name>
    <param-value>true</param-value>
</context-param>

```

Agora o validador `@NotNull` irá barrar `Strings` que não foram preenchidas pelo usuário e que antes eram aceitas por conterem uma `String` vazia.

Parte III

Desenvolvendo a fluência

Após conhecer o funcionamento das ferramentas com que temos trabalhado, é hora de desenvolver nossa fluência. Nessa parte final, vamos refinar nossa aplicação e explorar conceitos e técnicas importantes de componentização e performance.

CAPÍTULO 9

Enriquecendo nossa aplicação JSF

Na primeira parte desse livro nós vimos como construir uma simples aplicação integrando JSF e JPA. Já na segunda parte vimos mais sobre como cada framework funciona. Agora iremos enriquecer a aplicação criada na primeira parte usando o que aprendemos na segunda e acrescentando novos elementos a ela.

Continuaremos usando o modelo apresentado no capítulo 6, e você pode lembrá-lo na figura 9.1.

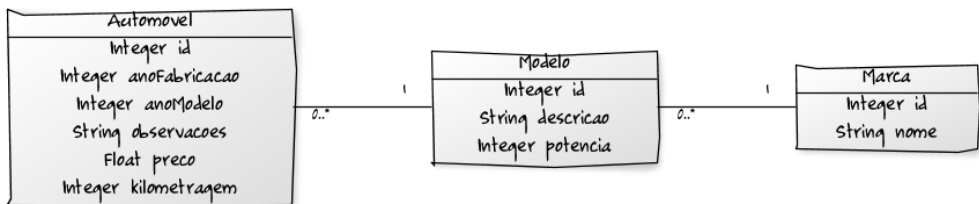
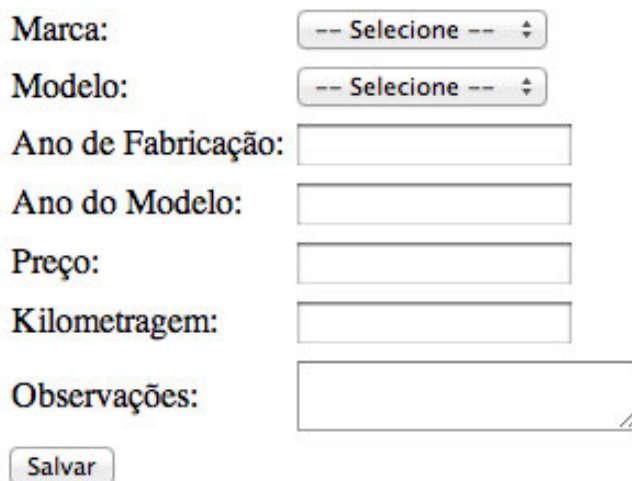


Figura 9.1: Modelo de classes usado nesse capítulo

A partir de agora, usaremos alguns componentes da biblioteca *Primefaces*, que você encontra em <http://primefaces.org/>, para apresentar cenários reais, onde poderíamos ficar limitados se utilizássemos os componentes do JSF.

9.1 COMBOS EM CASCATA E COMMANDS EM DATATABLE

Você deve ter notado que citamos combos em cascata em outras partes deste livro, mas o que faz desse exemplo tão importante? Primeiramente, vamos ver como ficou seu código, e a partir dele discutiremos as diferenças que teríamos no comportamento da aplicação dependendo do escopo que escolhêssemos.



O formulário apresenta os seguintes campos:

- Marca:** Dropdown menu com o texto "-- Selecione --" e uma seta para baixo.
- Modelo:** Dropdown menu com o texto "-- Selecione --" e uma seta para baixo.
- Ano de Fabricação:** Campo de texto.
- Ano do Modelo:** Campo de texto.
- Preço:** Campo de texto.
- Kilometragem:** Campo de texto.
- Observações:** Campo de texto maior, com uma seta para cima no canto inferior direito.
- Salvar:** Botão de submissão.

Figura 9.2: Tela de cadastro de Automovel com combos interdependentes

Primeiramente vejamos como ficou a nova versão da entidade `Automovel`.

```
@Entity
public class Automovel {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    private Modelo modelo;
```



```

    private Integer anoFabricacao;
    private Integer anoModelo;
    private Double preco;
    private Double kilometragem;
    private String observacoes;
}

```

A classe `AutomovelBean`, teve como alteração apenas o escopo, e a inclusão de uma propriedade do tipo `Marca` que guarda a escolha do usuário que servir de filtro para os modelos. Como podemos ver, na entidade `Automovel` guardamos apenas o `Modelo`, e não a `Marca`. Isso porque a partir do `Modelo` acessamos a `Marca`.

```

@ManagedBean
@ViewScoped
public class AutomovelBean {

    private Automovel automovel = new Automovel();
    private List<Automovel> automoveis;
    private Marca marca; // utilitário para buscar os modelos

    // getters, setters e método salvar
}

```

Por mais que nosso objeto `Automovel` esteja mais rico, o processo de manipulá-lo não muda: ponto para a Orientação a Objetos. A seguir veremos o código da tela.

```

<h:form>
    <h:messages />

    <h:panelGrid columns="2">
        Marca:
        <h:selectOneMenu label="marca" value="#{automovelBean.marca}"
            required="true" converter="entityConverter">

            <f:selectItem itemLabel="-- Seleccione --"
                noSelectionOption="true" />

            <f:selectItems value="#{marcaBean.marcas}" var="marca"
                itemValue="#{marca}" itemLabel="#{marca.nome}" />
        <f:ajax render="selectModelo" />
    </h:panelGrid>
</h:form>

```

```

</h:selectOneMenu>

Modelo:
<h:selectOneMenu id="selectModelo" label="modelo"
    value="#{automovelBean.automovel.modelo}" required="true"
    converter="entityConverter">

    <f:selectItem itemLabel="-- Selecione --"
        noSelectionOption="true" />
    <f:selectItems value="#{automovelBean.marca.modelos}"
        var="modelo" itemValue="#{modelo}"
        itemLabel="#{modelo.descricao}" />
</h:selectOneMenu>

Ano de Fabricação:
<h:inputText
    value="#{automovelBean.automovel.anoFabricacao}" />

Ano do Modelo:
<h:inputText
    value="#{automovelBean.automovel.anoModelo}" />

Preço:
<h:inputText
    value="#{automovelBean.automovel.preco}" />

Kilometragem:
<h:inputText
    value="#{automovelBean.automovel.kilometragem}" />

Observações:
<h:inputTextarea
    value="#{automovelBean.automovel.observacoes}" />
</h:panelGrid>

<h:commandButton value="Salvar"
    action="#{automovelBean.salvar(auto)}" />
</h:form>

```

Nesse código, declaramos dois *combos*, um para as marcas e o outro para os modelos. Em ambos os casos, temos um `selectItem` com a *label* “-- Selecione

--” apenas para essa mensagem vir na primeira linha do combo. Note a presença da propriedade `noSelectionOption="true"` indicando que essa opção não é selecionável. Ou seja, caso o usuário submeta o formulário com essa opção, será considerado um valor `null`.

No combo de marcas, usamos um outro *Managed Bean* de apoio para nos devolver a lista de `Marcas`. O método `getMarcas` do `MarcaBean`, que é chamado por meio da expressão `#{marcaBean.marcas}`, simplesmente devolve uma lista de `Marcas` que estiverem cadastradas. No restante o formato desse combo segue o mesmo padrão que foi apresentado no capítulo 7, tanto no seu funcionamento quanto no uso do conversor genérico de entidades. A diferença recai sobre o componente `f:ajax`. Por hora basta sabermos que ele mudará o valor do combo de marcas e irá recarregar o componente com id `selectModelo`, que no caso é o combo de `Modelos`. Na seção 9.4 entenderemos como trabalhar com requisições AJAX no JSF.

Analizando o combo de modelos, percebemos que precisamos de um escopo que mantenha o `automovelBean` entre as requisições. Isso se deve ao fato de os elementos do combo de modelos dependerem diretamente do valor do combo de marcas. Veja como o `value` do combo de marcas, `#{automovelBean.marca}`, serve de entrada para o `value` do `f:selectItems` do combo de modelos: `#{automovelBean.marca.modelos}`.

Considerando que o `AutomovelBean` estivesse com escopo *request*, a cada requisição o JSF se esqueceria desse *bean*, então a cada uma o JSF acharia que é a primeira requisição que estaria sendo enviada, o que é errado.

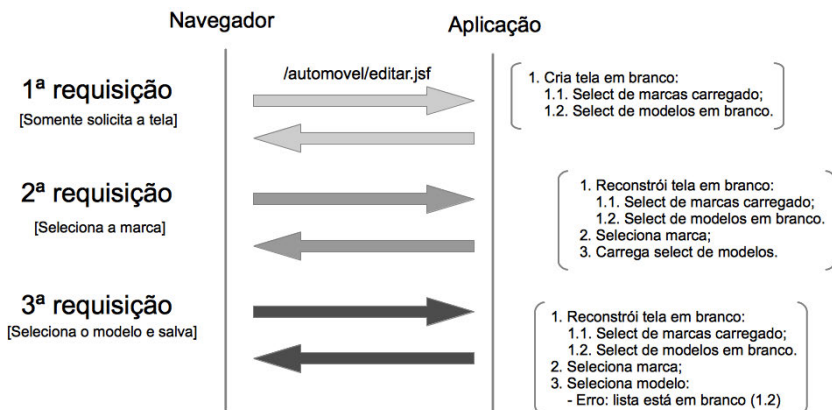


Figura 9.3: Sequência de combo em cascata que termina em erro

- 1) O usuário solicita a tela e o JSF cria a árvore de componentes que possui o combo de marcas preenchido e devolve para o usuário. Mas o combo de modelos não tem nenhuma opção, afinal, a propriedade `#{automovelBean.marca}` é `null`;
- 2) O usuário seleciona a marca e via AJAX é feita uma solicitação para o carregamento dos modelos. O JSF recria a árvore de componentes e, como o usuário enviou uma marca válida, ou seja, que estava nas opções do combo, então o JSF aceita a solicitação e devolve o combo de modelos preenchido;
- 3) O usuário seleciona o modelo e preenche os demais campos. Considerando que todos os outros campos preenchidos sejam válidos, o problema recai sobre o valor informado no combo de modelos. Como nosso bean está no escopo `request`, o JSF vai se comportar da mesma forma que na segunda requisição.

Como o `#{automovelBean}` que guardava a marca selecionada foi removido da memória devido a seu escopo ter acabado, um novo `AutomovelBean` sem uma marca selecionada é criado. O JSF considera que como a marca não foi selecionada, não teria como o combo de modelos possuir qualquer opção além da “-- Selecione --”, assim sendo, o valor é marcado como inválido.

Apesar desse cenário ser facilmente resolvido no JSF com o uso `@ViewScoped`, muitos projetos ainda utilizam JSF 1.X, em que ele não existe. E mesmo quem começar hoje com JSF e nunca venha a trabalhar com uma versão mais antiga do

framework vai precisar decidir qual escopo utilizar em cada *Managed Bean*. A indicação é sempre usar o menor escopo possível, que, dentre as opções que temos, é a *request*. Então mesmo em um projeto novo teremos que saber quando usar o escopo *view* em vez de *request*.

9.2 MOSTRE MENSAGENS COM H:MESSAGE E H:MESSAGES

Na seção 7.38 criamos validadores customizados, e nos exemplos apresentados nós geramos uma instância de `FacesMessage` e lançamos uma exceção com essa mensagem dentro. Em outros pontos da aplicação podemos criar mensagens, por exemplo, para informar que a operação foi efetuada com sucesso. Em todos esses casos, o JSF precisa de um componente que as exiba para o usuário. Caso não exista nenhum componente capaz de exibir essa mensagem, o JSF a coloca no console juntamente com o aviso que a tela não a está exibindo, para que o desenvolvedor possa perceber e corrigir o erro.

Percebemos, no entanto, que em vez de um, o JSF oferece dois componentes para o mesmo fim. O código a seguir servirá de base para os dois componentes de mensagem.

```
<h:form id="formAutomovel">
    Todas as mensagens: <br/>
    <h:messages/>

    <hr/>

    Apenas mensagens globais: <br/>
    <h:messages globalOnly="true"/>

    <hr/>

    Ano de Fabricação:
    <h:inputText value="#{automovelBean.automovel.anoFabricacao}">
        <f:validateLongRange minimum="1950" maximum="2012"/>
    </h:inputText>

    Ano do Modelo:
    <h:inputText id="anoModelo"
        value="#{automovelBean.automovel.anoModelo}">
        <f:validateLongRange minimum="1950" maximum="2013"/>
    </h:inputText>
```

```

</h:inputText>
<h:message for="anoModelo"/>

Preço:
<h:inputText value="#{automovelBean.automovel.preco}" />

Kilometragem:
<h:inputText value="#{automovelBean.automovel.kilometragem}" />

Observações:
<h:inputTextarea value="#{automovelBean.automovel.observacoes}" />
</h:form>

```

O `h:message` possui a propriedade `for` que o liga ao `id` de algum componente que possa gerar uma mensagem - no exemplo é o componente `anoModelo`. Dessa maneira, qualquer mensagem, seja de conversão, validação, ou mesmo gerada pela aplicação, que seja específica para esse `h:inputText`, será exibida no `h:message`.

Já o componente `h:messages` não tem vínculo com nenhum componente específico. Ele exibe todas as mensagens geradas para qualquer componente, ou mesmo mensagens globais, que não são endereçadas para um componente específico.

Utilizamos logo no começo um `h:messages` exibindo qualquer mensagem gerada na última requisição do usuário, mesmo que ela também seja exibida por um componente `h:message` específico. Isso pode causar um inconveniente, porque a mesma mensagem aparece duas vezes. Para resolver esse problema existe a propriedade `globalOnly` do `h:messages`, indicando que apenas as mensagens globais serão exibidas.

Diferenciando mensagens globais de mensagens específicas

Uma mensagem global é aquela que não é gerada para um componente específico. Por exemplo, um erro de conversão ou validação está relacionado com um componente, então não é uma mensagem global. Agora uma mensagem de sucesso ou falha ao salvar um `Automovel` no banco de dados não costuma ser relacionada com o botão que foi acionado para efetuar a operação. Geralmente essa mensagem não fica vinculada a nenhum componente, sendo, portanto, uma mensagem global.

```

FacesMessage msg = new FacesMessage("Automovel salvo com sucesso!");
FacesContext.getCurrentInstance().addMessage(null, msg);

```

Nesse código temos a criação de uma mensagem global, já que, ao adicionar a mensagem ao contexto to JSF através de `FacesContext.getCurrentInstance`, passamos `null` para o primeiro parâmetro, no qual iria o `id` do componente a quem essa mensagem se destina.

Mas vamos supor que vamos criar uma mensagem para o componente com `id` “anoModelo”.

```
FacesMessage msg = new FacesMessage("Ano modelo informado é inválido");
FacesContext.getCurrentInstance()
    .addMessage("formAutomovel:anoModelo", msg);
```

Um detalhe incômodo que notamos aqui é que, quando referenciamos um objeto dentro do XHTML, na maioria das vezes basta colocarmos o `id` dele. Porém, quando referenciamos um objeto dentro do nosso código Java, geralmente temos que referenciar o `clientId`. O `clientId` é o `id` que vai para o HTML gerado, que é o que o *client*, ou seja o navegador, recebe. Por isso o nome `clientId`.

Já sabemos como criar uma mensagem global e uma mensagem específica para um componente, mas como apresentar isso da melhor forma para o usuário? Um bom ponto de partida é definirmos se teremos componentes `h:message` dedicados para cada componente, ou pelo menos para aqueles que sabemos que devem gerar mensagens, ou ainda se teremos um local na página que mostrará todas as mensagens da tela.

Para ilustrar melhor o problema, vamos executar a tela apresentada no início da seção.

Todas as mensagens:

- `j_idt6:j_idt14`: Erro de validação: o atributo especificado não está entre os valores esperados de 1.950 e 2.012.
- `j_idt6:anoModelo`: Erro de validação: o atributo especificado não está entre os valores esperados de 1.950 e 2.013.

Apenas mensagens globais:

Ano de Fabricação:	<input type="text" value="2014"/>	
Ano do Modelo:	<input type="text" value="2015"/>	j_idt6:anoModelo: Erro de validação: o atributo especificado não está entre os valores esperados de 1.950 e 2.013.
Preço:	<input type="text"/>	
Kilometragem:	<input type="text"/>	
Observações:	<input type="text"/>	
<input type="button" value="Salvar"/>		

Figura 9.4: Erros de validação

Analisando o resultado, percebemos que tanto o campo “ano de fabricação”

quanto o “ano modelo” apresentaram erro de validação. Como o “ano modelo” tem um componente específico para ele, a mensagem saiu nos dois componentes. Já o componente `h:messages` que mostra apenas mensagens globais não mostrou mensagem alguma, porque nosso método de negócio nem chegou a executar.

Colocando labels e mensagens customizadas para erros de conversão e validação

Certamente um “detalhe” não passou despercebido, que é o nome estranho dado ao componente “ano de fabricação”. Isso acontece porque, diferente do “ano modelo”, não demos um `id` para o componente. Atribuir `ids` é muito importante quando formos trabalhar com AJAX, mas ainda não é o mais amigável para mensagens do usuário, apesar de ser melhor mostrar um `id` atribuído por nós do que um gerado pelo JSF.

Para resolver esse problema, temos algumas alternativas. A mais simples e direta é utilizar a propriedade `label` disponível em qualquer componente de entrada do JSF. Com isso, em vez de usar o `id`, o JSF usará a `label` para identificar o componente quando acontecer um erro.

```
<h:form id="formAutomovel" prependId="false">

    ...
    Ano de Fabricação:
    <h:inputText label="Ano de fabricação"
        value="#{automovelBean.automovel.anoFabricacao}">
        <f:validateLongRange minimum="1950" maximum="2012"/>
    </h:inputText>

    Ano do Modelo:
    <h:inputText id="anoModelo" label="Ano do modelo"
        value="#{automovelBean.automovel.anoModelo}">
        <f:validateLongRange minimum="1950" maximum="2013"/>
    </h:inputText>
    <h:message for="anoModelo"/>

    Preço:
    <h:inputText label="Preço"
        value="#{automovelBean.automovel.preco}" />

    Kilometragem:
```



```

<h:inputText label="Kilometragem"
              value="#{automovelBean.automovel.kilometragem}" />

Observações:
<h:inputTextarea label="Observações"
                  value="#{automovelBean.automovel.observacoes}" />
</h:form>

```

Agora, executando novamente a tela e forçando o mesmo erro teríamos o resultado a seguir.

Todas as mensagens:

- Ano de fabricação: Erro de validação: o atributo especificado não está entre os valores esperados de 1.950 e 2.012.
- Ano do modelo: Erro de validação: o atributo especificado não está entre os valores esperados de 1.950 e 2.013.

Apenas mensagens globais:

Figura 9.5: Erros de validação usando labels

A vantagem de usarmos `label` é que ela funciona tanto para erros de validação quanto de conversão. Dentro de erros de validação, podemos ter um valor fora do intervalo válido ou não informar um campo requerido, por exemplo. Mas se o que desejarmos for colocar uma mensagem específica nossa, temos algumas formas de o fazer. Podemos trocar a mensagem padrão que o JSF usa para as mensagens, que já tem suporte a português do Brasil, ou então especificar uma mensagem diferente somente para um componente.

Abordaremos agora a segunda opção, que é especificar uma mensagem exata para o que pode acontecer com aquele componente. Para isso, precisamos basicamente de 3 atributos:

- `converterMessage`: no qual podemos especificar para um determinado componente que mensagem queremos que apareça caso ocorra um erro de conversão.

- `requiredMessage`: no qual podemos especificar para um determinado componente que mensagem queremos que apareça quando o campo é obrigatório mas não foi informado.
- `validatorMessage`: no qual podemos especificar para um determinado componente que mensagem queremos que apareça caso ocorra um erro de validação.

Fazendo um apanhado geral sobre como o JSF monta a mensagem para o usuário, temos a seguinte sequência:

- 1) O JSF verifica se existe uma mensagem específica para o erro que está sendo gerado: conversão, campo requerido ou validação;
- 2) Não havendo uma mensagem específica, o JSF gera a mensagem padrão identificando o componente através da sua propriedade `label`;
- 3) Caso não tenha um `label` especificado para o componente, o JSF usa seu `id`, que pode ter sido especificado pelo desenvolvedor, ou ainda gerado automaticamente pelo JSF. Nesse último caso, a mensagem dificilmente ajudará o usuário do sistema.

Mas e se quisermos mudar as mensagens padrões do JSF?

9.3 INTERNACIONALIZANDO NOSSA APLICAÇÃO

A internacionalização de uma aplicação consiste em deixar seus textos de acordo com o idioma do usuário. O JSF em si já é dessa forma, tendo inclusive suporte a português do Brasil. Isso significa que todas as mensagens do JSF estão em arquivo de propriedades, o que nos dá uma opção a mais de customizá-las.

Por exemplo, para mudar a mensagem de campo requerido do JSF precisaríamos criar um arquivo chamado `Messages_pt_BR.properties` dentro de um pacote chamado `javax.faces` e nesse arquivo colocar um valor para a chave `javax.faces.component.UIInput.REQUIRED`.

```
javax.faces.component.UIInput.REQUIRED = Campo obrigatório. Informe-o!
```

Com isso, todas as mensagens de campo requerido passariam a usar o texto que nós informamos, e não o texto padrão do JSF.

Podemos também internacionalizar as mensagens que usamos tanto dentro do nosso código Java, como do `xhtml`. Primeiro, nós os declaramos um novo arquivo de mensagens no `faces-config.xml` e o usamos em toda a aplicação.

```
<application>
    ...
    <resource-bundle>
        <base-name>facesmotors.messages.Mensagens</base-name>
        <var>msgs</var>
    </resource-bundle>
</application>
```

E agora, considere a tela de cadastro do automóvel, que possui diversas mensagens espalhadas por ela:

Ano de Fabricação:

```
<h:inputText label="Ano de fabricação"
    value="#{automovelBean.automovel.anoFabricacao}"/>
```

Ano do Modelo:

```
<h:inputText label="Ano do modelo"
    value="#{automovelBean.automovel.anoModelo}"/>
```

Preço:

```
<h:inputText label="Preço"
    value="#{automovelBean.automovel.preco}" />
```

Uma das possibilidades de internacionalizar esse trecho de código seria conforme o código a seguir.

```
#{msgs.label_ano_fabricacao}:
```

```
<h:inputText label="#{msgs.label_ano_fabricacao}"
    value="#{automovelBean.automovel.anoFabricacao}"/>
```

```
#{msgs.label_ano_modelo}:
```

```
<h:inputText label="#{msgs.label_ano_modelo}"
    value="#{automovelBean.automovel.anoModelo}"/>
```

```
#{msgs.label_preco}:
```

```
<h:inputText label="#{msgs.label_preco}"
    value="#{automovelBean.automovel.preco}" />
```

Todos os textos que antes estavam literais dentro do `xhtml` foram extraídos para um arquivo chamado `Mensagens_pt_BR.properties`, que está dentro do pacote `facesmotors.messages`. O conteúdo desse arquivo é o seguinte:

```
label_ano_fabricacao = Ano de Fabricação
label_ano_modelo = Ano do Modelo
label_preco = Preço
label_km = Kilometragem
label_obs = Observações
```

Depois que registramos a variável `msgs` no `faces-config.xml`, podemos usá-la em qualquer lugar da aplicação. Além disso, podemos declarar várias ocorrências de `resource-bundle` dentro da tag `application`, permitindo que exista mais de um arquivo de mensagens.

É importante notarmos também que, apesar do nome completo do arquivo ser `facesmotors.messages.Mensagens_pt_BR.properties`, dentro do Java esse arquivo é tratado apenas como `facesmotors.messages.Mensagens`. A extensão é desconsiderada, e o sufixo `pt_BR` serve para indicar o idioma do arquivo.

Se nossa aplicação deve suportar mais de um idioma, versões alternativas do arquivo de mensagem devem ser criadas. Por exemplo, para criarmos uma versão desse mesmo arquivo em inglês, deveríamos dar-lhe o nome de `facesmotors.messages.Mensagens_en.properties`; e se fosse algo específico para o inglês americano, o arquivo deveria se chamar `facesmotors.messages.Mensagens_en_US.properties`.

Internacionalização além dos labels

Na seção 9.2, vimos como melhorar as mensagens apresentadas para o usuário. Vimos que uma forma de melhorar a indicação de qual o campo que apresentou algum erro é com o uso da propriedade `label` presente nos componentes de input em geral. Inclusive no exemplo que vimos de internacionalização essa propriedade estava sendo usada. Mas além da `label`, vimos que temos propriedades mais específicas, nas quais podemos colocar mensagens para serem usadas em casos de erro de validação, conversão ou de campo requerido.

```
{msgs.label_preco}:
<h:inputText label="{msgs.label_preco}"
    value="{auto.preco}"
    converterMessage="0 preço deve ser numérico" />
```

Observando esse exemplo, nem precisava ser dito que podemos também passar esse tipo de mensagem para um arquivo de propriedades. O JSF aceita *Expression Language* em praticamente qualquer propriedade de seus componentes, e o `converterMessage`, `requiredMessage` e `validatorMessage` não são diferentes. Como o mecanismo de carregamento de *bundles* nos disponibiliza uma variável acessível via *Expression Language*, então podemos usá-la e deixar nosso exemplo da seguinte forma:

```
#{msgs.label_preco}:
<h:inputText label="#{msgs.label_preco}"
    value="#{auto.preco}"
    converterMessage="#{msgs.converter_preco}" />
```

E agora, temos a atualização do arquivo `.properties`.

```
#===== labels do formulário de automovel =====
label_ano_fabricacao = Ano de Fabricação
label_ano_modelo = Ano do Modelo
label_preco = Preço
label_km = Kilometragem
label_obs = Observações

#===== mensagens customizadas do formulário de automovel =====
converter_preco = O preço do automóvel deve ser numérico (ex: 50000)
```

9.4 JSF E AJAX

O suporte a AJAX do JSF é algo bem interessante. No exemplo a seguir veremos como trabalhar com combos em cascata, mas o princípio é o mesmo do que se estivéssemos atualizando qualquer outro tipo de componente.

Marca:

```
<h:selectOneMenu label="marca" value="#{automovelBean.marca}"
    required="true" converter="entityConverter">
    <f:selectItem itemLabel="-- Selecione --"
        noSelectionOption="true" />
    <f:selectItems value="#{marcaBean.marcas}" var="marca"
        itemValue="#{marca}" itemLabel="#{marca.nome}" />
    <f:ajax render="selectModelo" />
</h:selectOneMenu>
```

Modelo:

```
<h:selectOneMenu id="selectModelo" label="modelo"
    value="#{automovelBean.automovel.modelo}" required="true"
    converter="entityConverter">
    <f:selectItem itemLabel="-- Selecione --"
        noSelectionOption="true" />
    <f:selectItems value="#{automovelBean.marca.modelos}"
        var="modelo" itemValue="#{modelo}"
        itemLabel="#{modelo.descricao}" />
</h:selectOneMenu>

...
```

Vamos analisar o que desejamos que aconteça nesse exemplo. Temos um combo que lista todas as Marcas de automóveis e, abaixo, outro que lista todos os Modelos da Marca selecionada no combo de cima. Mudando o de cima, o de baixo é recarregado. Apenas para recordarmos, nosso modelo está como na seguinte listagem:

```
public class Marca {
    ...
    private List<Modelo> modelos;
    ...
}

public class Modelo {
    ...
    private Marca montadora;
    ...
}
```

Desconsideramos nessa listagem os mapeamentos da JPA e outras propriedades das classes. O ponto importante é que a `Marca` possui uma listagem de `Modelos`. Como o mapeamento objeto relacional já foi feito, não precisamos nos preocupar com como a listagem de `Modelo` será carregada; o que importa é que, uma vez que temos a `Marca` em mãos, basta acessarmos a propriedade `modelos` e os objetos estarão lá.

Observando o `xhtml`, vemos que o combo de `Modelo` simplesmente acessa a propriedade `modelos` da `Marca` selecionada. No *Managed Bean* `automovelBean`, não existe nenhum código extra para buscar, apenas uma propriedade do tipo `Marca` que está sendo alimentada pelo primeiro combo. Uma vez

que essa dependência está toda configurada, basta atualizarmos o combo inferior quando o de cima mudar, que naturalmente a listagem de `Modelos` será atualizada quando mudarmos a `Marca`.

Porém, para uma melhor experiência do usuário, em vez de submetermos a tela inteira toda vez que o combo de `Marca` mudar, nós simplesmente mandamos o JSF renderizar novamente o combo de baixo. Para fazer isso bastou incluir o comando:

```
<f:ajax render="selectModelo" />
```

Dessa forma, indicamos o `id` do componente que queremos atualizar.

Já vimos que não precisamos fazer busca alguma pelos objetos do combo de `Modelo`, e agora vemos que também não precisamos fazer nada para atualizar os itens do combo. Isso acontece dessa maneira porque o JSF conhece a tela, ele sabe onde o `f:selectItems` está buscando os elementos, então sabe renderizar novamente apenas o trecho da árvore de componentes que solicitamos - que é a subárvore abaixo do `selectModelo` que informamos na propriedade `render` do `f:ajax`.

Além da tag `f:ajax`, o suporte a AJAX do JSF tem uma pequena biblioteca *javascript* que recebe o HTML resultante dessa nova renderização e, a partir do `id` do elemento, via DOM (*Document Object Model*), substitui o conteúdo antigo pelo novo.

Podemos imaginar um funcionamento parecido, se tivéssemos numa tela um botão que incluísse um elemento a uma lista usada para alimentar um `h:dataTable`. Para exemplificar, vamos considerar uma inclusão em massa de `Marca`, que no nosso modelo tem apenas uma descrição.

```
...
<h:commandButton value="Adicionar Marca" action="#{marcaBean.addMarca}">
    <f:ajax render="tabelaMarcas"/>
</h:commandButton>
<h:dataTable id="tabelaMarcas" value="#{marcaBean.marcasParaIncluir}"
    var="marca">
    <h:column><h:inputText value="#{marca.nome}"/></h:column>
</h:dataTable>
...

@ManagedBean
@ViewScoped
public class MarcaBean {
    ...
}
```

```

private List<Marca> marcasParaIncluir = new ArrayList<>();

public void addMarca(){
    marcasParaIncluir.add(new Marca());
}
// getter para a propriedade marcasParaIncluir
...
}

```



Adicionar Marca

Audi

Ferrari

Mercedes

Posche

Figura 9.6: Criação de marca em lote

Novamente, precisamos apenas executar nossa ação, que aqui é a adição de um elemento em uma lista, e mandar o JSF renderizar novamente a parte da tela que queremos atualizar via AJAX. O resto o JSF faz.

Propriedades suportadas

Até agora vimos somente a propriedade `render`, mas a tag `f:ajax` possui outras propriedades importantes:

- **event:** É o evento DOM que queremos usar para disparar a ação AJAX. O valor padrão dessa propriedade é `click` para links e botões, e `change` para componentes de entrada de dados. Existe também o evento `blur`, quando um componente perde o foco.

Se quisermos mudar o evento padrão de diversos componentes, ao mesmo tempo em que deixamos todos eles com suporte a AJAX, podemos fazer o seguinte:

```

...
<f:ajax event="keyup">

```



```
Ano de Fabricação:
<h:inputText value="#{auto.anoFabricacao}"/>

Ano do Modelo:
<h:inputText value="#{auto.anoModelo}"/>
</f:ajax>
...
```

Nesse caso, a cada tecla que o usuário digitar em qualquer um dos `h:inputText` será feita uma requisição e o componente terá seu valor atualizado servidor, mas nenhum resultado será percebido na tela porque não mandamos atualizar nada com a propriedade `render`.

Além dos eventos DOM, temos dois eventos “artificiais” adicionados pelo JSF: o `action`, que pode ser usado em componentes de ação como `h:commandButton` e `h:commandLink` e o `valueChange` que pode ser usado em qualquer componente de entrada de dados.

- `render`: Essa é a propriedade mais comum de se usar quando trabalhamos com AJAX no JSF. Através dela, informamos a subárvore, ou subárvores que devem ser renderizadas novamente para atualizar a tela. Cada `id` ou constante informada na propriedade `render` dá origem a uma subárvore nova, contendo o componente informado e todos os componentes abaixo deste. Sendo assim, se tivermos um `h:panelGrid` ou `h:panelGroup` na lista a ser renderizada, não precisamos informar o `id` de nenhum componente que estiver dentro desses *containers*, porque todos já serão automaticamente atualizados.
- `execute`: Através dessa propriedade, informamos qual subárvore deve ser submetida ao servidor para que o JSF processe. Essa árvore não é a mesma que irá ser renderizada novamente pela propriedade `render`.
- `listener`: Assim como uma *action* do JSF pode ter *action listeners* associada, uma requisição AJAX também pode ter um *listener*. Para isso basta ligarmos a propriedade `listener` com um método com a seguinte assinatura:

```
public void <nome do método>(javax.faces.event.AjaxBehaviorEvent event)
throws javax.faces.event.AbortProcessingException.
```

No exemplo de edição em lote de `Marcas`, pegamos um objeto em branco, ou seja, logo após dar um `new` nele, e o adicionamos na tabela, que era editável. Assim,

o usuário podia modificar os valores diretamente de dentro da mesma. Agora vamos mudar um pouco o exemplo para fazer com que o preenchimento do objeto seja feito antes de adicionarmos na tabela. O resultado seria como o que podemos ver a seguir.

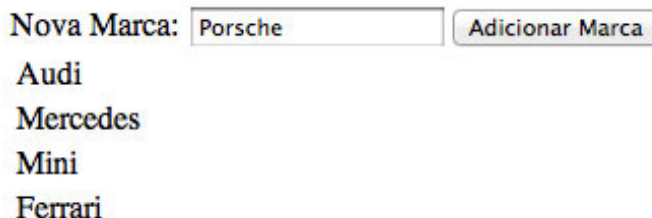
```
...
Nova Marca:
<h:inputText id="novaMarca"
             value="#{marcaBean.novaMarca.descricao}"/>

<h:commandButton value="Adicionar Marca"
                 action="#{marcaBean.addMarca}">
    <f:ajax execute="novaMarca" render="tabelaMarcas novaMarca"/>
</h:commandButton>

<h:dataTable id="tabelaMarcas"
             value="#{marcaBean.marcasParaIncluir}" var="marca">
    <h:column>#{marca.nome}</h:column>
</h:dataTable>
...

@ManagedBean
@ViewScoped
public class MarcaBean {
    ...
    private Marca novaMarca = new Marca();
    private List<Marca> marcasParaIncluir = new ArrayList<>();

    public void addMarca(){
        marcasParaIncluir.add(novaMarca);
        novaMarca = new Marca();
    }
    //getter e setters
    ...
}
```



Nova Marca:

Audi
Mercedes
Mini
Ferrari

Figura 9.7: Criação de marca em lote

A mudança no código não foi tão significativa. Agora temos um `input` antes do botão “Adicionar Marca”, enquanto na tabela temos somente `output`. A mudança mais importante para nós nesse momento é a inclusão da propriedade `execute`.

Para que possamos atualizar a tabela com o novo valor preenchido, precisamos que o JSF receba no servidor o conteúdo do `h:inputText` com `id="novaMarca"`. Por padrão, o componente que faz a requisição AJAX, que no nosso caso é o botão, só envia ele mesmo para o servidor. Dessa forma, sem especificarmos o `id` do `h:inputText` que tinha a informação necessária, o JSF até iria processar a requisição, mas não atualizaria o valor do `input` porque ele não teria sido submetido.

Além do `id` do `input`, no `execute` está implícito o envio de `@this`. Esse valor é uma constante que representa o próprio componente que está originando a requisição AJAX.

Percebemos que na propriedade `render`, informamos o `id` do `input` no qual é informado o nome da `Marca`. Isso é necessário para que o mesmo seja limpo automaticamente, refletindo o efeito do *Managed Bean*, onde criamos uma `Marca` nova.

Tanto a propriedade `execute` quanto a `render` aceitam as constantes:

- `@this`: usada para referenciar o próprio componente que está originando a requisição AJAX;
- `@form`: usada para referenciar todo o formulário que está em volta do componente que está originando a requisição AJAX;
- `@all`: usada para referenciar toda a view do JSF;
- `@none`: usada para informar que nenhuma árvore deve ser processada.

9.5 ORGANIZE O CÓDIGO DAS TELAS COM TEMPLATES

Uma das principais novidades do JSF 2 foi a incorporação do Facelets na especificação. Essa já era uma ferramenta indispensável mesmo quando era um framework separado, sendo usado tanto para aplicarmos *templates* na aplicação, quanto para construirmos componentes compondo outros componentes.

A utilização de *templates* facilita a padronização da aplicação, já que conseguimos deixar fixo tudo aquilo que raramente muda, como topo, rodapé e menu lateral das páginas, bem como informar somente a parte que muda, que geralmente é o meio da página. Esse *layout* que acabamos de descrever pode ser ilustrado pela figura 9.8.

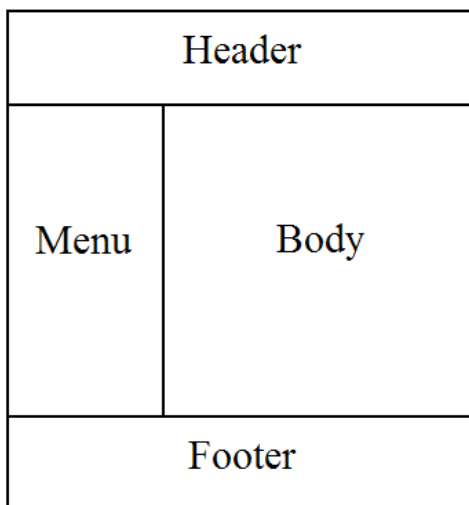


Figura 9.8: Exemplo de organização de um template

Defina o template

A definição do template é um trabalho bem parecido com o da construção de uma página comum. A diferença é que inserimos nessas páginas algumas marcações, indicando onde o conteúdo dinâmico será inserido. No código a seguir, temos um exemplo dessas definições.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
```

```
xmlns:f="http://java.sun.com/jsf/core"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>
  <title><ui:insert name="pageTitle">Faces Motors</ui:insert></title>
  <h:outputStylesheet library="css" name="template.css" />
</h:head>
<h:body>
  <div id="topo">
    <ui:insert name="topo">
      <ui:include src="/WEB-INF/comuns/topo.xhtml"/>
    </ui:insert>
  </div>
  <div id="menu">
    <ui:include src="/WEB-INF/comuns/menu.xhtml"/>
  </div>
  <div id="body">
    <ui:insert name="body"/>
  </div>
  <div id="rodape">
    <ui:include src="/WEB-INF/comuns/rodape.xhtml"/>
  </div>
</h:body>
</html>
```

No template apresentado, definimos quatro seções: `topo`, `menu`, `body` e `rodape`. A organização na tela será feita via CSS, mas para nós o que importa é como cada área dessa será preenchida. Na definição do template usamos basicamente duas tags: `ui:insert` e `ui:include`.

Abra espaço com o `ui:insert`

Usamos `ui:insert` quando queremos definir o ponto do template onde será inserido um conteúdo proveniente da página cliente, ou seja, da página que usa o template. Temos apenas a propriedade `name` para podermos referenciar essa área editável dentro da página cliente.

Como podemos observar, definimos as áreas `pageTitle` e `topo` já com um conteúdo padrão associado. Isso quer dizer que caso a página cliente não informe um valor para essas áreas, elas apresentarão os valores especificados no próprio *template*. Depois fizemos a definição do `body`, que é onde vai o corpo da página propriamente dito. Nesse caso não temos um conteúdo padrão.

Inclua conteúdo com o `ui:include`

A tag `ui:include` é usada quando queremos incluir o conteúdo de uma página dentro de outra. No nosso exemplo, usamos essa tag para definir o conteúdo padrão das seções do *template* que criamos.

A PÁGINA EM EXECUÇÃO NÃO É O TEMPLATE, E SIM A PÁGINA CLIENTE

Algo importante que temos que aprender sobre o uso de templates é que eles não existem por si só. Nunca teremos um *template* executando, afinal ele é uma página incompleta. Até mesmo por isso, é uma boa prática colocarmos os *templates* e as páginas que ele inclui dentro da pasta `WEB-INF`, para que o usuário não consiga acessá-las diretamente.

Se tentarmos escrever o nome da página sendo processada, não sairá o nome do *template*, e sim o nome da página cliente. Sendo assim, temos que tomar cuidado ao usarmos caminhos relativos dentro do *template*, porque esses caminhos serão relativos não a ele, e sim à página cliente. Por isso, no *template* apresentado usamos caminhos absolutos.

Componha os templates

Uma vez que temos o *template* definido, sobra para cada página da nossa aplicação usá-lo.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    Posso colocar qualquer coisa antes...
    <ui:composition template="/WEB-INF/comuns/template.xhtml">
        <ui:define name="body">
            Conteúdo da página
        </ui:define>
    </ui:composition>
    Ou depois da ui:composition, mas tudo será ignorado
</html>
```

A forma mais comum de definir uma página que usa um template é criá-la normalmente, usando a tag `html` e em seguida determinar a tag `ui:composition` que indica o template que está sendo usado. Apesar de ser uma opção válida, como está escrito no código anterior, tudo que for definido fora da tag `ui:composition` será ignorado. Na prática, a página começa e termina junto com a tag `ui:composition` e não com a tag `html` como geralmente acontece quando não usamos *template* algum.

Por esse motivo, uma forma diferente de definir a mesma página pode ser vista a seguir.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  template="/WEB-INF/comuns/template.xhtml">

  <ui:define name="body">
    Conteúdo da página
  </ui:define>
</ui:composition>
```

Agora já iniciamos nossa página com a tag `ui:composition`, eliminando assim a possibilidade de alguém definir por engano algum conteúdo que será ignorado, visto que a própria tag agora delimita o nosso arquivo XHTML. Em ambos os casos, o resultado da página cliente será o mesmo.

Defina o conteúdo com o `ui:define`

Na página cliente, usamos `ui:define` para definir o conteúdo de uma área criada com `ui:insert`.

Quando o usuário solicita a página `home.xhtml`, será processado o arquivo de mesmo nome, que usa o template. Essa página, por sua vez, define, via tag `ui:define`, conteúdos identificados por nomes.

Cuidados com páginas incluídas

Vimos no nosso template que boa parte do seu conteúdo foi definida usando inclusões. No entanto, o que acontece se simplesmente incluirmos uma página como a seguinte?

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:graphicImage library="images" name="logo.png"/>
    Desenvolvido pela Casa do Código

</html>

```

O resultado disso é que teremos o conteúdo total dessa página, que no caso é a `rodape.xhtml`, dentro da página que a incluiu. Porém, como podemos perceber, dentro da `rodape.xhtml` temos a tag `html`, o que resulta em uma tag `html` dentro da outra.

Para evitar isso, costumamos colocar as páginas que serão incluídas dentro de uma tag `ui:composition`, porque, como já vimos antes, tudo que fica fora dessa tag é ignorado, incluindo a tag `html`.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <ui:composition>
        <h:graphicImage library="images" name="logo.png"/>
        Desenvolvido pela Casa do Código
    </ui:composition>
</html>

```

Podemos utilizar a mesma abordagem de declarar um `ui:composition` nas páginas que serão incluídas, assim, apenas a parte dentro dessa tag será considerado para a inclusão.

A alternativa `ui:component`

Podemos usar essa tag nos mesmos casos em que usamos `ui:composition` sem um `template` associado. A diferença é que, em vez de simplesmente o conteúdo da tag ser copiado, como aconteceria no uso da `ui:composition`, utilizando `ui:component` o JSF irá criar na árvore de componentes um componente com todo o conteúdo da tag.

ui:insert anônima

A página que chamamos de `painel.xhtml` não deixa de ser um *template*, assim como vimos anteriormente ao definir o *layout* padrão da aplicação. Então, em ambos os casos, podemos usar tanto áreas nomeadas ou anônimas através do uso da tag `ui:insert`.

Quando não especificamos um `name` nessa tag, dizemos que ela é anônima, e para definir seu conteúdo não precisamos da tag `ui:define`. Basta inserir o conteúdo que quisermos dentro da tag `ui:composition` ou `ui:decorate` que ele será inserido no local do *template* onde estiver a tag `ui:insert` anônima.

Podemos ter diversas ocorrências de `ui:insert` em um mesmo *template*, mas somente uma pode ser anônima.

Passe parâmetros com ui:param

Podemos passar parâmetros para outra página, tanto usando `ui:composition`, `ui:decorate` ou `ui:include`. Em todos os casos, podemos passar objetos para a página que está sendo chamada, seja ela um *template* ou uma página que será incluída.

Na página que recebe o parâmetro não precisamos fazer nada, basta usá-lo a partir de seu nome.

9.6 COMPOSITE COMPONENTS: CRIANDO NOSSOS PRÓPRIOS COMPONENTES A PARTIR DE OUTROS

Uma das possibilidades mais interessantes que temos com o Facelets é a de criar componentes, compondo já existentes. Vamos analisar um exemplo bem simples, mas funcional: um componente que combine um `label`, um `h:inputText` e um `h:message` para esse input. Depois de analisar o exemplo e entender a ideia geral, veremos com detalhes como tudo funciona.

Código tradicional:

Ano de Fabricação:

```
<h:inputText id="anoFabricacao" label="Ano de fabricação"
  value="#{automovelBean.automovel.anoFabricacao}"/>
<h:message for="anoFabricacao"/>
```

Ano do Modelo:

```

<h:inputText id="anoModelo" label="Ano do modelo"
    value="#{automovelBean.automovel.anoModelo}"/>
<h:message for="anoModelo"/>

Preço:
<h:inputText id="preco" label="Preço"
    value="#{automovelBean.automovel.preco}" />
<h:message for="anoModelo"/>

Kilometragem:
<h:inputText id="kilometragem" label="Kilometragem"
    value="#{automovelBean.automovel.kilometragem}" />
<h:message for="kilometragem"/>
...

```

Não é difícil perceber quanta repetição de códigos temos nesse exemplo. Sempre temos o `label` aparecendo duas vezes, assim como temos sempre também o `h:messages`. Mesmo sem saber ainda como, não é difícil imaginar que seja possível algo menos repetitivo como o código a seguir.

```

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:fm="http://java.sun.com/jsf/composite/componentes">

    <fm:input label="Ano de Fabricação" id="anoFabricacao"
        value="#{automovelBean.automovel.anoFabricacao}"/>

    <fm:input label="Ano do Modelo" id="anoModelo"
        value="#{automovelBean.automovel.anoModelo}"/>

    <fm:input label="Preço" id="preco"
        value="#{automovelBean.automovel.preco}"/>

    <fm:input label="Kilometragem" id="kilometragem"
        value="#{automovelBean.automovel.kilometragem}"/>

</html>

```

Na seção 7.34 tivemos o primeiro contato com o suporte a recursos do JSF. O

mecanismo de *composite components* utiliza essa mesma estrutura. Dentro da pasta `resources` colocamos uma ou mais pastas, com os nomes que quisermos, para armazenar nossos componentes. Na nossa aplicação haverá uma pasta, chamada “componentes” como podemos ver na figura a seguir.

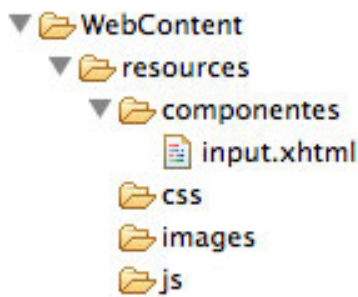


Figura 9.9: Estrutura de diretórios do componente input

Para utilizar os componentes customizados, temos a seguinte *uri* <http://java.sun.com/jsf/composite/< pasta com os componentes>>. Como nossa pasta se chama `componentes`, acabamos tendo a *uri* <http://java.sun.com/jsf/composite/componentes>.

Reparem que utilizamos o *namespace* `fm`, abreviação de faces-motors, apontando para a pasta “componentes”. A partir desse *namespace* temos uma tag para cada arquivo dentro da pasta. Não é preciso registrar nada em lugar nenhum, é tudo por convenção.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:cc="http://java.sun.com/jsf/composite"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <cc:interface>
    <cc:attribute name="id" required="true"/>
    <cc:attribute name="label" required="true"/>
    <cc:attribute name="value" required="true"/>
  </cc:interface>
```

```
</html>
```

Da mesma forma que temos o bloco `cc:interface` temos o `cc:implementation`, que é onde definimos o conteúdo do nosso componente.

```
...
<cc:implementation>
    #{cc.attrs.label}:
    <h:inputText id="#{cc.attrs.id}" label="#{cc.attrs.label}"
        value="#{cc.attrs.value}" />
    <h:message for="#{cc.attrs.id}" />
</cc:implementation>
...
```

A implementação é basicamente o mesmo código que tínhamos antes de começar a usar nosso próprio componente, com a diferença de que parametrizamos tudo que era variável. Na implementação do componente, devemos utilizar a expressão `#{cc.attrs}` para acessar os atributos do nosso componente, e então, a partir dos atributos, vamos acessando cada um deles conforme a necessidade.

Lidando com children

Analise o seguinte código:

```
Ano de Fabricação:
<h:inputText value="#{automovelBean.automovel.anoFabricacao}">
    <f:validateLongRange minimum="1950" maximum="2012"/>
</h:inputText>
```

Dentro do `h:inputText` temos um `f:validateLongRange`, ou seja, a *tag* de validação é filha do *input*. Então, se quisermos que nosso componente customizado aceite critérios de validação, como os componentes nativos, precisamos fazer com que ele aceite tags filhas.

A implementação disso é muito simples, basta usarmos `cc:insertChildren` onde quisermos que fique o conteúdo colocado como filho do nosso componente.

```
...
<cc:implementation>
    #{cc.attrs.label}:
    <cc:insertChildren />
```

```
<h:inputText id="#{cc.attrs.id}" label="#{cc.attrs.label}"
    value="#{cc.attrs.value}">
    <cc:insertChildren/>
</h:inputText>
<h:message for="#{cc.attrs.id}"/>
</cc:implementation>
...
```

A única diferença em relação à implementação anterior é onde puxamos o conteúdo que colocamos dentro do nosso componente na tela que o usar.

```
...
<fm:input label="Ano de Fabricação" id="anoFabricacao"
    value="#{automovelBean.automovel.anoFabricacao}">
    <f:validateLongRange minimum="1950" maximum="2012"/>
</fm:input>

<fm:input label="Ano do Modelo" id="anoModelo"
    value="#{automovelBean.automovel.anoModelo}">
    <f:validateLongRange minimum="1950" maximum="2013"/>
</fm:input>

<fm:input label="Preço" id="preco" value="#{auto.preco}"/>
...
```

Então agora podemos colocar tag de validação, ou conversão, ou mesmo `f:ajax` dentro do nosso componente. Enfim, agora ele suporta dentro qualquer outro elemento que um componente padrão JSF aceitaria.

Isso é muito importante quando trabalhamos com componentes customizados. Afinal, a principal desvantagem é quando eles limitam as possibilidades de uso. Adicionando suporte a *tags* filhas nós deixamos nosso componente mais versátil, diminuindo a necessidade de adaptações quando ele for reutilizado em novos cenários.

Lidando com facets

O que acontece quando temos mais de um conteúdo passível de ser inserido dentro da tag que acabamos de criar?

Por exemplo, considere um componente que é um painel customizado, e que podemos informar tanto o valor do cabeçalho quanto do rodapé desse painel. Parece simples, não? Seria apenas passarmos os respectivos valores via propriedades

do componente. Agora considere que, além de um texto, queiramos permitir a colocação de um ícone no cabeçalho e um botão de ação no rodapé como na imagem a seguir.



 Adicionar Automóvel	
Marca:	-- Selecione -- ▾
Modelo:	-- Selecione -- ▾
Ano de Fabricação:	<input type="text"/>
Ano do Modelo:	<input type="text"/>
Preço:	<input type="text"/>
Kilometragem:	<input type="text"/>
Observações:	<input type="text"/>
<input type="button" value="Salvar"/>	

Figura 9.10: Panel usando facet para montar cabeçalho com figura e rodapé com botão

Mas para enriquecer mais ainda as possibilidades, podemos em outro momento colocar um input no cabeçalho e um botão para filtrar o conteúdo do painel.

ID	Marca	Modelo
1	Volkswagen	Fusca
2	Volkswagen	Gol
3	Fiat	Palio
4	Porsche	Porsche 911 Turbo
5	Porsche	Panamera
6	Volkswagen	Polo
7	Fiat	Punto
8	Chevrolet	Camaro
9	Ford	Focus
10	Ford	Fiesta

Figura 9.11: Panel com `h:inputText` e `h:commandButton` no cabeçalho

Com isso percebemos que não dá pra ficar passando dezenas de propriedades para o componente no intuito de deixá-lo mais versátil. Nesse momento, vemos o porquê da existência da `tag f:facet`. Ela nos permite criar seções nomeadas dentro do nosso componente. Não é apenas uma propriedade, é uma área inteira que fica dentro do nosso componente, podendo conter diversos componentes dentro de si. Então, baseado no nome dessa área, conseguimos recuperar seu conteúdo.

```
<fm:painel colunas="2">
    <f:facet name="header">
        <h:graphicImage library="images" name="icon.png"/><br/>
        Adicionar Automóvel
    </f:facet>
    <f:facet name="footer">
        <h:commandButton value="Salvar"
            action="#{automovelBean.salvar(auto)}/>
    </f:facet>
    ...
    Ano de Fabricação:
    <h:inputText value="#{automovelBean.automovel.anoFabricacao}" />
```

```

Ano do Modelo:
<h:inputText value="#{automovelBean.automovel.anoModelo}" />

Preço:
<h:inputText value="#{automovelBean.automovel.preco}" />

Kilometragem:
<h:inputText value="#{automovelBean.automovel.kilometragem}" />

Observações:
<h:inputTextarea value="#{automovelBean.automovel.observacoes}" />
</fm:painel>

```

Para lidar com *facets* temos duas opções, o `cc:insertFacet` e o `cc:renderFacet`.

Insira as facetas com `cc:insertFacet`

Usamos essa tag dentro do bloco *implementation* do nosso componente quando queremos inserir naquele local o `f:facet` que foi definido. É como se copiássemos e colássemos o `f:facet` para o local onde usarmos a tag `cc:insertFacet`. Assim, dentro do nosso componente teremos um `f:facet`.

Quando temos a implementação do nosso componente e há um outro que espera a presença de determinadas *facets*, precisamos inseri-lo:

```

...
<cc:implementation>
  <p:panelGrid columns="#{cc.attrs.colunas}">
    <cc:insertFacet name="header"/>
    <cc:insertFacet name="footer"/>
    <cc:insertChildren/>
  </p:panelGrid>
</cc:implementation>
...

```

Nessa implementação, fizemos um invólucro do componente `p:panelGrid`, e ele já suporta dois *facets* chamados “header” e “footer”, que servem para definir os conteúdos do cabeçalho e rodapé, respectivamente.

Mostre as facetas com `cc:renderFacet`

Essa *tag* deve ser usada quando, em vez de copiar a `f:facet` para dentro do nosso componente, nós simplesmente quisermos renderizar seu conteúdo no local onde colocamos a `cc:renderFacet`.

Enquanto `cc:insertFacet` copia a *tag* `f:facet` utilizada, a `cc:renderFacet` copia e renderiza o conteúdo dentro da faceta.

Poderíamos usar essa *tag* para renderizar um conteúdo alternativo para nosso componente. Considere uma *tag* de segurança, que só mostra o que está envolvido por ela se o usuário tiver acesso, e caso não tenha, um conteúdo diferente é mostrado.

Esse conteúdo alternativo poderia ser fixo ou somente um texto, mas para enriquecer nosso exemplo, vamos considerar que o usuário do nosso componente pode especificar qualquer conteúdo alternativo.

```
...
<seguranca:proteger>
  <h:commandButton value="Excluir"
    action="#{automovelBean.excluir(objeto)}"/>
  <f:facet name="semAcesso">
    <h:commandButton value="Solicitar Exclusão"
      action="#{controller.solicitaExcusao(objeto)}"/>
  </f:facet>
</seguranca:proteger>
...
```

Criamos aqui uma *tag* que verifica se o usuário tem ou não acesso para excluir um determinado objeto. Caso não tenha, nosso componente permite não mostrar nada, mostrar uma mensagem, ou como no exemplo, exibir um botão alternativo para solicitar a exclusão. Solicitação essa, que deverá ser analisada por um administrador.

A implementação do nosso componente é a seguinte:

```
...
<cc:implementation>
  <ui:fragment rendered="#{currentUser.role == 'admin'}">
    <cc:insertChildren/>
  </cc:fragment>
  <ui:fragment rendered="#{currentUser.role != 'admin'}">
    <cc:renderFacet name="semAcesso"/>
  </cc:fragment>
</cc:implementation>
```

```
</cc:implementation>
...
```

Novamente temos uma implementação bem simples. Como podemos ver, temos um `if-else` verificando se o usuário é administrador. Nesse caso, mostramos o conteúdo interno do nosso componente via `cc:insertChild`; e caso não tenha acesso, renderizamos o que estiver dentro da `f:facet` chamada “semAcesso”.

9.7 CRIAR UM JAR COM COMPONENTES CUSTOMIZADOS

Os *composite components* usam a mesma estrutura de recursos que o JSF usa para imagens, folhas de estilos e javascripts, vista na seção 7.34. Em todos esses casos, para disponibilizar esses recursos em um jar basta colocarmos a pasta “resources” dentro do `META-INF` do jar.

A seguir vemos como ficam os recursos dentro do projeto e dentro do jar.

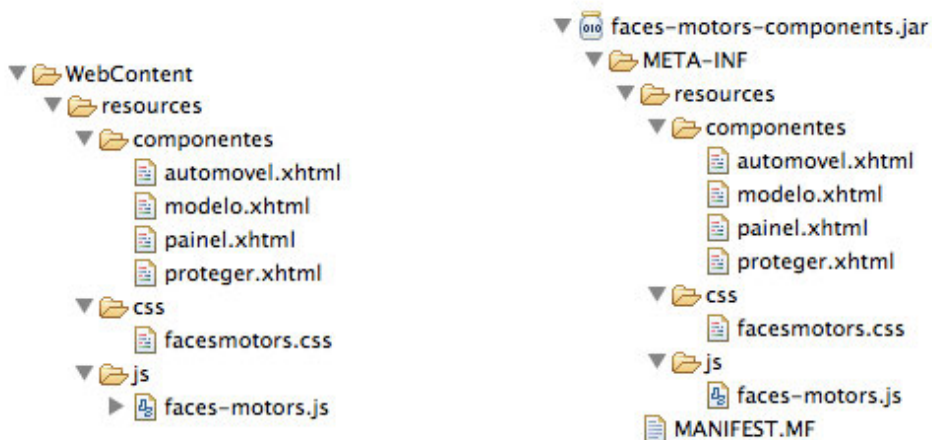


Figura 9.12: A estrutura dos componentes no projeto e no jar é a mesma

Adicionando o jar em outra aplicação, esses recursos ficam disponíveis como se estivessem dentro dela. Podemos referenciar qualquer recurso e qualquer componente customizado da mesma forma que fazemos quando eles estão dentro da nossa aplicação, e não em um jar.

CAPÍTULO 10

Truques que podem aumentar a escalabilidade e a performance da sua aplicação

Vimos como usar corretamente carregamento *lazy*, evitando problema de N+1 consultas, vimos como usar o escopo correto e várias outras técnicas que envolviam JPA e o JSF. No entanto, mesmo com tudo isso, pode ser que precisemos de algo mais em nossa aplicação.

Nesse capítulo, veremos como trabalhar com os diversos caches que as implementações da JPA nos fornecem. Usando JSF, aprenderemos como podemos diminuir o número de requisições ao servidor e, mesmo quando as tiver de fazer, como elas podem ser executadas bem mais rápido.

10.1 UTILIZE O CACHE DE SEGUNDO NÍVEL DA JPA

A JPA possui dois níveis de cache. O primeiro deles, mesmo sem sabermos, já estamos usando desde o início desse livro, já que não requer nenhuma configuração adicional. Na seção 4.2 foi dito que JPA é contextual; só faltou dizer que esse contexto provê o que chamamos de cache de primeiro nível.

Cada objeto que é carregado pelo `EntityManager` acaba ficando no cache desse `EntityManager`. A partir de então, toda vez que o mesmo objeto for buscado pela chave, ele será devolvido imediatamente, sem necessidade de um novo acesso ao banco para isso. A imagem a seguir ilustra esse funcionamento.

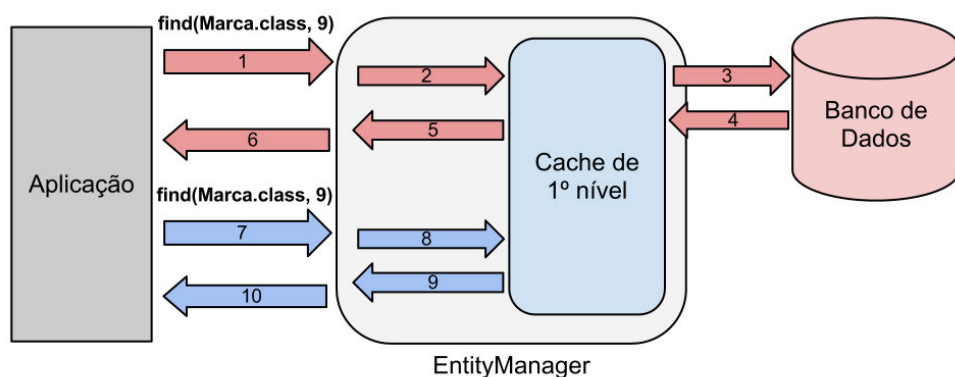


Figura 10.1: Cache de primeiro nível evita que a mesma Marca seja buscada novamente no banco

Esse mecanismo tem um potencial muito grande de economizar consultas. Basta que a mesma instância de `EntityManager` seja usada, pois o cache de primeiro nível é feito apenas dentro dela. O problema é que, geralmente, cada instância dessa vive apenas durante uma requisição do usuário, sendo fechada logo em seguida para liberar recursos do servidor.

Ativando o cache de segundo nível

Já vimos que nenhuma configuração é necessária para usarmos o cache de primeiro nível, porém ele vive somente enquanto viver a instância de `EntityManager`. Como esse tempo é curto, temos um cache bom, mas que pode ser usado por pouco

tempo. Outra limitação do cache de primeiro nível é que ele não é compartilhado, assim, um usuário carrega a lista de `Marca` e o outro usuário tem que carregar novamente a mesma lista.

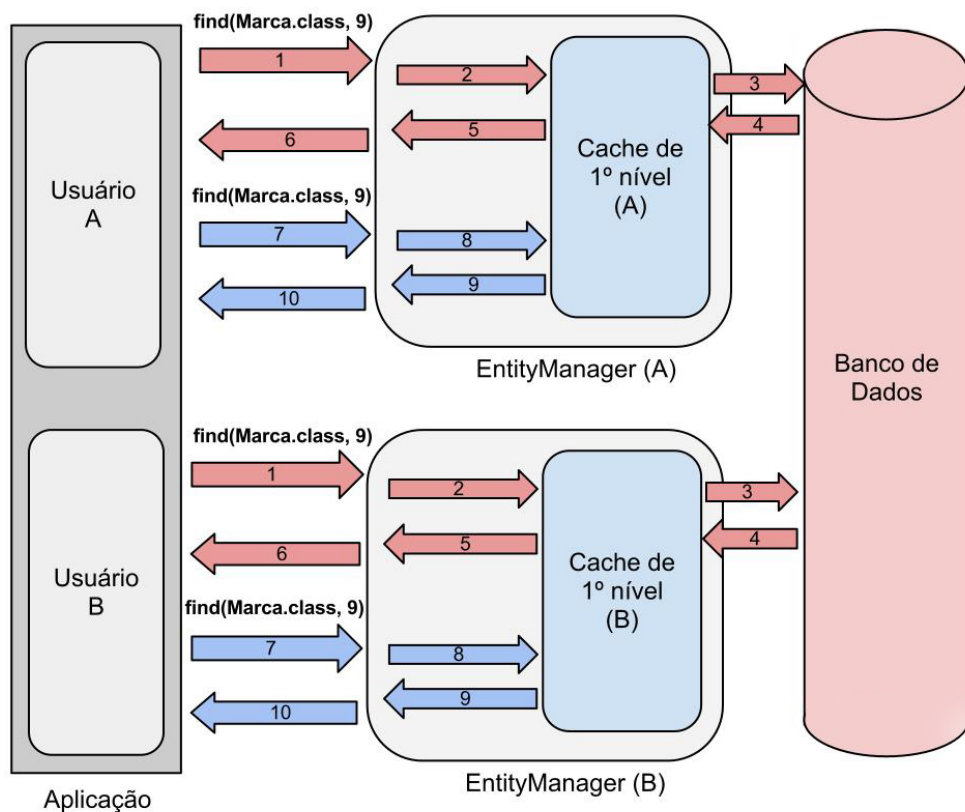


Figura 10.2: Cache de primeiro nível

Para resolvermos essa situação, temos o cache de segundo nível, que é compartilhado entre todas as instâncias de `EntityManager`. Na prática, quer dizer que se um usuário carregou a lista de `Marca` ela estará disponível, seja pra ele mesmo usar novamente em requisições futuras ou para outros usuários.

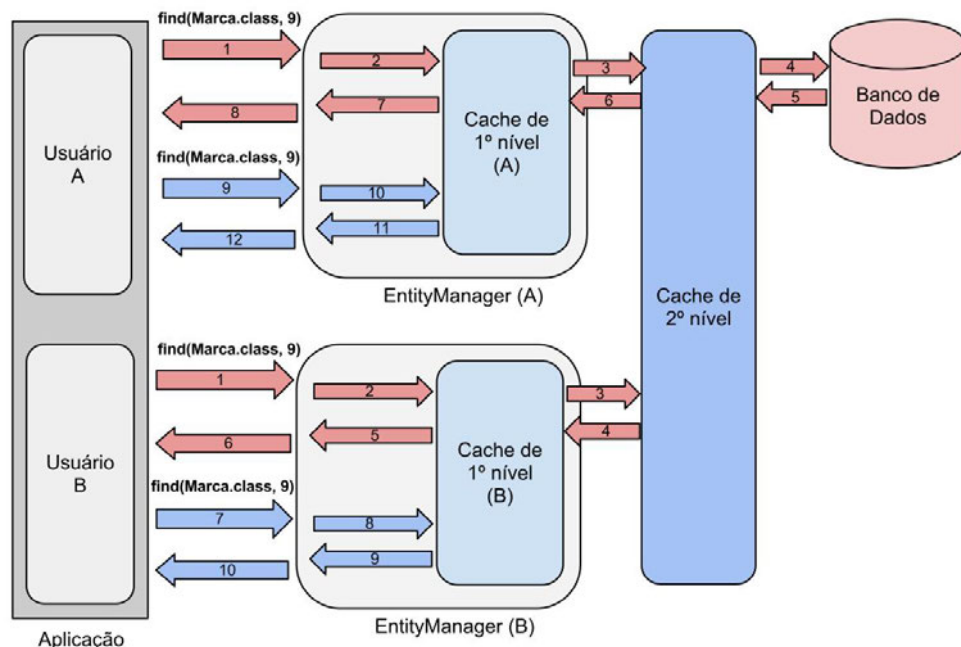


Figura 10.3: Cache de segundo nível

Com o cache de segundo nível ativado, uma vez que alguma instância de `EntityManager` buscou o objeto, ele está disponível para as outras. Na figura vemos que o que foi buscado pelo usuário A beneficiou o usuário B. Como o cache de segundo nível é compartilhado, não faz muita diferença se os `EntityManagers` que estão acessando foram criados para o mesmo usuário ou para usuários diferentes.

A JPA 2.0 tem suporte nativo ao cache de segundo nível e ele é configurado através do elemento `shared-cache-mode` do `persistence.xml`. Podemos especificar cinco modos diferentes de cache:

- `ALL`: automaticamente habilita todas as entidades no cache de segundo nível;
- `NONE`: desabilita o cache de segundo nível para todas as entidades;
- `ENABLE_SELECTIVE`: habilita o cache de segundo nível para todas as entidades que estiverem anotadas com `@Cacheable`;

- `DISABLE_SELECTIVE`: habilita o cache de segundo nível para todas as entidades, desabilitando somente as que estiverem anotadas com `@Cacheable(false)`;
- `UNSPECIFIED`: quando não especificamos nada, esse é o valor assumido pelo `shared-cache-mode`, e nesse caso, cada implementação de JPA tem liberdade para definir qual modo será habilitado.

Um exemplo de configuração pode ser visto a seguir:

```
<persistence ...>
  <persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
    <properties>
      ...
    </properties>
  </persistence-unit>
</persistence>
```

Uma vez que nossa entidade está configurada no cache de segundo nível, pelo uso da anotação `@Cacheable`, não precisamos fazer nada diferente ao utilizá-la. Automaticamente a teremos disponível por bastante tempo, evitando muitos acessos ao banco de dados.

```
@Entity @Cacheable
public class Marca {
    ...
}
```

Mas ainda precisamos de uma implementação de cache, sendo que, possivelmente, as mais utilizadas são *EHCache* e *Infinispan*.

Ambas as implementações possuem configurações específicas, nas quais podemos indicar para cada entidade o número máximo de objetos que ficarão em memória, e depois o que será armazenado em disco, etc. Como essas configurações variam muito de acordo com cada tipo de aplicação, no trecho de código a seguir colocaremos apenas um exemplo, em cuja configuração padrão usaremos o módulo *hibernate-ehcache*.

```
...
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
```

```
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
<properties>
    ...
    <property name="hibernate.cache.region.factory_class"
        value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
</properties>
</persistence-unit>
...
```

Como funciona a alteração de um objeto que está no cache

Agora que temos um cache de segundo nível funcionando, o que ocorre quando já carregamos uma lista de `Marca` nesse cache e um usuário edita uma delas?

Como a alteração será feita pela própria JPA, no momento em que esse objeto alterado é enviado para o banco, ele é também atualizado no cache de segundo nível. No entanto, caso a versão antiga do objeto estivesse carregada no cache de primeiro nível de algum outro `EntityManager`, esse objeto ficaria desatualizado dentro dessa outra `EntityManager`.

Como a JPA cuida da atualização do cache de segundo nível, nosso único ponto de falha passa a ser o cache de primeiro nível. Agora o curto tempo de vida da `EntityManager`, que antes era uma desvantagem, passa a ser uma vantagem. Como o tempo que dura uma requisição geralmente é bem curto, e esse seria também o tempo de vida da `EntityManager`, para que houvesse algum erro de concorrência precisaríamos, no mesmo “segundo”, que dois usuários editassem uma mesma instância de uma classe de entidade - que geralmente muda pouco, porque se assim não fosse, não estariam no cache de segundo nível.

Temos, no entanto, que ponderar dois importantes pontos. O primeiro é a duração das nossas requisições. Se tivermos processamentos demorando muito, a chance de concorrência será maior. E por isso a palavra “segundo” do parágrafo anterior está entre aspas, se nosso processamento demorar dez segundos, estaremos aumentando a chance de concorrência.

Outro ponto que precisamos levar em consideração é a probabilidade de concorrência versus o benefício do cache. Como cada aplicação possui uma dinâmica diferente, isso precisa ser avaliado caso a caso. Na nossa aplicação de exemplo, certamente teremos muito mais usuários consultando automóveis do que os alterando. Isso nos leva a crer que para essa aplicação as três entidades, `Marca`, `Modelo` e `Automovel`, valem a pena estar no cache de segundo nível.

A estrutura do cache de segundo nível

Nas figuras que ilustravam o funcionamento dos caches de primeiro e segundo níveis usamos sempre como exemplo o método `find`, passando a classe e o `id` que gostaríamos de buscar. A estrutura do cache de segundo nível é parecida com a figura 10.4.

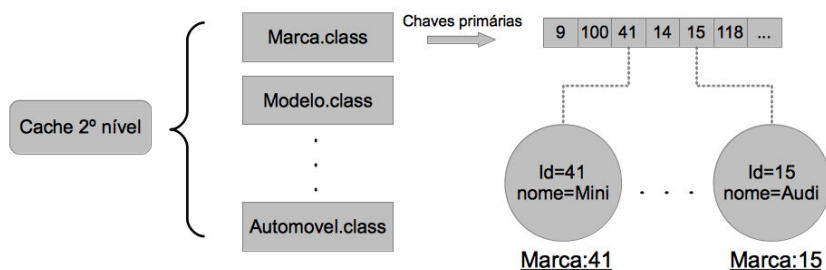


Figura 10.4: Estrutura do cache de segundo nível

Com isso o cache sabe recuperar objetos de qualquer entidade, desde que baseados na chave primária. Mas e quando não buscamos pela chave? O que acontece se fizermos uma pesquisa com diversos filtros e ela sempre retornar os mesmos resultados? Nesse caso, precisaremos de um cache diferente, o cache de consulta. Com o cache de segundo nível como vimos até agora só conseguimos encontrar objetos pela chave.

10.2 FAÇA CACHE DE CONSULTAS

A página inicial da nossa aplicação irá apresentar os últimos automóveis cadastrados, e seria interessante que pudéssemos nos beneficiar do cache, afinal de contas a lista muda muito pouco se comparado com a quantidade de visualizações que acontece nela. Imagine quantas vezes a mesma consulta seria feita, sempre com o mesmo resultado.

Para resolver esse problema, temos o cache de consultas. Esse recurso, no entanto, ainda não faz parte da JPA, então precisaremos usar recursos específicos do Hibernate. De toda forma, a JPA já prevê que em determinados momentos iremos precisar de funcionalidades específicas da implementação que estamos usando, possibilitando-nos desenvolver aplicações que, mesmo com recursos específicos de uma implementação, continuem portáveis até certo ponto.

Para ativar o cache de consultas na nossa aplicação, precisamos da seguinte configuração:

```
...
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    ...
    <property name="hibernate.cache.use_query_cache"
      value="true" />
  </properties>
</persistence-unit>
...
```

O cache de consultas só funciona se o cache de segundo nível estiver ativado. E somente será possível guardar no cache consultas que retornem objetos que estão com o cache habilitado com a anotação `@Cacheable`. Além disso, precisamos habilitar o cache em cada consulta que realizarmos. No trecho de código a seguir, vemos como podemos habilitar o cache em uma `@NamedQuery`.

```
@NamedQuery(name=Automovel.LISTAR_DESTAQUES,
            query="select a from Automovel a",
            hints={
                @QueryHint(name="org.hibernate.cacheable", value="true"),
                @QueryHint(name="org.hibernate.cacheRegion",
                    value=Automovel.LISTAR_DESTAQUES)})

@Entity
@Cacheable
public class Automovel {

    public static final String LISTAR_DESTAQUES =
        "Automovel.buscarDestaque";

    ...
}
```

Com a anotação `@QueryHint`, podemos colocar informações específicas da implementação de JPA que estamos usando nas *queries* e *named queries*. Apesar da implementação em questão ser o Hibernate, o funcionamento seria o mesmo usando outra implementação.

Através das *query hints*, por mais que especifiquemos uma informação proprietária, usamos apenas *Strings* para isso, evitando uma dependência com classes do Hibernate. Caso encontre uma *query hint* que ela não conhece, ela será ignorada.

No exemplo acima estamos usando uma `@NamedQuery`, mas podemos usar as *query hints* em objetos `Query` também.

```
Query query = JpaUtil.getEntityManager()
    .createQuery("select a from Automovel a", Automovel.class);
query.setHint("org.hibernate.cacheable", true);
query.setHint("org.hibernate.cacheRegion", Automovel.LISTAR_DESTAQUES);
List<Automovel> automoveis = query.getResultList();
```

Em ambos os exemplos usamos o nome da região igual ao nome da *named query*. Podemos dar qualquer nome para a região de cache que usaremos.

Obviamente podemos facilitar esse uso com métodos auxiliares como o seguinte:

```
Query query = JpaUtil.getEntityManager()
    .createQuery("select a from Automovel a", Automovel.class);
JpaUtil.enableCache(query, Automovel.LISTAR_DESTAQUES);
List<Automovel> automoveis = query.getResultList();

...
public class JpaUtil {
    ...
    public static Query enableCache(Query query, String region) {
        query.setHint("org.hibernate.cacheable", true);
        query.setHint("org.hibernate.cacheRegion", region);
        return query;
    }
}
```

Diferentemente do cache de segundo nível de entidades, que guarda o objeto pelo tipo e pelo `id`, a estrutura do cache de consulta funciona como o esquema a seguir:

Consulta	Parâmetros	Resultado
<code>select m from Modelo m</code>	---	Modelo:12, Modelo:17, Modelo:1, Modelo:4, Modelo:70,...
<code>select a from Automovel a where a.modelo.potencia > :potencia</code>	<code>potencia=500</code>	Automovel:71, Automovel:94, Automovel:101, Automovel:14,...
<code>select a from Automovel a where a.modelo.potencia > :potencia</code>	<code>potencia=200</code>	Automovel:1, Automovel:2, Automovel:3, Automovel:12, Automovel:71, Automovel:94,...

Figura 10.5: Estrutura do cache de consultas

Como vemos, o cache de consulta toma como base tanto a consulta quanto os parâmetros passados, já que a mesma consulta pode ser executada com parâmetros diferentes, tendo outro resultado.

Uma vez que a consulta é executada, o cache guarda os tipos e `id` dos objetos retornados pela consulta e, a partir de então, o funcionamento é o mesmo do cache de segundo nível de entidades. É como se o cache de consultas transformasse a nossa pesquisa em uma série de buscas baseadas em chave primária, e como já vimos, nesse caso tiramos proveito do cache de segundo nível.

Tratando cache desatualizado

Enquanto o cache de segundo nível consegue lidar bem com atualização de objetos, o cache de consulta não consegue fazer isso automaticamente. Ao alterarmos uma propriedade de um objeto, pode ser que ele passe a aparecer ou então deixe de aparecer como resultado de uma consulta que já está no cache. Como saber se o cache deve ou não ser invalidado? A resposta é que automaticamente não sabemos.

Para solucionar essa questão, usamos as regiões que definimos nos exemplos anteriores. Podemos colocar várias consultas ou entidades em uma região, e então, ao invalidarmos uma região, todas as consultas e entidades que estavam armazenadas nela saem do cache. Fica a cargo do desenvolvedor invalidar a região necessária a cada momento.

Se existe uma região chamada `Automovel.buscarDestques`, que armazena os automóveis recentemente cadastrados, ao inserirmos um novo `Automovel` iremos invalidar essa região para que da próxima vez que a consulta seja feita ela traga esse novo objeto do banco. A partir da segunda vez que a consulta for executada, o cache voltará a retornar a resposta sem precisar acessar o banco de dados.

Para invalidar o cache de consultas precisamos explicitamente acessar a API do Hibernate, mas para manter a portabilidade do nosso código, utilizamos métodos auxiliares como podemos ver a seguir.

```
@ManagedBean @ViewScoped
public class AutomovelBean {
    public String salvar(Automovel auto) {
        EntityManager em = JpaUtil.getEntityManager();
        em.persist(auto);

        JpaUtil.evictCache(em, Automovel.LISTAR_DESTAQUES);

        return "listar";
    }
}

public class JpaUtil {
    ...
    public static void evictCache(EntityManager em, String region){
        try {
            Session session = (Session) em.getDelegate();
            Cache cache = session.getSessionFactory().getCache();
            cache.evictQueryRegion(region);
        }
        catch(Exception e) {
            // provavelmente a implementação não é o Hibernate
        }
    }
}
```

Esse tipo de código é uma forma de contornar a dependência que acabamos criando com uma determinada implementação da JPA, deixando-a encapsulada em um ponto do nosso código.

10.3 COLHENDO ESTATÍSTICAS DA NOSSA CAMADA DE PERSISTÊNCIA

Quando falamos de otimização na camada de persistência, pensamos em evitar consultas ao banco e em fazer consultas que retornem somente os dados necessários.

Na seção 5.14, enquanto falávamos da importância dos relacionamentos lazy, comentamos dos milhares de registros que deixamos de ler, porém, essa conta foi feita em cima de um cenário simples, conhecido e controlado. Mas como saber quantos objetos são trazidos do banco para montar uma página que lista vinte automóveis, por exemplo?

Para obtermos dados estatísticos da camada de persistência, utilizaremos mais um recurso específico do Hibernate, que provê estatísticas de quantos objetos foram trazidos do banco, quantas consultas foram feitas e quantas não precisaram acessar o banco porque foi resolvida pelo cache, bem como quantidade de *inserts*, *updates* e *deletes*. Enfim, temos uma telemetria da camada de persistência. Para isso, o primeiro passo é habilitar essa funcionalidade:

```
...
<persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    ...
    <property name="hibernate.generate_statistics" value="true"/>
  </properties>
</persistence-unit>
...
```

A partir desse momento, podemos gerar no log a informação estatística que nos interessar, a cada requisição, por meio de um `Filter`, ou podemos ainda colocar essas informações dentro de um `PhaseListener` e gerar essas informações para cada fase do ciclo de vida do JSF. Assim, isolamos o código de obtenção de estatísticas.

As estatísticas são obtidas a partir da `SessionFactory` do Hibernate, que nos permite acessar o objeto que contém as informações, e que implementa a interface `org.hibernate.stat.Statistics`.

```
public class JpaUtil {
    ...
    public static Statistics getStatistics(){
        EntityManager em = JpaUtil.getEntityManager();
        Session session = (Session) em.getDelegate();
        SessionFactory sf = session.getSessionFactory();
        Statistics stats = sf.getStatistics();
        return stats;
    }
}
```

```
}  
}
```

Nesse método, temos que sair da JPA e entrar na API do Hibernate. Podemos ainda fazer um método que gere as informações estatísticas e as adicione no log, e depois chamá-lo, seja no `Filter` ou no `PhaseListener`.

```
public class JpaUtil {  
    ...  
    public static void printStatistics(){  
        Statistics stats = JpaUtil.getStatistics();  
  
        log.info("Qtde de entidades buscadas: " +  
                stats.getEntityFetchCount());  
        log.info("Qtde de entidades carregadas: " +  
                stats.getEntityLoadCount());  
        log.info("Qtde de listas buscadas: " +  
                stats.getCollectionFetchCount());  
        log.info("Qtde de listas carregadas: " +  
                stats.getCollectionLoadCount());  
  
        double queryCacheHitCount = stats.getQueryCacheHitCount();  
        double queryCacheMissCount = stats.getQueryCacheMissCount();  
        double totalQueries = queryCacheHitCount + queryCacheMissCount;  
        double queryCacheHitRatio = (totalQueries == 0) ? 0 :  
                                     queryCacheHitCount / totalQueries;  
  
        log.info("Qtde de consultas encontradas no cache: " +  
                queryCacheHitCount);  
  
        log.info("Qtde de consultas fora do cache: " +  
                queryCacheMissCount);  
        log.info("Proporção de acerto do cache: " +  
                queryCacheHitRatio);  
  
        log.info("Qtde de consultas executadas: " +  
                stats.getQueryExecutionCount());  
  
        String[] queries = stats.getQueries();  
        for (int i = 0; i < queries.length; i++) {  
            log.info("Consulta " + i + ": " + queries[i]);  
        }  
    }  
}
```

```

    }

    log.info("Query + lenta: " +
            stats.getQueryExecutionMaxTimeQueryString());

    stats.clear();
}
}

```

Escrevemos informações de quantas entidades foram buscadas e quantas efetivamente foram trazidas do banco, e depois fizemos o mesmo com as coleções. Essas coleções são carregamentos de relacionamento “*-to-many” que foram disparados.

Fizemos também uma análise da efetividade do cache de consultas e no final até escrevemos cada consulta executada. Essas estatísticas são exemplificativas, e muitas outras são possíveis, como por exemplo uma análise do cache de segundo nível de entidades, estatísticas por região de cache entre outras informações.

No final do método, limpamos as estatísticas para que os números apresentados sejam referentes somente a cada requisição ou fase do JSE, e não cumulativos.

10.4 RELACIONAMENTOS EXTRA-LAZY

A seção 5.14, onde vimos a importância do *lazy load*, é muito importante, já que, sem esse carregamento sob demanda, a performance das nossas aplicações seria sofrível. Lá, vimos o exemplo do relacionamento entre as entidades `Marca` e `Modelo`, no qual, mantendo o *lazy* de `Marca` com sua lista de `Modelo`, a listagem abaixo não carrega nenhuma instância de `Modelo` do banco:

```

public class Marca {

    @Id @GeneratedValue
    private Integer id;
    ...
    @OneToMany(mappedBy="montadora")
    private List<Modelo> modelos;
    ...
}

public class MarcaBean {
    private List<Marca> marcas;
}

```



```
...
public List<Marca> getMarcas() {
    if (marcas == null) {
        marcas = JpaUtil.getEntityManager()
            .createQuery("select m from Marca m", Marca.class)
            .getResultList();
    }
    return marcas;
}
}
```

```
<h:dataTable value="#{marcaBean.marcas}" var="marca">
    <h:column>
        <f:facet name="header">Nome da Marca</f:facet>
        #{marca.nome}
    </h:column>
</h:dataTable>
```

Nesse `h:dataTable`, estamos apenas listando as `marcas`. Como já vimos que, por padrão, o relacionamento é *lazy*, a lista de `modelos` da `Marca` não será carregada. Mas se alterarmos nossa listagem para incluir a quantidade de modelos como a seguir, o comportamento muda.

```
<h:dataTable value="#{marcaBean.marcas}" var="marca">
    <h:column>
        <f:facet name="header">Nome da Marca</f:facet>
        #{marca.nome}
    </h:column>
    <h:column>
        <f:facet name="header">Qtde de Modelos</f:facet>
        #{fn:length(marca.modelos)}
    </h:column>
</h:dataTable>
```

A função JSTL `fn:length` mostra o tamanho de uma `String` ou então o `size()` de uma `Collection`. Nesse exemplo, é como se estivéssemos chamando `marca.getModelos().size()`. Como invocamos um método da API do Java, que não foi desenvolvido pensando em banco de dados, ele usa os dados em memória, e para que isso funcione, o comportamento padrão é buscar a lista.

Poderíamos aqui querer otimizar por nós mesmos e criar um novo método que execute um `count` via `SQL`, imaginando que a camada de persistência, em casos

simples como esse, mais complicada do que ajuda. Porém, estaríamos enganados mais uma vez. Apesar de não estar especificado na JPA, temos um recurso específico do Hibernate chamado *extra lazy*, que habilita uma análise mais aprofundada do uso que se pretende fazer da coleção, para somente então decidir se os elementos serão ou não buscados.

Com *extra lazy*, é analisado que método está sendo chamado nessa coleção, e caso seja algo resolvível via banco, sem precisar carregar os elementos, o Hibernate efetua essa operação em vez de carregá-los. Como configurar e alguns exemplos de uso podem ser vistos nos trechos de código a seguir.

```
import org.hibernate.annotations.LazyCollection;
import org.hibernate.annotations.LazyCollectionOption;
...
public class Marca {

    @Id @GeneratedValue
    private Integer id;
    ...
    @OneToMany(mappedBy="montadora")
    @LazyCollection(LazyCollectionOption.EXTRA)
    private List<Modelo> modelos;
    ...
}
```

Exemplos de uso do *extra lazy*:

```
// a Marca é carregada do banco, mas a lista de modelos não
Marca marca = entityManager.find(Marca.class, 1);

// com extra lazy, é feito um count no banco:
// select count(id) from Modelo where marca_id=?
int qtdeModelos = marca.getModelos().size();

// isEmpty pode ser descoberto a partir do resultado do count
boolean listaVazia = marca.getModelos().isEmpty();

Modelo modelo = entityManager.find(Modelo.class, 1);
// contains não traz a lista inteira, e sim, faz um select invertido:
// select 1 from Modelo where marca_id=? and id=?
boolean modeloNaLista = marca.getModelos().contains(modelo);
```

10.5 PAGINAÇÃO VIRTUAL E REAL DE DADOS

O conceito de paginação é comum a todos nós. Ele está presente quando acessamos nosso e-mail ou quando fazemos uma busca no Google, quando entramos em um site que possui produtos à venda ou quando acessamos redes sociais. Paginação é ir carregando a informação para o usuário conforme ele vai avançando na visualização dos dados. Ao fazer uma pesquisa que retorne milhares de resultados, por que deveríamos carregar todos eles se o usuário geralmente olha apenas os primeiros dez ou vinte elementos exibidos?

O que costuma ocorrer é que é mais simples carregar tudo do banco e deixar algum componente visual exibir os dados em páginas de dez ou vinte elementos cada. O problema é que isso geralmente funciona bem em aplicações jovens, com poucos dados, mas à medida em que a quantidade de dados aumenta, a performance da aplicação diminui.

Costumamos chamar essa exibição fracionada de dados de paginação virtual, que traz somente benefícios cosméticos para nossa aplicação. Já a paginação que realmente busca os dados em porções pequenas é chamada de paginação real.

Devemos procurar desde o início da aplicação trabalhar com paginação real, porque se logo do começo fizermos a virtual teremos dois trabalhos: o de fazer e o de refazer. Obviamente, como sempre existem casos e casos, em situações bem pontuais, nas quais a natureza dos dados já faz com que eles sejam limitados, o uso de paginação virtual não poderá não ser fator determinante para uma possível performance ruim da aplicação. Mas a boa prática é procurarmos sempre trabalhar com paginação real.

Vamos mostrar como fazer as paginações tanto virtual, quanto real, através do uso do componente `dataTable` do Primefaces, já que esse oferece suporte simples às paginações.

```
<h:form>
  <p:dataTable value="#{modeloBean.modelos}" var="modelo"
    paginator="true" rows="10">
    <p:column>
      <f:facet name="header">ID</f:facet>
      #{modelo.id}
    </p:column>
    <p:column>
      <f:facet name="header">Marca</f:facet>
      #{modelo.marca.nome}
    </p:column>
  </p:dataTable>
</h:form>
```

```
</p:column>
<p:column>
    <f:facet name="header">Descrição</f:facet>
    #{modelo.descricao}
</p:column>
</p:dataTable>
</h:form>

@ManagedBean @ViewScoped
public class ModeloBean {
    private List<Modelo> modelos;
    ...
    public List<Modelo> getModelos() {
        if (modelos == null) {
            modelos = JpaUtil.getEntityManager()
                .createQuery("select m from Modelo m", Modelo.class)
                .getResultList();
        }

        return modelos;
    }
    ...
}
```

Olhando o exemplo acima, a única diferença em ambos os códigos é na declaração do `p:dataTable`, onde especificamos que teremos paginação e que cada página terá 10 linhas. Com isso já temos o resultado a seguir.

<div><div><div>1</div><div>2</div><div>3</div></div><div><div><<</div><div>>></div></div></div>		
ID	Marca	Modelo
11	Ford	Fusion
12	Ford	F1000
13	Fiat	Bravo
14	Ferrari	F40
15	Ferrari	F50
16	Ferrari	Enzo
17	Hyundai	ix35
18	Hyundai	i30
19	Hyundai	Azera
20	Hyundai	Sonata
<div><div><div>1</div><div>2</div><div>3</div></div><div><div><<</div><div>>></div></div></div>		

Figura 10.6: Listagem simples usando paginação

Nesse caso, tivemos a paginação virtual, já que todas as informações foram carregadas do banco de dados, mesmo as que não estão sendo mostradas.

Usando paginação real

O importante para a performance da aplicação é a paginação real, porém esta não vem de graça como a virtual. No nível da JPA, temos métodos para limitar a quantidade de registros retornados em uma consulta e para informar a partir de qual registro vamos começar a listar. A listagem a seguir mostra um exemplo de paginação real.

```
TypedQuery<Modelo> query = em.  
    createQuery("select m from Modelo m", Modelo.class);  
  
int tamanhoPagina = 10;  
query.setMaxResults(tamanhoPagina);  
  
List<Modelo> pagina1 = query.setFirstResult(0 * tamanhoPagina).  
    getResultList();  
  
List<Modelo> pagina2 = query.setFirstResult(1 * tamanhoPagina).
```

```

        getResultList();

List<Modelo> pagina3 = query.setFirstResult(2 * tamanhoPagina).
        getResultList();

```

Usando os métodos `setMaxResults` e `setFirstResult` podemos delimitar o tamanho de cada página e qual será o primeiro elemento de cada uma. Porém, como integrar isso ao componente `p:dataTable` que irá exibir cada página? E como saber qual o total de páginas?

O `primefaces`, além dos componentes visuais, possui classes auxiliares para eles. No caso do `p:dataTable` existe uma classe chamada `LazyDataModel`, que é um `javax.faces.model.DataModel` com suporte a carregamento de objetos sob demanda.

Toda `dataTable`, seja ela nativa do JSF ou específica do `Primefaces`, obtém seus dados a partir de um `DataModel` e mesmo quando passamos diretamente uma lista para a `dataTable`, por dentro, ela transforma isso em um `DataModel`.

Assim, criando um `LazyDataModel`, estaremos passando para a `p:dataTable` algo com que ela naturalmente sabe lidar, porque, além do comportamento normal de uma `dataTable`, o componente do `Primefaces` conhece o conceito de paginação.

É perfeitamente possível fazer paginação real com o componente padrão `h:dataTable`. A diferença é que teríamos que criar links ou botões para navegar nas páginas para frente e para trás.

A maior dificuldade na implementação da nossa `LazyDataModel` é que precisaremos de uma consulta para saber quantos elementos existem no total. Sem isso, não é possível criar corretamente os botões para mostrar até que página temos.

```

<h:form>
    <p:dataTable value="#{modeloBean.modelosLazyDataModel}"
        var="modelo" paginator="true" rows="10" lazy="true">

        ...

    </p:dataTable>
</h:form>

```

```

@ManagedBean @ViewScoped
public class ModeloBean {
    private LazyDataModel<Modelo> modelosLazyDataModel;

```

```

    public LazyDataModel<Modelo> getModelosLazyDataModel() {
        if (modelosLazyDataModel == null) {
            String jpql = "select m from Modelo m";
            String count = "select count(m.id) from Modelo m";
            modelosLazyDataModel = new
                QueryDataModel<Modelo>(jpql, count);
        }

        return modelosLazyDataModel;
    }
    ...
}

```

Até aqui podemos perceber a mudança na `p:dataTable`, que agora além de apontar para o `modelosLazyDataModel`, também tem a propriedade `lazy=true`.

No código do *Managed bean*, saiu a listagem direta no banco e, em vez disso, instanciamos a classe `facesmotors.persistence.QueryDataModel`, passando no construtor duas consultas: a primeira retorna os objetos a serem listados e a segunda devolve a quantidade de registros. A implementação da `QueryDataModel` é a seguinte:

```

public class QueryDataModel<T> extends LazyDataModel<T> {

    private String jpql;

    public QueryDataModel(String jpql, String jpqlCount) {
        this.jpql = jpql;

        Long count = (Long) JpaUtil.getEntityManager()
            .createQuery(jpqlCount)
            .getSingleResult();

        setRowCount(count.intValue());
    }

    @Override
    public List<T> load(int first, int pageSize, String sortField,
        SortOrder sortOrder, Map<String, String> filters) {

```

```
        return JpaUtil.getEntityManager().createQuery(jpql)
            .setFirstResult(first)
            .setMaxResults(pageSize)
            .getResultList();
    }
}
```

Como o componente `p:dataTable` suporta ordenação e filtragem, essas informações são também passadas para o método `load` que precisamos implementar. Para a nossa necessidade, porém, basta usar os dois primeiros parâmetros, que representam a linha atual e o tamanho da página.

O inconveniente dessa implementação é que precisamos informar duas consultas, uma para buscar os dados e outra para realizar o `count`.

Agora que temos uma implementação de `LazyDataModel`, podemos fazer qualquer consulta da nossa aplicação passar a usar paginação real!

A apresentação da `dataTable` com paginação real não se altera, mas a performance da nossa aplicação, quando tivermos exibindo listas com muitos registros, com certeza melhora.

10.6 UTILIZANDO POOL DE CONEXÕES

Para nos conectar com o banco de dados, estamos especificando propriedades como caminho do banco, usuário e senha no arquivo `persistence.xml`. Além disso, já vimos que existem configurações específicas do Hibernate que podemos adicionar, a fim de habilitar comportamentos que nos interessam, como por exemplo cache de consultas.

Veremos agora como configurar nesse mesmo arquivo um *pool* de conexões. Esse recurso é muito importante, já que cada vez que precisamos criar uma instância de `EntityManager`, ou seja, praticamente a cada requisição de cada usuário, essa instância internamente precisa de uma conexão com o banco de dados. Porém, conexão é um recurso caro demais para se ficar criando e fechando o tempo todo.

Toda vez que temos recursos muito caros, costumamos fazer um *pool*, que é uma reserva de objetos prontos para serem usados, em vez de demandar a criação de novos objetos sempre que precisarmos.

O Hibernate tem integração nativa com o pool que iremos utilizar aqui, o `c3p0`. A configuração de um pool não costuma mudar muito.

Aqui veremos a configuração do `c3p0`, mas o conceito é parecido para outras implementações. A configuração necessária é a seguinte:

```
<persistence...>
  <persistence-unit name="default" transaction-type="RESOURCE_LOCAL">
    ...
    <properties>
      <property name="javax.persistence.jdbc.url" value="..." />
      <property name="javax.persistence.jdbc.driver"
        value="..." />
      <property name="javax.persistence.jdbc.user" value="..." />
      <property name="javax.persistence.jdbc.password"
        value="..." />

      <property name="hibernate.c3p0.min_size" value="5" />
      <property name="hibernate.c3p0.max_size" value="20" />
      <property name="hibernate.c3p0.timeout" value="300" />
      <property name="hibernate.c3p0.max_statements" value="50" />
      ...
    </properties>
  </persistence-unit>
</persistence>
```

Como a grande maioria das nossas iterações com o banco de dados irá durar algo entre alguns milissegundos a poucos segundos, a quantidade de operações em execução simultaneamente no banco de dados não costuma ser grande, por isso especificamos um máximo de 20 conexões com o banco, usando a propriedade `hibernate.c3p0.max_size`.

Utilizamos a propriedade `hibernate.c3p0.min_size` para especificar que desejamos que sempre haja pelo menos 5 conexões prontas para uso. Como definimos um máximo de 20, sabemos que o número de conexões criadas vai flutuar dentro desse intervalo.

O que vai regular o número de conexões é a propriedade `hibernate.c3p0.timeout`, em que especificamos um tempo de 300 segundos. Se tivermos uma conexão sem uso por esse tempo, ela será fechada, respeitando somente o tamanho mínimo do *pool*.

O interessante ao utilizarmos um pool de conexões é que, por mais que a aplicação solicite explicitamente que uma conexão seja fechada, na prática ela somente volta para o pool para ser reutilizada em outro momento. Somente o próprio pool fecha as conexões quando elas não são mais necessárias.

E por fim, mas também muito importante, temos a propriedade `hibernate.c3p0.max_statements` que especifica um cache de `java.sql.PreparedStatement`. Quando damos os primeiros passos usando JDBC, aprendemos que é uma boa prática utilizar `PreparedStatement`, uma vez que os comandos de banco criados usando esse objeto ficam previamente preparados no banco. Porém, não conseguimos tirar proveito dessa preparação se sempre descartarmos o objeto `PreparedStatement`.

A utilização desse cache reflete claramente no banco de dados. Analisando seu log, percebemos apenas a chamada de comandos previamente compilados e otimizados, sejam eles comandos de `select`, `insert`, `update` ou `delete`. No entanto, temos que lembrar que por mais que dois `inserts` no banco tenham parâmetros diferentes, se tiverem a mesma estrutura, estaremos falando de um único `PreparedStatement`. Logo, o número de comandos no cache não leva em consideração os parâmetros passados, somente a estrutura do comando.

Todos os números apresentados aqui são exemplificativos, mas servem como uma boa base para iniciar a configuração da aplicação.

Porém, como qualquer otimização, é preciso verificar o comportamento da aplicação em cenários reais. Para isso, é possível também utilizar as estatísticas do Hibernate para medir a efetividade do pool, assim como medimos a efetividades dos nossos caches.

10.7 TORNANDO O JSF MAIS LEVE COM USO INTELIGENTE DE AJAX

Tratamos de AJAX e JSF na seção 9.4, mas o enfoque antes era melhorar a usabilidade da nossa aplicação. Veremos agora como a mesma ferramenta pode ser usada para melhorar a performance.

Quando fazemos uma requisição AJAX, uma subárvore de componentes é criada, e o que precisamos fazer é procurar criar uma subárvore que seja a menor possível, já que isso influencia na performance.

Vamos analisar as duas opções a seguir:

```
...
<f:ajax event="blur" render="dadosAutomovel">
  <h:panelGroup id="dadosAutomovel">
    ...
    Ano de Fabricação:
```

```

    <h:inputText id="anoFabricacao"
        value="#{automovelBean.automovel.anoFabricacao}"/>
    <h:message for="anoFabricacao"/>

    Ano do Modelo:
    <h:inputText id="anoModelo"
        value="#{automovelBean.automovel.anoModelo}"/>
    <h:message for="anoModelo"/>
    <h:panelGroup>
</f:ajax>
...

```

A classe `Automovel` poderia ter um código parecido com o seguinte:

```

@Entity
public class Automovel {
    ...
    @NotNull
    private Integer anoFabricacao;
    @NotNull
    private Integer anoModelo;
    ...
}

```

De acordo com o código do XHTML, cada vez que tirarmos o foco de algum componente que aceite o evento DOM *onblur*, o JSF irá fazer uma requisição para o servidor enviando o componente que originou o evento, por padrão, o `execute="@this"`.

Como retorno, todo o `h:panelGroup` com `id="dadosAutomovel"` seria renderizado novamente. Renderizar todo esse *container* novamente só para mostrar uma possível mensagem de erro seria um desperdício muito grande de processamento. A situação só não é pior porque, por padrão, a propriedade `execute` só envia o componente que dispara o evento em questão, no caso o *onblur*.

Uma possível solução para o problema seria:

```

<h:messages id="todasMensagens"/>
<f:ajax event="blur" render="todasMensagens">
    <h:panelGroup id="dadosAutomovel">
        ...
        Ano de Fabricação:
        <h:inputText id="anoFabricacao"

```

```

        value="#{auto.anoFabricacao}"/>
    Ano do Modelo:
    <h:inputText id="anoModelo"
        value="#{auto.anoModelo}"/>
    <h:panelGroup>
</f:ajax>

```

Com isso continuamos enviando somente o valor de cada `h:inputText` e na volta renderizamos somente um componente, o `h:messages`. Contudo, essa solução não é tão visual para o usuário quanto as anteriores, já que agora, para qualquer erro ocorrido, em vez de a mensagem aparecer do lado do campo que originou o erro, aparecerá no topo da página.

Uma solução tão eficiente quanto essa última, e que ainda assim mostra cada mensagem de erro ao lado do campo, é a seguinte:

```

...
<f:ajax event="blur">
    <h:panelGroup id="dadosAutomovel">
        ...

        Ano de Fabricação:
        <h:inputText id="anoFabricacao"
            value="#{automovelBean.automovel.anoFabricacao}">
            <f:ajax render="anoFabricacaoMsg"/>
        </h:inputText>
        <h:message id="anoFabricacaoMsg" for="anoFabricacao"/>

        Ano do Modelo:
        <h:inputText id="anoModelo"
            value="#{automovelBean.automovel.anoModelo}">
            <f:ajax render="anoModeloMsg"/>
        </h:inputText>
        <h:message id="anoModeloMsg" for="anoModelo"/>

    <h:panelGroup>
</f:ajax>
...

```

Agora, o `f:ajax` serve apenas para especificar o evento *onblur* como padrão para os componentes dentro dele. Poderíamos também tirar o `f:ajax` que envolve os demais e repetir `event="blur"` nos `f:ajax` internos.

Nessas duas últimas opções temos um único componente indo, e um único componente voltando do JSF. É criada uma subárvore com um único componente dentro, e como vimos na seção 7.3, o JSF faz uma série de processamentos baseado em sua árvore. Quanto mais *inputs* tivermos, mais será necessário converter, validar, aplicar valor; e depois mais será preciso percorrer a árvore gerando HTML como resposta.

10.8 QUANDO POSSÍVEL, MANIPULE COMPONENTES JSF NO LADO DO CLIENTE

Na seção anterior, revimos técnicas para fazer o ajuste fino do que é enviado e recuperado do servidor, através de AJAX, para termos somente o processamento necessário em cada requisição. Mas podemos fazer ainda mais.

Uma característica do JSF é que sua tela reflete o estado da árvore de componente que fica no servidor. Sendo assim, não temos como simplesmente criar um `<input type="text">` na tela via javascript, uma vez que, se esse campo não foi criado também na árvore do JSF, ele será ignorado.

Mas muitas vezes é interessante termos a alternativa de realizar alguns controles apenas na tela, sem ter que gerar uma requisição para, por exemplo, ocultar ou mostrar um outro campo, tabela ou outro componente.

Quando usamos componentes do Primefaces, temos uma facilidade que é a presença da propriedade `widgetVar`. Ela expõe um objeto javascript que podemos manipular do lado cliente, como podemos fazer com qualquer objeto usando bibliotecas javascript como o *jQuery*.

Combinando a API javascript que temos via `widgetVar` e o *jQuery*, podemos manipular nossa tela de forma muito flexível. Vamos explorar um caso simples de uma aplicação de venda de ingressos. Ao comprarmos um ingresso, podemos informar que temos acesso a alguma forma de desconto. Dentre os mecanismos de desconto temos: Estudante, Cartão de crédito e Idoso. Uma possível implementação da tela de compra de ingresso seria a seguinte.

```
@Entity
public class VendaIngresso {
    ...
    public enum TipoDesconto {ESTUDANTE, CARTAO_CREDITO, IDOSO}

    private String nomeComprador;
```

```

@ManyToOne
private Evento evento;

private boolean vendaComDesconto;
private TipoDesconto tipoDesconto;
....
}

...
<h:selectBooleanCheckbox value="#{ingressoBean.venda.vendaComDesconto}">
    <f:ajax render="tipoDescontoPanel"/>
</h:selectBooleanCheckbox>
Compra com desconto?

<h:panelGroup id="tipoDescontoPanel">
    <h:selectOneMenu value="#{ingressoBean.venda.tipoDesconto}"
        rendered="#{ingressoBean.venda.vendaComDesconto}">
        <f:selectItems value="#{IngressoBean.tiposDesconto}"
            var="desconto" itemValue="#{desconto}"
            itemLabel="#{desconto}"/>
    </h:selectOneMenu>
</h:panelGroup>

```

Não precisamos nem entrar muito em detalhes nesse código, afinal ele é bem simples e tem como objetivo nos lembrar de uma prática comum no desenvolvimento com JSF: a renderização condicional de uma parte da tela, que é combinada com alguma requisição AJAX que atualiza o componente que aparece ou some. Pela natureza simples da tela, o trecho que aparece ou some é pequeno, mas em uma tela complexa pode não ser.

Uma alternativa para isso é trazer o componente de desconto apenas oculto e então, via *jQuery*, mostrá-lo sem a necessidade de qualquer requisição para o servidor.

```

...
<h:form prependId="false">
    <h:outputScript library="primefaces" name="jquery/jquery.js"
        target="head"/>

    <h:outputScript>
        function atualizaValorDesconto(){
            if ($("#temDesconto").is(":checked")) {
                $("#tipoDesconto").show();
            }
        }
    </h:outputScript>

```

```
        }
        else {
            $("#tipoDesconto").hide();
        }
    }
}
</h:outputScript>

<h:selectBooleanCheckbox id="temDesconto"
    value="#{ingressoBean.venda.vendaComDesconto}" />
Compra com desconto?

<span id="tipoDesconto" style="display: none;">
    <h:selectOneMenu value="#{ingressoBean.venda.tipoDesconto}">
        <f:selectItems value="#{ingressoBean.tiposDesconto}"
            var="desconto" itemValue="#{desconto}"
            itemLabel="${desconto}" />
    </h:selectOneMenu>
</h:panelGroup>
</h:form>
```

Com essa alteração, não é necessária nenhuma requisição adicional para exibir o combo com as opções de desconto. Decidir qual a melhor solução não é trivial, já que a equação envolve variáveis de segurança, performance e manutenibilidade do código.

10.9 CONSIDERAÇÕES FINAIS SOBRE OTIMIZAÇÕES

Não é de hoje que o tema otimização causa fascínio em grande parte dos desenvolvedores. A oportunidade de reduzir o tempo de processamento de horas para segundos, ou de muitos minutos para milissegundos, dá uma sensação de poder. Porém, a busca por essa sensação pode se tornar um vício, e a otimização em excesso pode se tornar uma droga.

Muitas vezes, antes mesmo de se conhecer a quantidade de usuários, o ambiente onde a aplicação vai executar, e sem ter conhecimento das tecnologias envolvidas no desenvolvimento da aplicação, lançamos mão de velhos hábitos de otimização que consideramos boas práticas. Muitas dessas boas práticas inclusive fizeram sentido em algum momento do passado, mas muitas vezes não faz no projeto atual. Não é tão simples perceber isso quando estamos tentados a repetir a sensação da otimização.

Obviamente existem casos excepcionais, nos quais requisitos de performance

extremos guiam todo o *design* da aplicação, mas a menos que você trabalhe em um lugar especializado nesses tipos de aplicação, elas serão exceção, e não a regra.

Precisamos ponderar performance versus manutenibilidade de código. Não adianta muito ganhar um milissegundo e perder uma semana de desenvolvimento e depois mais várias horas todas as vezes que alguém precisar voltar àquele código.

Muitas vezes, por mais que a implementação da otimização não seja difícil, é mais código para manter, e fazer determinadas otimizações não compensam. Somente analisando detalhadamente a aplicação é que poderemos saber o que precisa ser otimizado.

CAPÍTULO 11

Conclusão

As especificações das quais esse livro trata trazem consigo um peso extra da mudança de paradigma. É comum ainda hoje pensar que orientação a objetos é boa em teoria, mas quando o assunto é performance o melhor é abandonar tudo e partir para coisas específicas de banco. Em um caso ou outro isso será verdade, mas a proporção com que isso é acontece é bem baixa, se comparada ao número de vezes que essa hipótese é levantada.

Igualmente temos a mudança de paradigma quando trabalhamos com JSF, já que o desenvolvimento Web é predominantemente *Action Based*. Mas vimos que, além produtivo, é perfeitamente possível desenvolver usando esse framework e o *Component Based*.

Apesar de suas vantagens, não estamos atrás do “santo graal”, e sabemos que cada ferramenta tem cenários onde ela é útil e onde ela não é a melhor opção. Por exemplo, imagine uma aplicação que você está desenvolvendo, na qual surge a necessidade de escrever suas próprias instruções SQL. Apesar das facilidades que uma ferramenta ORM oferece, pode ser que você tenha que escrever suas SQLs em algum momento.

Aqui a ideia não é otimização de performance, afinal, na grande maioria das vezes isso conseguimos fazer sem abrir mão de uma ferramenta ORM, como vimos no capítulo 10.

Você acha que seria interessante utilizar uma ferramenta como a JPA que vai gerar os comandos SQL sendo que é justamente isso que você precisa controlar? Apesar de ser possível alterar a geração do SQL das principais ferramentas ORM, provavelmente não é isso que queremos fazer quando iniciamos um projeto.

Usei a JPA para introduzir um exemplo que vai servir para o JSF. Imagine agora que você tem uma aplicação web onde você precisa - ou deseja - controlar o HTML final. Nesse caso, a quantidade de pessoas que precisam disso é muito maior que no exemplo com SQL. Isso porque existem diversos tipos de aplicações, e em cada um a necessidade de controle do HTML final varia muito, como um site, portal etc. A necessidade de controle do HTML visando diversas otimizações de *pagespeed*, layout, acessibilidade e outros requisitos não funcionais é muito grande.

Agora pensando em aplicações web, basicamente composta de formulários complexos - ou mesmo simples -, em que é preciso um controle de estado mais elaborado, com cadastros tão complexos que precisam ser divididos em diversas páginas (como passos de um wizard), a preocupação com esses requisitos não funcionais continuam existindo, mas o mais importante é a funcionalidade.

O fato é que o JSF vai gerar o HTML para você, assim como a JPA gera o SQL. O problema é que às vezes as pessoas lutam muito contra o HTML gerado pelo JSF, enquanto dificilmente lutam contra as SQLs da JPA. Mas veja bem, “não lutar contra” em momento algum significa abrir mão dos requisitos não funcionais já citados. Obviamente, ao usar JPA, não podemos ignorar se estão acontecendo joins desnecessários ou *N+1 queries*.

O que precisamos avaliar é que provavelmente seremos mais felizes se não precisarmos alterar a formatação do comando SQL em si. Da mesma forma, no mesmo capítulo 10, vimos como evitar requisições desnecessárias ao servidor e, quando a requisição for necessária, como evitar o processamento desnecessário por meio do uso eficiente de AJAX.

Mas ainda assim, você será mais feliz se não precisar customizar todo o HTML gerado.

O objetivo do livro foi mostrar tanto JSF quanto JPA desde seu uso básico, até seu funcionamento em situações que antes não entendíamos, ou apelávamos para soluções de contorno. Porém, mostrar como duas especificações tão usadas funcionam em diferentes situações sem deixar de ser prático e objetivo não é tarefa fácil.

Buscamos sempre apresentar as soluções mais utilizadas em vez de escrever cada possibilidade que a especificação e cada implementação proporcionam.

CAPÍTULO 12

Apêndice: iniciando projeto com eclipse

Neste livro tratamos as ferramentas de forma independente de ferramenta, então você pode aplicar facilmente o que for aprendido na sua IDE preferida. No entanto, visando facilitar o início do projeto, veremos como fazer isso utilizando a IDE eclipse. Caso ainda não a tenha em seu computador, baixe a versão “for Java EE Developers” no site do projeto (<http://www.eclipse.org/downloads/>) .

12.1 ADICIONANDO O TOMCAT NO ECLIPSE

Para iniciar a configuração, baixe a última versão do servidor tomcat (<http://tomcat.apache.org/>) e descompacte-o no diretório de sua preferência. Não é necessário realizar nenhuma instalação. Neste livro estamos utilizando o tomcat 7.0.X, mas versões mais novas deverão funcionar também.

No eclipse, vá até a *view* “Servers” e clique no link “new server wizard” como na

imagem a seguir.

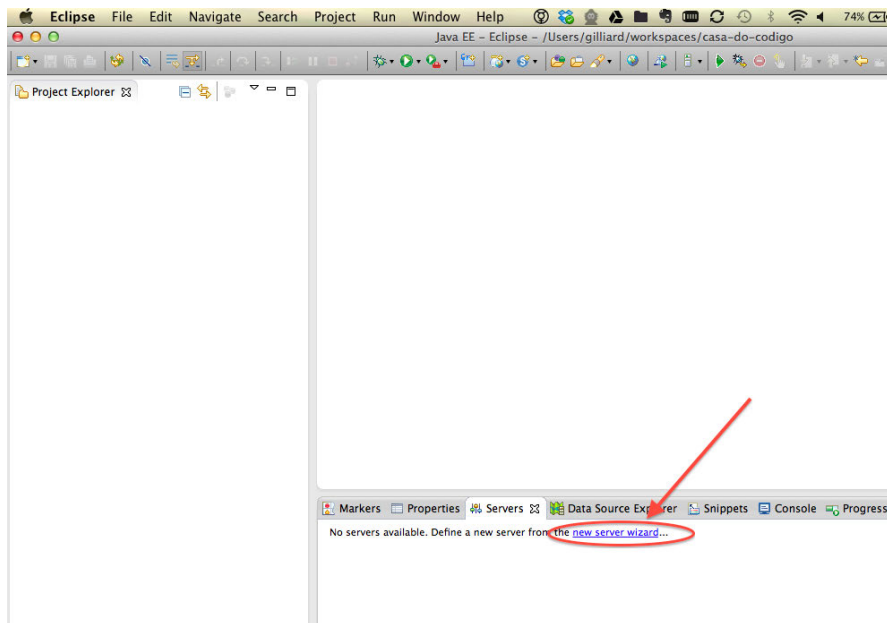


Figura 12.1: Adicionando o tomcat no eclipse

Na próxima tela, selecione o servidor “Tomcat v7.0” ou o da versão adequada ao seu tomcat.

Em seguida, clique no botão “Browse...” e selecione o local onde você descompactou o tomcat.

Ao final, na *view* “Servers” o tomcat está aparecendo.

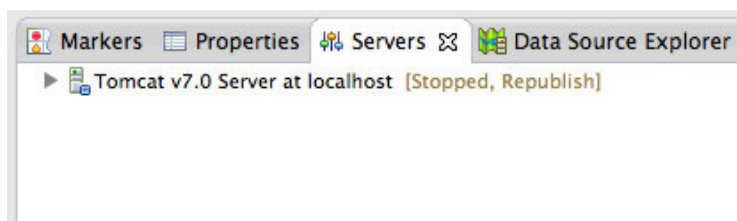


Figura 12.2: Servidor tomcat configurado no eclipse

12.2 CRIANDO O PROJETO WEB

Agora que o eclipse já aponta para o servidor, vamos criar nosso projeto web. Para isso clique com o botão direito na *view* “Project Explorer” e selecione a opção “Dynamic Web Project”.

Após dar o nome do projeto, vamos configurar o JSF no projeto. Para isso basta mudar a opção no combo indicado na figura.

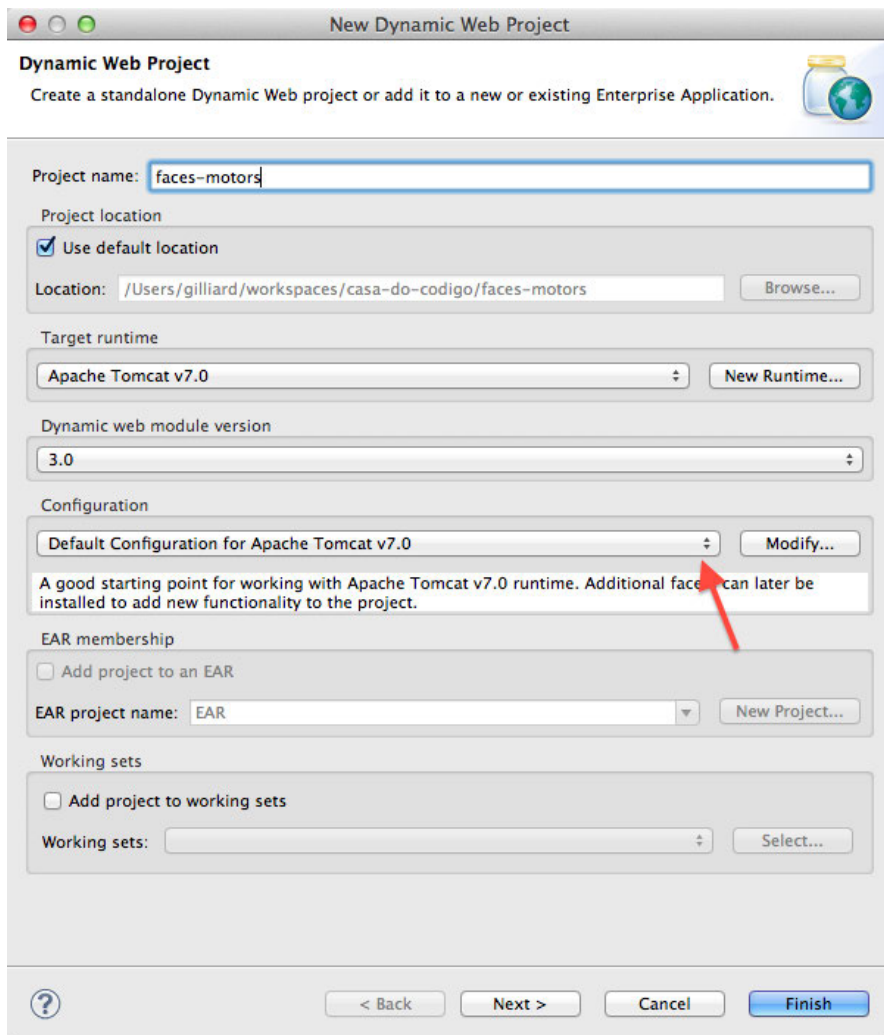


Figura 12.3: Configurando o JSF no projeto

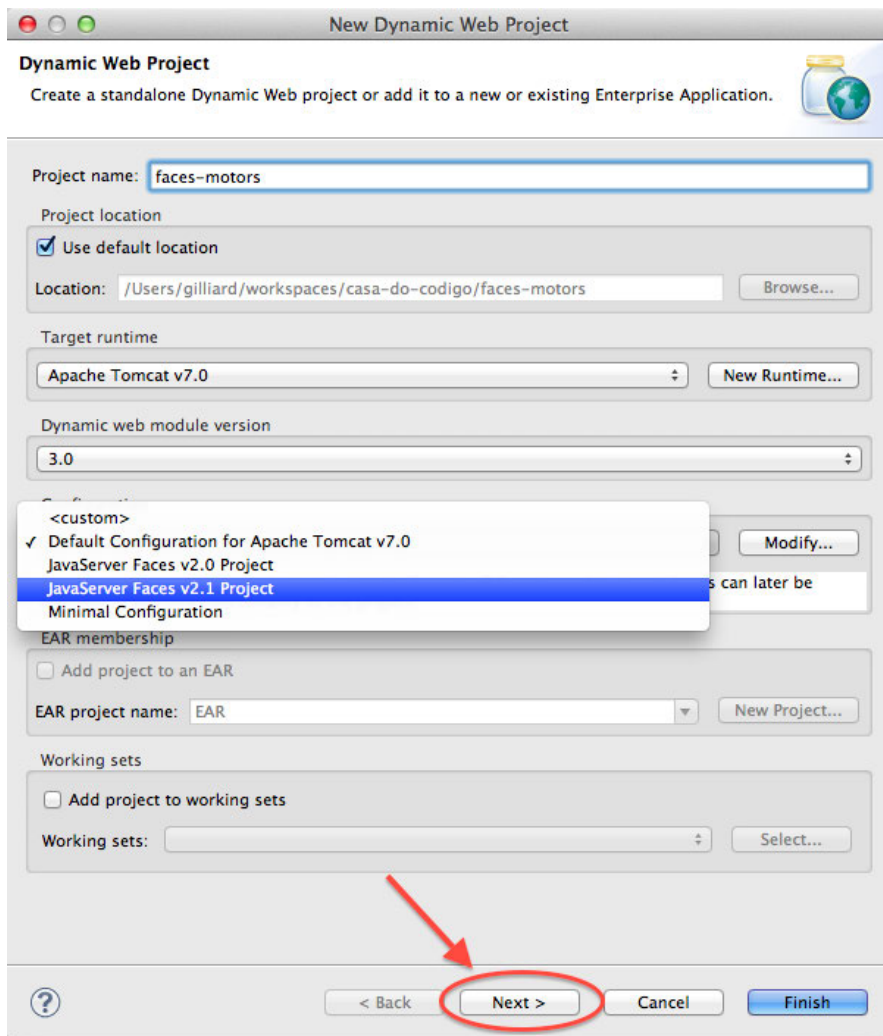


Figura 12.4: Configurando o JSF no projeto

Depois de selecionarmos o JSF, clicamos no botão “Next”, que nos levará para uma tela onde podemos configurar a pasta onde ficarão os fontes e onde ficarão os compilados pelo eclipse. Deixaremos tudo no valor padrão e clicaremos novamente em “Next”.

A tela que aparece em sequência serve para configurarmos onde o eclipse irá obter os jars do JSF. Existe a possibilidade de baixarmos da internet via eclipse, mas em vez disso vamos desabilitar a gerencia dos jars pelo eclipse.

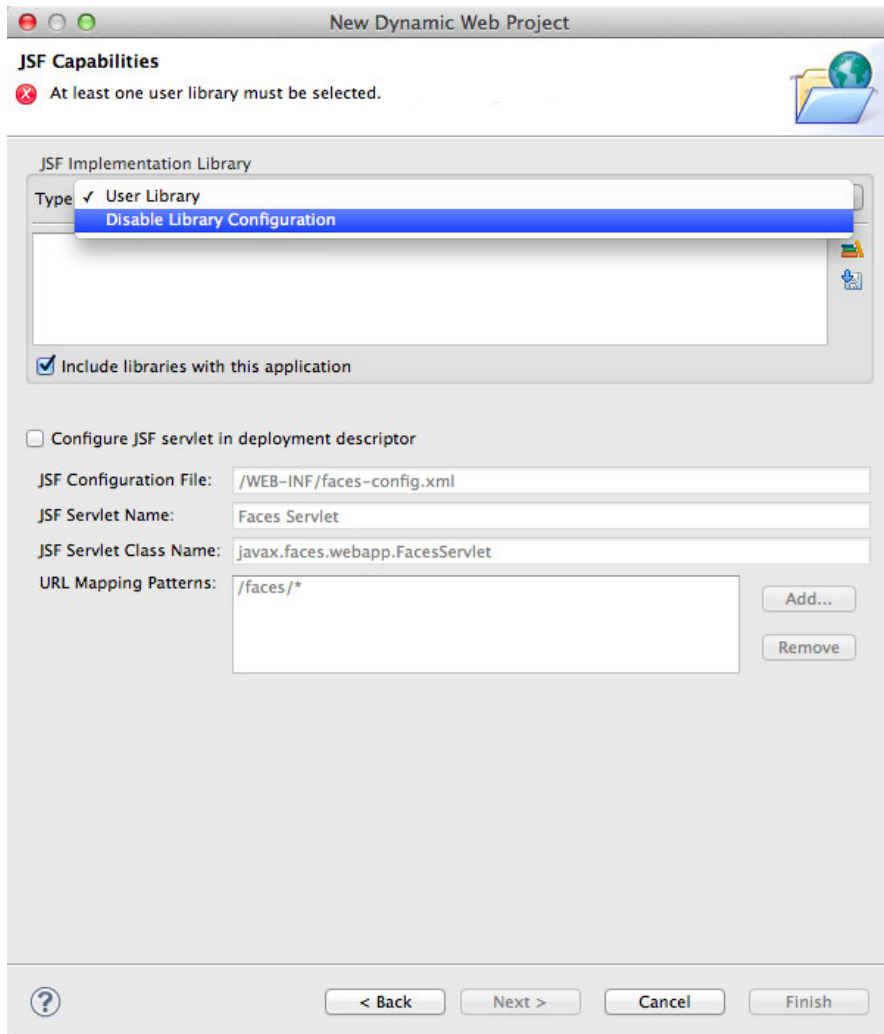


Figura 12.5: Configurando o JSF no projeto

Nesse momento temos um projeto web criado e com a funcionalidade de JSF habilitada. Porém como desabilitamos a gestão dos jars pelo eclipse, precisamos fazer o download manualmente.

12.3 BAIXANDO OS JARS DO JSF

Uma forma fácil de procurar qualquer jar é buscando no repositório central do maven (<http://search.maven.org>).

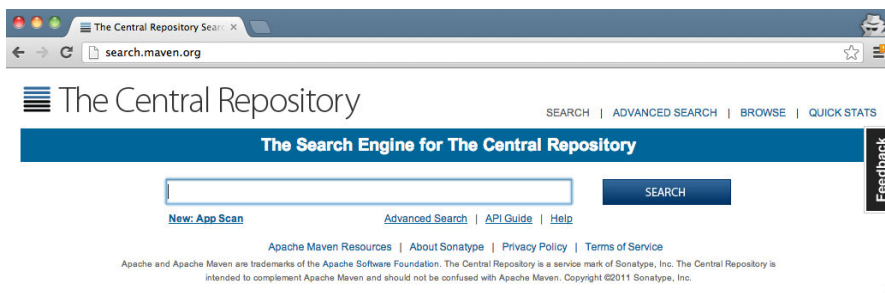


Figura 12.6: Buscando jars do JSF

Na página, basta buscarmos por `com.sun.faces` e várias bibliotecas JSF serão encontradas. Baixe os jars `jsf-impl` e `jsf-api` da versão mais atual.

Depois de baixarmos os jars, devemos colocá-los dentro da pasta `WebContent/WEB-INF/lib` do nosso projeto. Pronto, agora você está preparado para criar seus projetos JSF.

12.4 CONFIGURANDO A JPA

Agora que nosso projeto já tem o JSF funcionando, vamos configurar a JPA. A implementação que utilizaremos é o Hibernate. Para baixá-lo podemos acessar a página de download do Hibernate (<http://hibernate.org/downloads>). Vamos descompactar zip baixado e colocar os jars no projeto.

Depois de baixar, descompactamos o arquivo e copiamos os jars obrigatórios (`required`) e também o da JPA.

Nome	Data de Modific.	Tamanho	Tipo
▼ hibernate-release-4.1.8.Final	Hoje 13:51	--	Pasta
▶ documentation	01/11/2012 02:32	--	Pasta
▼ lib	Hoje 13:51	--	Pasta
▶ envers	01/11/2012 02:32	--	Pasta
▼ jpa	01/11/2012 02:32	--	Pasta
hibernate-entitymanager-4.1.8.Final.jar	01/11/2012 02:13	479 KB	Jar File
▶ optional	01/11/2012 02:32	--	Pasta
▼ required	01/11/2012 02:32	--	Pasta
antlr-2.7.7.jar	31/10/2012 17:28	445 KB	Jar File
dom4j-1.6.1.jar	31/10/2012 17:27	314 KB	Jar File
hibernate-common-...ions-4.0.1.Final.jar	31/10/2012 17:28	81 KB	Jar File
hibernate-core-4.1.8.Final.jar	01/11/2012 02:13	4,5 MB	Jar File
hibernate-jpa-2.0-api-1.0.1.Final.jar	31/10/2012 17:28	103 KB	Jar File
javassist-3.15.0-GA.jar	31/10/2012 17:27	648 KB	Jar File
jboss-logging-3.1.0.GA.jar	31/10/2012 17:28	61 KB	Jar File
jboss-transaction-...spec-1.0.0.Final.jar	31/10/2012 17:28	11 KB	Jar File
▶ project	01/11/2012 02:32	--	Pasta
changelog.txt	01/11/2012 02:09	298 KB	TextW...ument
hibernate_logo.gif	27/09/2012 10:21	1 KB	Graph...t (GIF)
lgpl.txt	27/09/2012 10:21	26 KB	TextW...ument

Figura 12.7: Copiando jars do Hibernate

Depois de copiar os jars, basta colocá-los na pasta `WebContent/WEB-INF/lib` do projeto, e já teremos o hibernate pronto, faltando apenas a configuração do banco de dados, que aprendemos na seção 2.2

O código fonte do livro está disponível no github: <https://github.com/gscordeiro/faces-motors>.

Índice Remissivo

- ABS, [120](#)
- begin, [35](#)
- binding, [47](#)
- body, [43](#)
- commandLink, [61](#)
- commit, [35](#)
- CONCAT, [119](#)
- CURRENT_DATE, [120](#)
- CURRENT_TIME, [120](#)
- CURRENT_TIMESTAMP, [120](#)
- dataTable, [55](#)
- dialect, [31](#)
- Entity, [28](#)
- EntityManager, [34](#)
- EntityManagerFactory, [33](#)
- expression language, [47](#)
- f:facet, [58](#)
- f:setPropertyActionListener, [164](#)
- faces-redirect, [187](#)
- FacesMessage, [180](#)
- Filter, [134](#)
- form, [43](#)
- GeneratedValue, [28](#)
- h:body, [43](#)
- h:column, [56](#)
- h:commandButton, [46](#)
- h:form, [43](#)
- h:inputText, [44](#)
- h:inputTextarea, [44](#)
- h:panelGrid, [45](#)
- hibernate.hbm2ddl.auto, [32](#)
- html, [22](#)
- Id, [28](#)
- Impedância Objeto-Relacional, [5](#)
- Impedance Mismatch, [5](#)
- INDEX, [120](#)
- Java Persistence API, [12](#)
- JBoss, [52](#)
- JDBC, [5](#)
- JPA, [12](#)
- LENGTH, [119](#)
- LOCATE, [119](#)
- LOWER, [119](#)
- MOD, [120](#)
- namespace, [42](#)
- naming container, [149](#)
- navigation-case, [185](#)
- navigation-rule, [185](#)
- Persistence, [34](#)

`persistence-unit`, [29](#)
`persistence.xml`, [29](#)
POJO, [47](#)
`PreparedStatement`, [7](#)
`prependId`, [149](#)

`rendered`, [158](#)
`ResultSet`, [8](#)
`rollback`, [35](#)

`Session`, [11](#)
`SIZE`, [120](#)
`SQRT`, [120](#)
`SUBSTRING`, [119](#)

`Taglib`, [42](#)
`Tomcat`, [52](#)
`Transaction`, [35](#)
`TRIM`, [119](#)

`UPPER`, [119](#)

`Vendor lock-in`, [12](#)

`where`, [118](#)

`xhtml`, [22](#)

Referências Bibliográficas