



K9

TREINAMENTOS

Desenvolvimento Web com JSF2 e JPA2

Desenvolvimento Web com JSF2 e JPA2

28 de agosto de 2012

Sumário	i
Sobre a K19	1
Seguro Treinamento	2
Termo de Uso	3
Cursos	4
1 Banco de dados	1
1.1 Sistemas Gerenciadores de Banco de Dados	1
1.2 MySQL Server	2
1.3 Bases de dados (<i>Databases</i>)	2
1.4 Criando uma base de dados no MySQL Server	2
1.5 Tabelas	3
1.6 Criando tabelas no MySQL Server	3
1.7 CRUD	4
1.8 Chaves Primária e Estrangeira	5
1.9 Exercícios de Fixação	5
1.10 Exercícios Complementares	9
2 JDBC	25
2.1 Driver	26
2.2 JDBC	27
2.3 Instalando o Driver JDBC do MySQL Server	28
2.4 Criando uma conexão	28
2.5 Inserindo registros	29
2.6 Exercícios de Fixação	29
2.7 Exercícios Complementares	31
2.8 SQL Injection	31
2.9 Exercícios de Fixação	32
2.10 Exercícios Complementares	33
2.11 Listando registros	33

2.12 Exercícios de Fixação	34
2.13 Exercícios Complementares	35
2.14 Connection Factory	35
2.15 Exercícios de Fixação	36
2.16 Exercícios Complementares	37
2.17 Desafios	38
3 JPA 2 e Hibernate	39
3.1 Múltiplas sintaxes da linguagem SQL	39
3.2 Orientação a Objetos VS Modelo Relacional	39
3.3 Ferramentas ORM	40
3.4 O que é JPA e Hibernate?	41
3.5 Bibliotecas	41
3.6 Configuração	42
3.7 Mapeamento	42
3.8 Gerando Tabelas	44
3.9 Exercícios de Fixação	44
3.10 Manipulando entidades	46
3.11 Exercícios de Fixação	48
3.12 Repository	49
3.13 Exercícios de Fixação	50
4 Web Container	53
4.1 Necessidades de uma aplicação web	53
4.2 Web Container	55
4.3 Servlet e Java EE	55
4.4 Exercícios de Fixação	55
4.5 Aplicação Web Java	56
4.6 Exercícios de Fixação	57
4.7 Processando requisições	57
4.8 Servlet	57
4.9 Exercícios de Fixação	58
4.10 Frameworks	58
5 Visão Geral do JSF 2	61
5.1 MVC e Front Controller	61
5.2 Configurando uma aplicação JSF	61
5.3 Managed Beans	62
5.4 Processamento de uma requisição	66
5.5 Exemplo Prático	69
5.6 Exercícios de Fixação	71
6 Componentes Visuais	77
6.1 Estrutura Básica de uma Página JSF	77
6.2 Formulários	78
6.3 Caixas de Texto	78
6.4 Campos Ocultos	81
6.5 Caixas de Seleção	81
6.6 Botões e Links	88
6.7 Exercícios de Fixação	90

6.8 Exercícios Complementares	93
6.9 Textos e Imagens	93
6.10 Exercícios de Fixação	95
6.11 Componentes de Organização	96
6.12 Tabelas	97
6.13 Exercícios de Fixação	100
6.14 Exercícios Complementares	102
6.15 Mensagens	102
6.16 Adicionando JavaScript e CSS	102
6.17 Outros Componentes	103
6.18 Exercícios de Fixação	104
6.19 Exercícios Complementares	105
7 Templates e Modularização	107
7.1 Templates	107
7.2 Exercícios de Fixação	110
7.3 Modularização	111
7.4 Exercícios de Fixação	113
7.5 Exercícios Complementares	114
8 Navegação	117
8.1 Navegação Implícita	117
8.2 Navegação Explícita	119
8.3 Exercícios de Fixação	120
8.4 Navegações Estática e Dinâmica	123
8.5 Exercícios de Fixação	125
8.6 Exercícios Complementares	127
9 Escopos	129
9.1 Request	129
9.2 View	130
9.3 Session	131
9.4 Application	133
9.5 Exercícios de Fixação	133
10 Conversão e Validação	139
10.1 Conversão	139
10.2 Conversores Padrão	139
10.3 Exercícios de Fixação	141
10.4 Exercícios Complementares	143
10.5 Mensagens de Erro	143
10.6 Exercícios de Fixação	146
10.7 Validação	146
10.8 Validadores Padrão	147
10.9 Exercícios de Fixação	148
10.10 Exercícios Complementares	150
10.11 Bean Validation	150
10.12 Exercícios de Fixação	153
10.13 Criando o seu Próprio Conversor	154
10.14 Exercícios de Fixação	156

10.15 Exercícios Complementares	158
10.16 Criando o seu Próprio Validador	158
10.17 Exercícios de Fixação	161
10.18 Exercícios Complementares	163
10.19 Criando o seu Próprio Bean Validator	163
10.20 Exercícios de Fixação	164
11 Eventos	167
11.1 FacesEvent	167
11.2 Exercícios de Fixação	170
11.3 PhaseEvent	172
11.4 Exercícios de Fixação	173
11.5 SystemEvent	174
11.6 Exercícios de Fixação	175
11.7 Immediate	177
11.8 Exercícios de Fixação	177
12 Ajax	181
12.1 Fazendo requisições AJAX	181
12.2 Processando uma parte específica da tela	182
12.3 Recarregando parte da tela	183
12.4 Associando um procedimento a uma requisição AJAX	183
12.5 Palavras especiais	183
12.6 Exercícios de Fixação	184
13 Integração JSF e JPA	187
13.1 Bibliotecas	187
13.2 Configuração	187
13.3 Mapeamento	188
13.4 Inicialização e Finalização	188
13.5 Transações	189
13.6 Recuperando o EntityManager da Requisição	190
13.7 Exercícios de Fixação	190
13.8 Otimizando o número de consultas ao SGDB	195
13.9 Exercícios de Fixação	196
A Autenticação	199
A.1 Exercícios de Fixação	199
B Páginas de Erro	205
B.1 Exercícios de Fixação	205
C Projeto Futebol K19	207
C.1 Integração JSF e JPA	207
C.2 Exercícios de Fixação	207
C.3 Modelo	209
C.4 Exercícios de Fixação	209
C.5 Managed Beans	211
C.6 Exercícios de Fixação	211
C.7 Telas	213

C.8 Exercícios de Fixação	213
C.9 Autenticação	220
D Respostas	223







Sobre a K19

A K19 é uma empresa especializada na capacitação de desenvolvedores de software. Sua equipe é composta por profissionais formados em Ciência da Computação pela Universidade de São Paulo (USP) e que possuem vasta experiência em treinamento de profissionais para área de TI.

O principal objetivo da K19 é oferecer treinamentos de máxima qualidade e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos se tornam capacitados para atuar no mercado de trabalho.

Visando a máxima qualidade, a K19 mantém as suas apostilas em constante renovação e melhoria, oferece instalações físicas apropriadas para o ensino e seus instrutores estão sempre atualizados didática e tecnicamente.



Seguro Treinamento

Na K19 o aluno faz o curso quantas vezes quiser!

Comprometida com o aprendizado e com a satisfação dos seus alunos, a K19 é a única que possui o Seguro Treinamento. Ao contratar um curso, o aluno poderá refazê-lo quantas vezes desejar mediante a disponibilidade de vagas e pagamento da franquia do Seguro Treinamento.

As vagas não preenchidas até um dia antes do início de uma turma da K19 serão destinadas ao alunos que desejam utilizar o Seguro Treinamento. O valor da franquia para utilizar o Seguro Treinamento é 10% do valor total do curso.



Termo de Uso

Termo de Uso

Todo o conteúdo desta apostila é propriedade da K19 Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal . Além disso, este material didático pode ser utilizado como material de apoio em cursos de ensino superior desde que a instituição correspondente seja reconhecida pelo MEC (Ministério da Educação) e que a K19 seja citada explicitamente como proprietária do material.

É proibida qualquer utilização desse material que não se enquadre nas condições acima sem o prévio consentimento formal, por escrito, da K19 Treinamentos. O uso indevido está sujeito às medidas legais cabíveis.



Conheça os nossos cursos

-  K01 - Lógica de Programação
-  K11 - Orientação a Objetos em Java
-  K12 - Desenvolvimento Web com JSF2 e JPA2
-  K21 - Persistência com JPA2 e Hibernate
-  K22 - Desenvolvimento Web Avançado com JFS2, EJB3.1 e CDI
-  K23 - Integração de Sistemas com Webservices, JMS e EJB
-  K31 - C# e Orientação a Objetos
-  K32 - Desenvolvimento Web com ASP.NET MVC

www.k19.com.br/cursos

BANCO DE DADOS

Em geral, as aplicações necessitam armazenar dados de forma persistente para consultá-los posteriormente. Por exemplo, a aplicação de uma livraria precisa armazenar os dados dos livros e dos autores de forma persistente.

Suponha que esses dados sejam armazenados em arquivos do sistema operacional. Vários fatores importantes nos levam a descartar tal opção. A seguir, apresentamos as principais dificuldades a serem consideradas na persistência de dados.

Segurança: O acesso às informações potencialmente confidenciais deve ser controlado de forma que apenas usuários e sistemas autorizados possam manipulá-las.

Integridade: Restrições relacionadas aos dados armazenados devem ser respeitadas para que as informações estejam sempre consistentes.

Consulta: O tempo gasto para realizar as consultas aos dados armazenados deve ser o menor possível.

Concorrência: Em geral, diversos sistemas e usuários acessarão concorrentemente as informações armazenadas. Apesar disso, a integridade dos dados deve ser preservada.

Considerando todos esses aspectos, concluímos que um sistema complexo seria necessário para persistir as informações de uma aplicação de maneira adequada. Felizmente, tal tipo de sistema já existe e é conhecido como **Sistema Gerenciador de Banco de Dados** (SGBD).

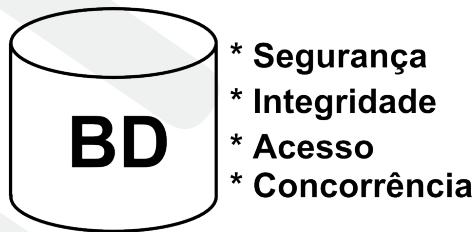


Figura 1.1: Sistema Gerenciador de Banco de Dados

Sistemas Gerenciadores de Banco de Dados

No mercado, há diversas opções de sistemas gerenciadores de banco de dados. Os mais populares são:

- Oracle Database

- SQL Server
- MySQL Server
- PostgreSQL

MySQL Server

Neste treinamento, utilizaremos o MySQL Server, que é mantido pela Oracle e amplamente utilizado em aplicações comerciais. Para instalar o MySQL Server, você pode utilizar o artigo disponível em nosso site: <http://www.k19.com.br/artigos/installando-mysql/>

Bases de dados (*Databases*)

Um sistema gerenciador de banco de dados é capaz de gerenciar informações de diversos sistemas ao mesmo tempo. Por exemplo, as informações dos clientes de um banco, além dos produtos de uma loja virtual ou dos livros de uma livraria.

Suponha que os dados fossem mantidos sem nenhuma separação lógica. Implementar regras de segurança específicas seria extremamente complexo. Tais regras criam restrições quanto ao conteúdo que pode ser acessado por cada usuário. Por exemplo, determinado usuário poderia ter permissão de acesso aos dados dos clientes do banco, mas não às informações dos produtos da loja virtual, ou dos livros da livraria.

Para obter uma organização melhor, os dados são armazenados separadamente em um SGDB. Daí surge o conceito de **base de dados** (database). Uma base de dados é um agrupamento lógico das informações de um determinado domínio.

Criando uma base de dados no MySQL Server

Para criar uma base de dados no MySQL Server, podemos utilizar o comando **CREATE DATABASE**.

```
mysql> CREATE DATABASE livraria;
Query OK, 1 row affected (0.02 sec)
```

Terminal 1.1: Criando uma base de dados.

Podemos utilizar o comando **SHOW DATABASES** para listar as bases de dados existentes.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| livraria          |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.03 sec)
```

Terminal 1.2: Listando as bases de dados existentes.

Repare que, além da base de dados **livraria**, há outras três bases. Essas bases foram criadas automaticamente pelo próprio MySQL Server para teste ou para armazenar configurações.

Quando uma base de dados não é mais necessária, ela pode ser removida através do comando **DROP DATABASE**.

```
mysql> DROP DATABASE livraria;
Query OK, 0 rows affected (0.08 sec)
```

Terminal 1.3: Destruindo uma base de dados.

Tabelas

Um servidor de banco de dados é dividido em bases de dados com o intuito de separar as informações de domínios diferentes. Nessa mesma linha de raciocínio, podemos dividir os dados de uma base a fim de agrupá-los segundo as suas correlações. Essa separação é feita através de **tabelas**. Por exemplo, no sistema de um banco, é interessante separar o saldo e o limite de uma conta, do nome e CPF de um cliente. Então, poderíamos criar uma tabela para os dados relacionados às contas e outra para os dados relacionados aos clientes.

Cliente		
nome	idade	cpf
José	27	31875638735
Maria	32	30045667856

Conta		
numero	saldo	limite
1	1000	500
2	2000	700

Tabela 1.1: Tabelas para armazenar os dados relacionados aos clientes e às contas

Uma tabela é formada por **registros** (linhas) e os registros são formados por **campos** (colunas). Por exemplo, considere uma tabela para armazenar as informações dos clientes de um banco. Cada registro dessa tabela armazena em seus campos os dados de um determinado cliente.

Criando tabelas no MySQL Server

As tabelas no MySQL Server são criadas através do comando **CREATE TABLE**. Na criação de uma tabela, é necessário definir quais são os nomes e os tipos das colunas.

```
mysql> CREATE TABLE 'livraria'.'Livro' (
    -> 'titulo' VARCHAR(255),
    -> 'preco' DOUBLE
    -> )
    -> ENGINE=MyISAM;
Query OK, 0 rows affected (0.14 sec)
```

Terminal 1.4: Criando uma tabela.

As tabelas de uma base de dados podem ser listadas através do comando **SHOW TABLES**. Antes de utilizar esse comando, devemos selecionar uma base de dados através do comando **USE**.

```
mysql> USE livraria;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
```

```
mysql> SHOW TABLES;
+-----+
| Tables_in_livraria |
+-----+
| Livro              |
+-----+
1 row in set (0.00 sec)
```

Terminal 1.5: Listando as tabelas de uma base de dados.

Se uma tabela não for mais desejada, ela pode ser removida através do comando **DROP TABLE**.

```
mysql> DROP TABLE Livro;
Query OK, 0 rows affected (0.00 sec)
```

Terminal 1.6: Destruindo uma tabela.

CRUD

As operações básicas para manipular os dados persistidos são: inserir, ler, alterar e remover.

Essas operações são realizadas através de uma linguagem de consulta denominada **SQL (Structured Query Language)**. Essa linguagem oferece quatro comandos básicos: **INSERT**, **SELECT**, **UPDATE** e **DELETE**. Esses comandos são utilizados para inserir, ler, alterar e remover registros, respectivamente.

```
mysql> INSERT INTO Livro (titulo, preco) VALUES ('Java', 98.75);
Query OK, 1 row affected (0.00 sec)
```

Terminal 1.7: Inserindo um registro.

```
mysql> SELECT * FROM Livro;
+-----+-----+
| titulo | preco |
+-----+-----+
| Java   | 98.75 |
+-----+-----+
1 row in set (0.00 sec)
```

Terminal 1.8: Selecionando registros.

```
mysql> UPDATE Livro SET preco = 115.9 WHERE titulo = 'Java';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1    Changed: 1    Warnings: 0
```

Terminal 1.9: Alterando registros.

```
mysql> SELECT * FROM Livro;
+-----+-----+
| titulo | preco |
+-----+-----+
| Java   | 115.9 |
+-----+-----+
1 row in set (0.00 sec)
```

Terminal 1.10: Selecionando registros.

```
mysql> DELETE FROM Livro WHERE titulo = 'Java';
Query OK, 1 row affected (0.00 sec)
```

Terminal 1.11: Removendo registros.

```
mysql> SELECT * FROM Livro;
Empty set (0.00 sec)
```

Terminal 1.12: Selecionando registros.

Chaves Primária e Estrangeira

Suponha que os livros da nossa livraria sejam classificados por editoras. As editoras possuem nome e telefone. Para armazenar esses dados, uma nova tabela deveria ser criada.

Nesse momento, teríamos duas tabelas (Livro e Editora). Constantemente, a aplicação da livraria deverá descobrir qual é a editora de um determinado livro ou quais são os livros de uma determinada editora. Para isso, os registros da tabela Editora devem estar relacionados aos da tabela Livro.

Na tabela Livro, poderíamos adicionar uma coluna para armazenar o nome da editora dos livros. Dessa forma, se alguém quiser recuperar as informações da editora de um determinado livro, deve consultar a tabela Livro para obter o nome da editora correspondente. Depois, com esse nome, deve consultar a tabela Editora para obter as informações da editora.

Porém, há um problema nessa abordagem. A tabela Editora aceita duas editoras com o mesmo nome. Dessa forma, eventualmente, não conseguiríamos descobrir os dados corretos da editora de um determinado livro. Para resolver esse problema, deveríamos criar uma restrição na tabela Editora que proíba a inserção de editoras com o mesmo nome.

Para resolver esse problema no MySQL Server, poderíamos adicionar a propriedade **UNIQUE** no campo nome da tabela Editora. Porém, ainda teríamos mais um problema. Na tabela Livro, poderíamos adicionar registros vinculados a editoras inexistentes, pois não há nenhuma relação explícita entre as tabelas. Para solucionar esses problemas, devemos utilizar o conceito de **chave primária** e **chave estrangeira**.

Toda tabela pode ter uma chave primária, que é um conjunto de um ou mais campos que devem ser únicos para cada registro. Normalmente, um campo numérico é escolhido para ser a chave primária de uma tabela, pois as consultas podem ser realizadas com melhor desempenho.

Então, poderíamos adicionar um campo numérico na tabela Editora e torná-lo chave primária. Vamos chamar esse campo de **id**. Na tabela Livro, podemos adicionar um campo numérico chamado **editora_id** que deve ser utilizado para guardar o valor da chave primária da editora correspondente ao livro. Além disso, o campo **editora_id** deve estar explicitamente vinculado com o campo **id** da tabela Editora. Para estabelecer esse vínculo, o campo **editora_id** da tabela Livro deve ser uma chave estrangeira associada à chave primária da tabela Editora.

Uma chave estrangeira é um conjunto de uma ou mais colunas de uma tabela que possuem valores iguais aos da chave primária de outra tabela.

Com a definição da chave estrangeira, um livro não pode ser inserido com o valor do campo **editora_id** inválido. Caso tentássemos fazer isso, obteríamos uma mensagem de erro.



Exercícios de Fixação

- 1 Abra um terminal, crie e acesse uma pasta com o seu nome.

```
cosen@k19:~$ mkdir rafael
cosen@k19:~$ cd rafael/
cosen@k19:~/rafael$
```

Terminal 1.13: Criando e acessando uma pasta com o seu nome.

- 2 Estando dentro da sua pasta, acesse o MySQL Server utilizando o usuário **root** e a senha **root**.

```
k19@k19-11:~/rafael$ mysql -u root -p
Enter password:
```

Terminal 1.14: Logando no MySQL Server.

- 3 Caso exista uma base de dados chamada **livraria**, remova-a. Utilize o comando **SHOW DATABASES** para listar as bases de dados existentes e o comando **DROP DATABASE** para remover a base **livraria** se ela existir.

```
mysql> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| livraria      |
| mysql          |
| test           |
+-----+
4 rows in set (0.00 sec)

mysql> DROP DATABASE livraria;
Query OK, 1 row affected (0.12 sec)
```

*Terminal 1.15: Listando as bases de dados existentes e removendo a base **livraria**.*

- 4 Crie uma nova base de dados chamada **livraria**. Utilize o comando **CREATE DATABASE**. Você vai utilizar esta base nos exercícios seguintes.

```
mysql> CREATE DATABASE livraria;
Query OK, 1 row affected (0.00 sec)
```

*Terminal 1.16: Criando a base **livraria**.*

- 5 Abra um editor de texto e digite o código abaixo para criar uma tabela com o nome **Editora**. Depois salve o arquivo com o nome **create-table-editora.sql** dentro da pasta com o seu nome.

```
1 USE livraria;
2 CREATE TABLE Editora (
3     id BIGINT NOT NULL AUTO_INCREMENT,
4     nome VARCHAR (255) NOT NULL,
5     email VARCHAR (255) NOT NULL,
6     PRIMARY KEY (id)
7 )
8 ENGINE = InnoDB;
```

*Código SQL 1.1: Criando a tabela **Editora***

- 6 Dentro do terminal, use o comando **source** para executar o arquivo que você acabou de criar.

```
mysql> source create-table-editora.sql
Database changed
```

```
Query OK, 0 rows affected (0.08 sec)
```

Terminal 1.17: Executando a tabela Editora.

- 7 Abra um novo editor de texto e digite o código abaixo para criar uma tabela com o nome **Livro**. Em seguida, salve o arquivo com o nome `create-table-livro.sql` dentro da pasta com o seu nome.

```
1 USE livraria;
2 CREATE TABLE Livro (
3     id BIGINT NOT NULL AUTO_INCREMENT ,
4     titulo VARCHAR(255) NOT NULL ,
5     preco DOUBLE NOT NULL ,
6     editora_id BIGINT NOT NULL ,
7     PRIMARY KEY(id),
8     CONSTRAINT fk_editora FOREIGN KEY fk_editora(editora_id)
9     REFERENCES Editora(id)
10    ON DELETE RESTRICT
11    ON UPDATE RESTRICT
12 )
13 ENGINE = InnoDB;
```

Código SQL 1.2: Criando a tabela Livro

- 8 Dentro do terminal, use o comando `source` para executar o código do arquivo `create-table-livro.sql`.

```
mysql> source create-table-livro.sql
Database changed
Query OK, 0 rows affected (0.08 sec)
```

Terminal 1.18: Executando a tabela Livro.

- 9 Abra um novo editor de texto e digite o código abaixo para adicionar alguns registros na tabela **Editora**. Depois salve o arquivo com o nome `adicionando-registros-editora.sql` dentro da pasta com o seu nome.

```
1 INSERT INTO Editora (nome, email) VALUES ('Oreilly', 'oreilly@email.com');
2
3 INSERT INTO Editora (nome, email) VALUES ('Wrox', 'wrox@email.com');
4
5 INSERT INTO Editora (nome, email) VALUES ('Apress', 'apress@email.com');
```

Código SQL 1.3: Adicionando registros na tabela Editora

- 10 Dentro do terminal, execute o arquivo que você acabou de criar para adicionar alguns registro na tabela **Editora**.

```
mysql> source adicionando-registros-editora.sql
Query OK, 1 row affected (0.03 sec)

Query OK, 1 row affected (0.04 sec)

Query OK, 1 row affected (0.04 sec)
```

Terminal 1.19: Inserindo editoras.

- 11 Abra um novo editor de texto e digite o código abaixo para adicionar alguns registros na tabela **Livro**. Depois salve o arquivo com o nome `adicionando-registros-livro.sql` dentro da pasta com o seu nome.

```
1 INSERT INTO Livro (titulo, preco, editora_id) VALUES ('Aprendendo C#', 89.90, 1);
2
```

```

3 INSERT INTO Livro (titulo, preco, editora_id) VALUES ('Introdução ao JSF 2',
4 122.90, 3);
5
6 INSERT INTO Livro (titulo, preco, editora_id) VALUES ('JSF 2 Avançado', 149.90, 3);

```

Código SQL 1.4: Adicionando alguns registros na tabela Livro

- 12 Dentro do terminal, execute o arquivo que você acabou de criar para adicionar alguns registros na **Livro**.

```

mysql> source adicionando-registros-livro.sql
Query OK, 1 row affected (0.02 sec)

Query OK, 1 row affected (0.04 sec)

Query OK, 1 row affected (0.04 sec)

```

Terminal 1.20: Inserindo livros.

- 13 Consulte os registros da tabela Editora e da tabela Livro. Utilize o comando **SELECT**.

```

mysql> SELECT * FROM Editora;
+----+-----+-----+
| id | nome      | email           |
+----+-----+-----+
| 1  | Oreilly   | oreilly@email.com |
| 2  | Wrox      | wrox@email.com  |
| 3  | Apress    | apress@email.com |
+----+-----+-----+
3 rows in set (0.00 sec)

```

Terminal 1.21: Selecionando as editoras.

```

mysql> SELECT * FROM Livro;
+----+-----+-----+-----+
| id | titulo          | preco | editora_id |
+----+-----+-----+-----+
| 1  | Aprendendo C#  | 89.9  | 1        |
| 2  | Introdução ao JSF 2 | 122.9 | 3        |
| 3  | JSF 2 Avançado | 149.9 | 3        |
+----+-----+-----+-----+
3 rows in set (0.00 sec)

```

Terminal 1.22: Selecionando os livros.

- 14 Altere alguns dos registros da tabela Livro. Utilize o comando **UPDATE**.

```

mysql> UPDATE Livro SET preco=92.9 WHERE id=1;
Query OK, 1 row affected (0.07 sec)
Rows matched: 1  Changed: 1  Warnings: 0

```

Terminal 1.23: Alterando livros.

- 15 Altere alguns dos registros da tabela Editora. Utilize o comando **UPDATE**.

```

mysql> UPDATE Editora SET nome='OReilly' WHERE id=1;
Query OK, 1 row affected (0.09 sec)
Rows matched: 1  Changed: 1  Warnings: 0

```

Terminal 1.24: Alterando editoras.

- 16 Remova alguns registros da tabela Livro. Utilize o comando **DELETE**.

```

mysql> DELETE FROM Livro WHERE id=2;
Query OK, 1 row affected (0.07 sec)

```

Terminal 1.25: Removendo livros.

- 17 Remova alguns registros da tabela Editora. Preste atenção para não remover uma editora que tenha algum livro relacionado já adicionado no banco. Utilize o comando **DELETE**.

```
mysql> DELETE FROM Editora WHERE id=2;
Query OK, 1 row affected (0.05 sec)
```

Terminal 1.26: Removendo editoras.

- 18 Faça uma consulta para buscar todos os livros de uma determinada editora.

```
mysql> SELECT * FROM Livro as L, Editora as E WHERE L.editora_id=E.id and E.id = 1;
+----+-----+-----+-----+-----+
| id | titulo | preco | editora_id | id | nome   | email      |
+----+-----+-----+-----+-----+
| 1  | Aprendendo C# | 92.9 | 1 | 1 | OReilly | oreilly@email.com |
+----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Terminal 1.27: Selecionando os livros de uma editora.



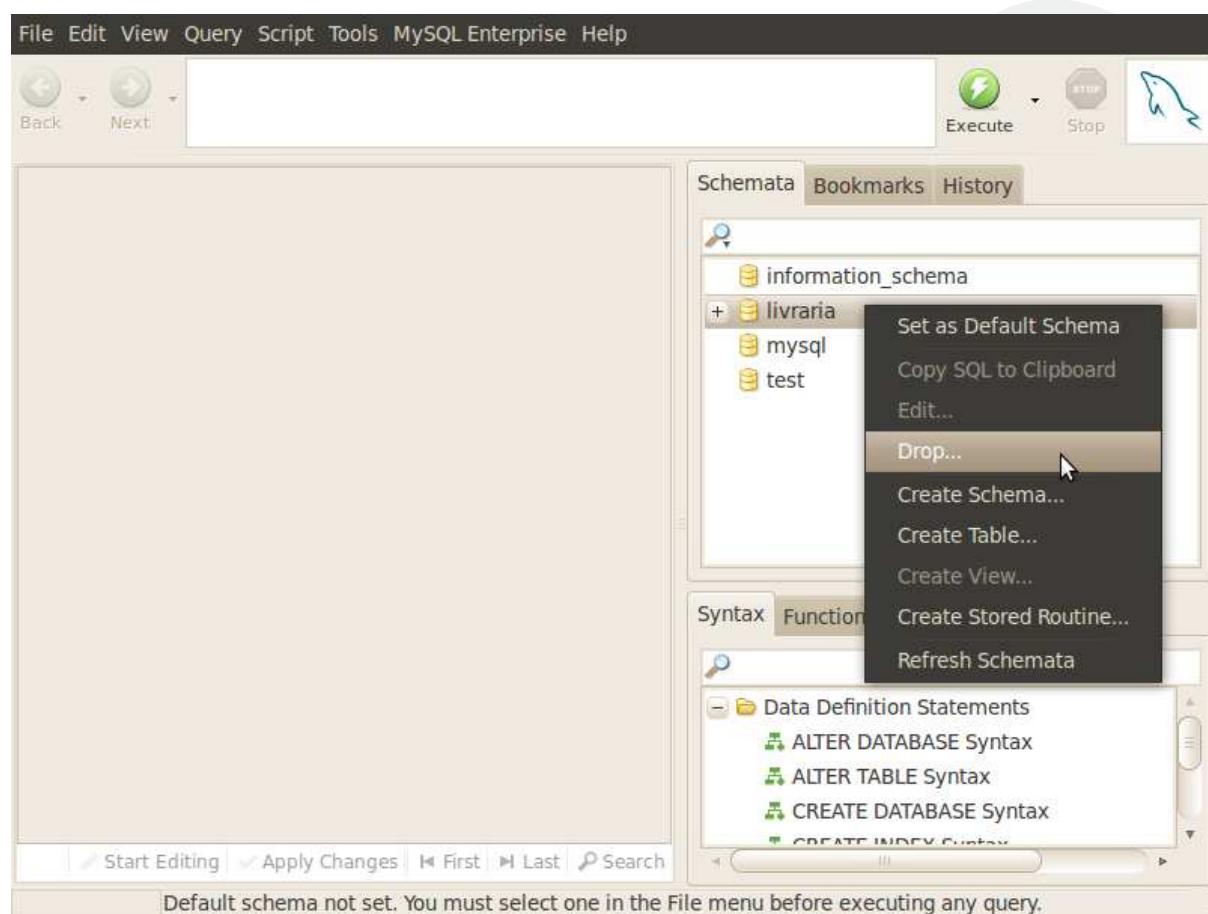
Exercícios Complementares

Utilize o MySQL Query Browser para refazer os exercícios anteriores.

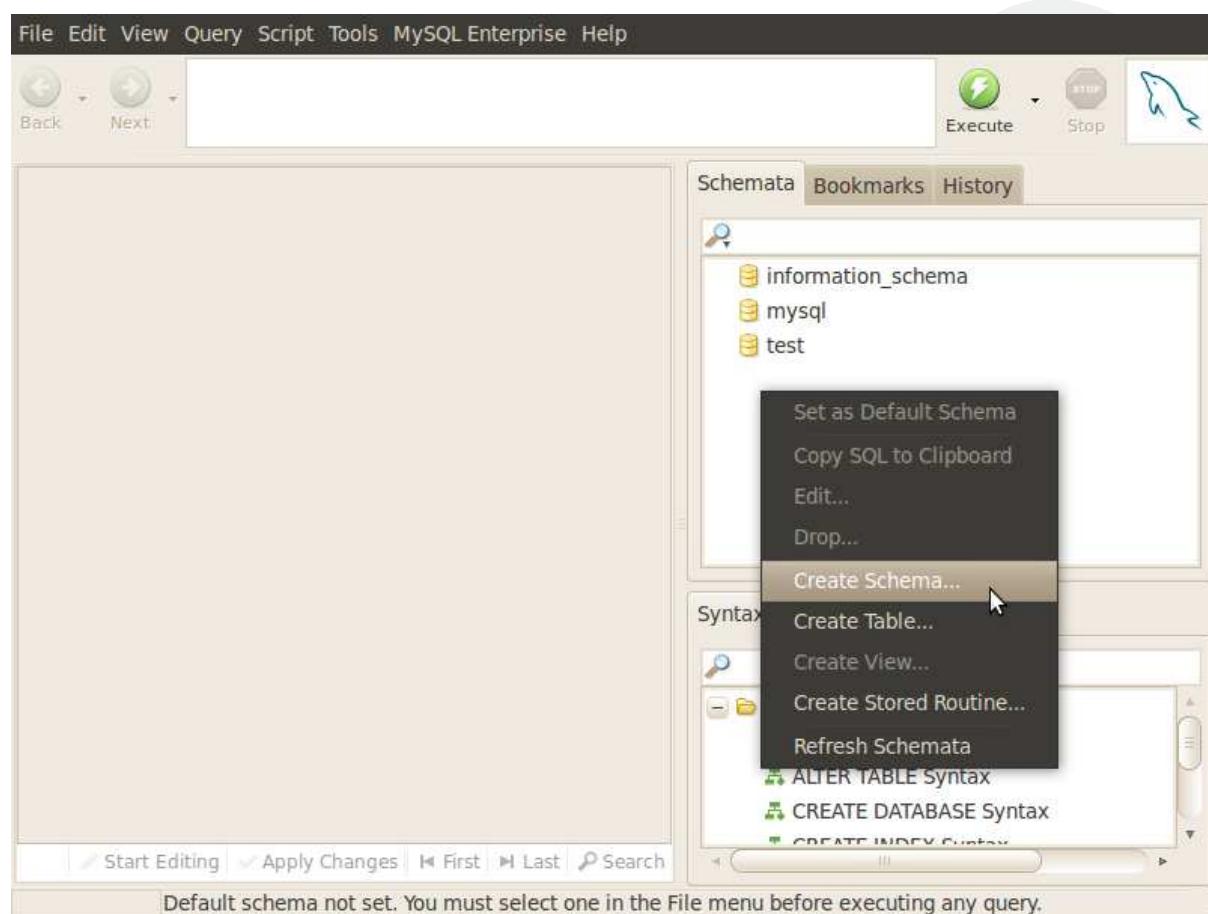
- 1 Abra o **MySQL Query Browser** utilizando **localhost** como Server Hostname, **root** como Username e **root** como Password.



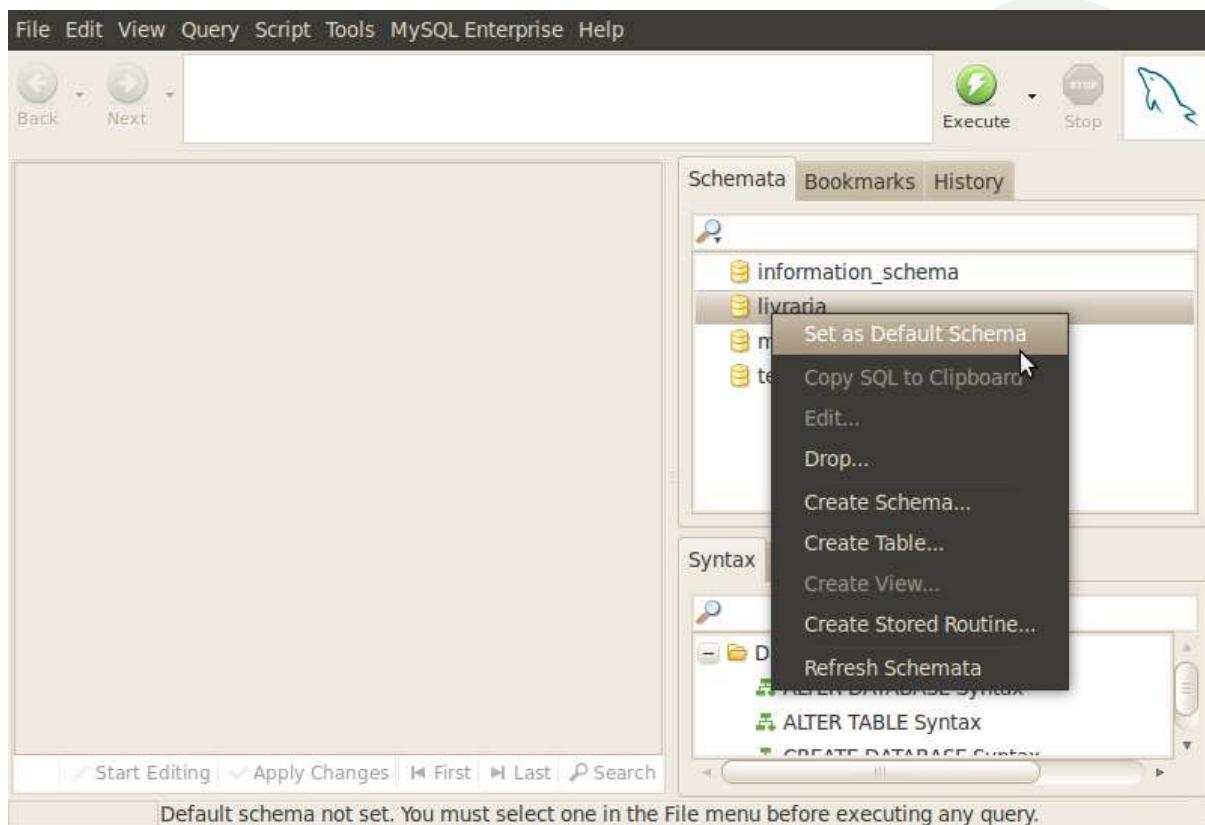
- 2 Caso exista uma base de dados chamada **livraria**, remova-a conforme a figura abaixo.



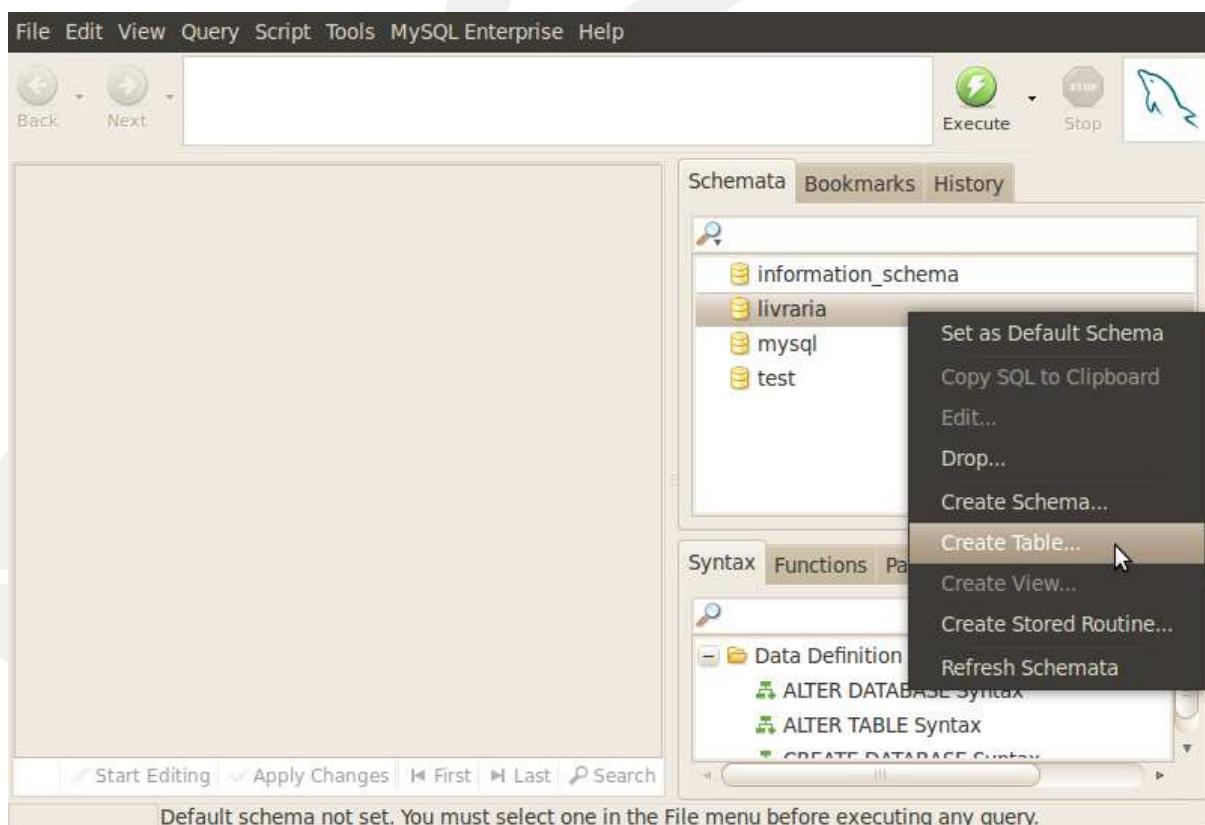
- 3 Crie uma nova base de dados chamada **livraria**, conforme mostrado na figura abaixo. Você vai utilizar esta base nos exercícios seguintes.



- 4 Seleione a base de dados livraria como padrão.



- 5 Crie uma tabela chamada **Editora** conforme as figuras abaixo.



Altere o modo de criação da tabela para **InnoDB**, conforme mostrado na figura.



Crie os campos conforme a figura e não esqueça de tornar todos os campos obrigatórios, marcando a opção **NOT NULL**. Além disso, o campo **id** deve ser uma chave primária e automaticamente incrementada.

Table Name: Editora Comment:

Columns and Indices Table Options Advanced Options

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value	Comments
id	BIGINT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			
nome	VARCHAR(255)	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
email	VARCHAR(255)	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
		<input checked="" type="checkbox"/>	<input type="checkbox"/>			

Column Details Indices Foreign Keys

Name: id Data Type: BIGINT Default Value: NULL

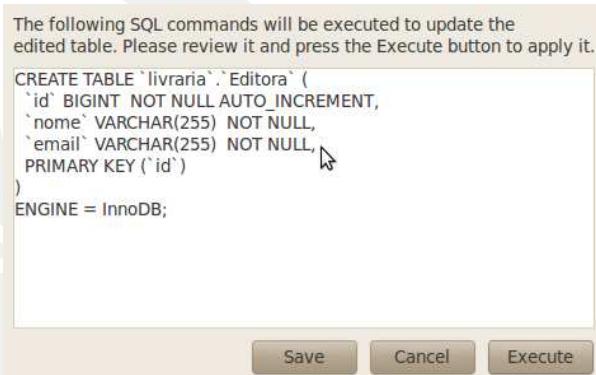
Column Options Flags: UNSIGNED ZEROFILL Character Set: Collation:

Primary Key
 Not NULL
 Auto Increment

Comment:

Discard Changes Apply Changes Close

Após clicar no botão “Apply Changes”, aparecerá uma janela mostrando os comandos SQL gerados. Clique no botão “Execute”.



- 6 Crie uma tabela chamada **Livro** conforme as figuras abaixo. Altere o modo de criação da tabela para **InnoDB**, conforme mostrado na figura.

Table Name: Livro Comment:

Columns and Indices Table Options Advanced Options

Storage Engine: InnoDB

Supports transactions, row-level locking, and foreign keys

Default Character Set

Character Set: The default character set that is used for the table. Starting from MySQL 4.1 you can specify an individual character set for each column.

Collation: Collation method that is used to compare text and sort columns.

Novamente, adicione os campos conforme a figura abaixo, lembrando de marcar a opção **NOT NULL**. Além disso, o campo **id** deve ser uma chave primária e automaticamente incrementada.

Table Name: Livro Comment:

Columns and Indices Table Options Advanced Options

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value	Comments
id	BIGINT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			
titulo	VARCHAR(255)	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
preco	DOUBLE	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
editora_id	BIGINT	<input checked="" type="checkbox"/>	<input type="checkbox"/>			

Column Details Indices Foreign Keys

Name: id Data Type: BIGINT Default Value: NULL

Column Options Flags: UNSIGNED ZEROFILL Character Set:

Primary Key Not NULL Auto Increment Comment:

Character Set: Collation:

Discard Changes Apply Changes Close

Você precisa tornar o campo **editora_id** uma chave estrangeira. Selecione a aba **Foreign Keys** e clique no botão com o símbolo “+” para adicionar uma chave estrangeira. Depois, siga os procedimentos conforme mostrados na figura abaixo.

Table Name: Livro Comment:

Columns and Indices Table Options Advanced Options

Column Name	Data Type	NOT NULL	AUTO INC	Flags	Default Value	Comments
id	BIGINT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			
titulo	VARCHAR(255)	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
preco	DOUBLE	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
editora_id	BIGINT	<input checked="" type="checkbox"/>	<input type="checkbox"/>			

Column Details Indices Foreign Keys

Foreign Key Settings

Key Name: fk_editora
On Delete: RESTRICT
On Update: RESTRICT
Refer. Table: Editora

Column	Foreign Column
editora_id	id

Use drag and drop to add columns to the list above.

Discard Changes Apply Changes Close

The following SQL commands will be executed to update the edited table. Please review it and press the Execute button to apply it.

```
CREATE TABLE `livraria`.`Livro` (
  `id` BIGINT NOT NULL AUTO_INCREMENT,
  `titulo` VARCHAR(255) NOT NULL,
  `preco` DOUBLE NOT NULL,
  `editora_id` BIGINT NOT NULL,
  PRIMARY KEY (`id`),
  CONSTRAINT `fk_editora` FOREIGN KEY `fk_editora` (`editora_id`)
    REFERENCES `Editora` (`id`)
    ON DELETE RESTRICT
    ON UPDATE RESTRICT
)
ENGINE = InnoDB;
```

Save Cancel Execute

- 7 Adicione alguns registros na tabela Editora. Veja exemplos na figura abaixo.

The screenshot shows the MySQL Enterprise Workbench interface. In the top menu bar, the following items are visible: File, Edit, View, Query, Script, Tools, MySQL Enterprise, and Help. Below the menu is a toolbar with icons for Undo, Redo, Open, Save, Continue, Step, Execute, Stop, and a magnifying glass. The main window contains a query editor titled "Resultset 1" and "New Script". The query editor displays the following SQL code:

```
1 • INSERT INTO Editora (nome, email)
2   VALUES ('Oreilly', 'oreilly@email.com');
3
4 • INSERT INTO Editora (nome, email)
5   VALUES ('Wrox', 'wrox@email.com');
6
7 INSERT INTO Editora (nome, email)
8   VALUES ('Apress', 'apress@email.com');
```

To the right of the query editor is a "Schemata" browser pane showing the database structure. It lists the databases: information_schema, livraria, mysql, and test. Under the "livraria" database, there are tables for Editora and Livro. Below the schemata browser is a "Syntax" pane with sections for Data Definition Statements, ALTER DATABASE Syntax, ALTER TABLE Syntax, and CREATE DATABASE Syntax.

- 8 Adicione alguns registros na tabela Livro. Veja exemplos na figura abaixo.

The screenshot shows the MySQL Enterprise Workbench interface. In the top menu bar, the following items are visible: File, Edit, View, Query, Script, Tools, MySQL Enterprise, and Help. Below the menu is a toolbar with icons for Undo, Redo, Open, Save, Continue, Step, Execute, and Stop. A status bar at the bottom shows the time as 11:36.

The main area contains two tabs: "Resultset 1" and "New Script". The "Resultset 1" tab displays the following SQL code:

```
1 • INSERT INTO Livro (titulo, preco, editora_id)
2   VALUES ('Aprendendo C#', 89.90, 1);
3
4 • INSERT INTO Livro (titulo, preco, editora_id)
5   VALUES ('Introdução ao JSF 2', 122.90, 3);
6
7 • INSERT INTO Livro (titulo, preco, editora_id)
8   VALUES ('JSF 2 Avançado', 149.90, 3);
```

To the right of the code editor is a "Schemata" browser window. It lists the databases: information_schema, livraria, mysql, and test. The "livraria" database is expanded, showing its tables: Editora and Livro.

Below the schemata browser is a "Syntax" panel containing links to Data Definition Statements, ALTER DATABASE Syntax, ALTER TABLE Syntax, and CREATE DATABASE Syntax.

- 9 Consulte os registros da tabela Editora e, em seguida, consulte a tabela Livro. Veja exemplos logo abaixo.

File Edit View Query Script Tools MySQL Enterprise Help

`SELECT * FROM Editora;`

Back Next Execute Stop

Resultset 1 New Script

	id	nome	email
1	Oreilly	oreilly@email.com	
2	Wrox	wrox@email.com	
3	Apress	apress@email.com	

3 rows fetched in 0:00.0240 Start Editing Apply Changes First Last Search

Query finished.

Schemata Bookmarks History

- information_schema
- livraria
 - + Editora
 - + Livro
- mysql
- test

Syntax Functions Params Trx

- Data Definition Statement
 - ALTER DATABASE Syntax
 - ALTER TABLE Syntax
 - CREATE DATABASE Syntax

File Edit View Query Script Tools MySQL Enterprise Help

`SELECT * FROM Livro;`

Back Next Execute Stop

Resultset 1 New Script

	id	titulo	preco	editora_id
1	Aprendendo C#	89.9	1	
2	Introdução ao JSF 2	122.9	3	
3	JSF 2 Avançado	149.9	3	

3 rows fetched in 0:00.0518 Start Editing Apply Changes First Last Search

Query finished.

Schemata Bookmarks History

- information_schema
- livraria
 - + Editora
 - + Livro
- mysql
- test

Syntax Functions Params Trx

- Data Definition Statement
 - ALTER DATABASE Syntax
 - ALTER TABLE Syntax
 - CREATE DATABASE Syntax

- 10 Altere alguns dos registros da tabela Livro. Veja o exemplo abaixo.

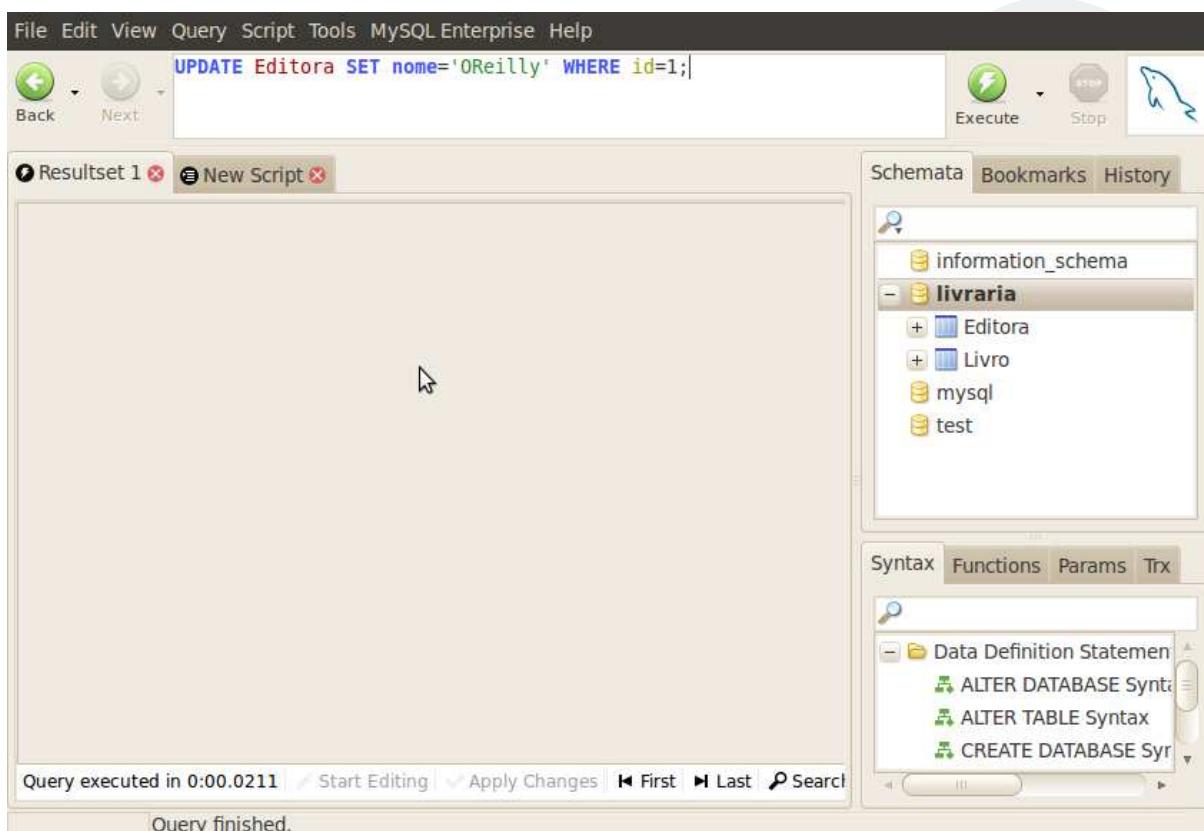
The screenshot shows the MySQL Workbench interface. The top menu bar includes File, Edit, View, Query, Script, Tools, MySQL Enterprise, and Help. The main area contains a query editor with the following SQL statement:

```
UPDATE Livro SET preco=92.9 WHERE id=1;
```

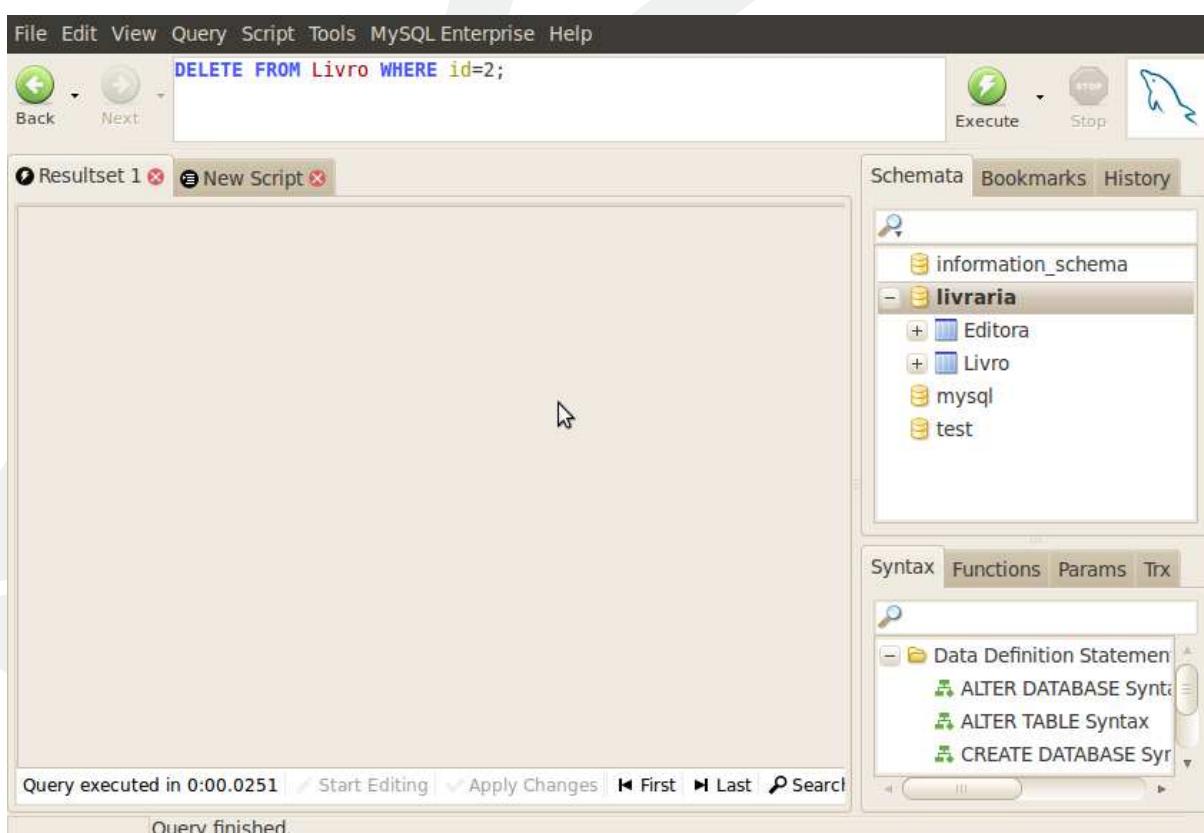
Below the query editor are two tabs: Resultset 1 and New Script. To the right of the query editor are three buttons: Back, Next, Execute, and Stop. The Execute button is highlighted with a green lightning bolt icon. The bottom status bar displays the message "Query finished."

The right side of the interface features a sidebar titled "Schemata" which lists databases: information_schema, livraria (selected), Editora, Livro, mysql, and test. Below the schemata is a "Syntax" tab containing links to Data Definition Statements: ALTER DATABASE Syntax, ALTER TABLE Syntax, and CREATE DATABASE Syntax.

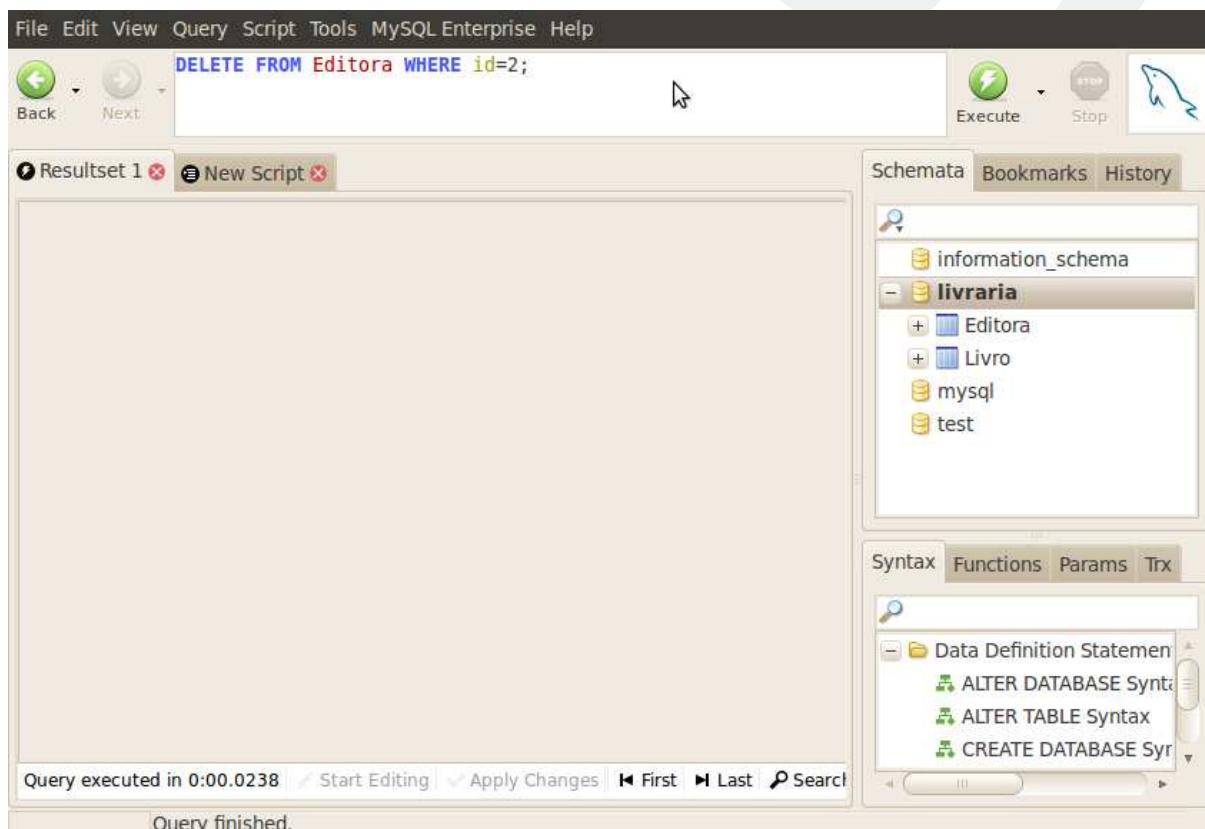
- 11 Altere alguns dos registros da tabela Editora. Veja o exemplo abaixo.



- 12 Remova alguns registros da tabela Livro. Veja o exemplo abaixo.



- 13) Remova alguns registros da tabela Editora. Preste atenção para não remover uma editora que tenha algum livro relacionado já adicionado no banco. Veja o exemplo abaixo:



- 14) Faça uma consulta para buscar todos os livros associados as suas respectivas editoras. Veja um exemplo na figura abaixo.

The screenshot shows the MySQL Workbench interface. At the top, there's a menu bar with File, Edit, View, Query, Script, Tools, MySQL Enterprise, and Help. Below the menu is a toolbar with Back, Next, Execute, and Stop buttons. The main area contains a query editor window with the following SQL code:

```
SELECT * FROM Livro as L, Editora as E
WHERE L.editora_id=E.id;
```

Below the query editor is a result set titled "Resultset 1" showing two rows of data from the "Livro" table:

	id	titulo	preco	editora_id	id	nome	email
1	Aprendendo C#	92.9	1	1	O'Reilly	oreilly@email.com	
3	JSF 2 Avançado	149.9	3	3	Apress	apress@email.com	

On the right side, there's a sidebar with tabs for Schemata, Bookmarks, and History. The Schemata tab is selected, showing the database structure with "livraria" expanded to show "Editora" and "Livro". Below the sidebar is a Syntax tab with sections for Data Definition Statements, ALTER DATABASE Syntax, ALTER TABLE Syntax, and CREATE DATABASE Syntax.

JDBC

No capítulo anterior, aprendemos que utilizar bancos de dados é uma ótima alternativa para armazenar os dados de uma aplicação. Entretanto, você deve ter percebido que as interfaces disponíveis para interagir com o MySQL Server não podem ser utilizadas por qualquer pessoa. Para utilizá-las, é necessário conhecer a linguagem SQL e os conceitos do modelo relacional. Em geral, as interfaces dos outros SGDBs exigem os mesmos conhecimentos.

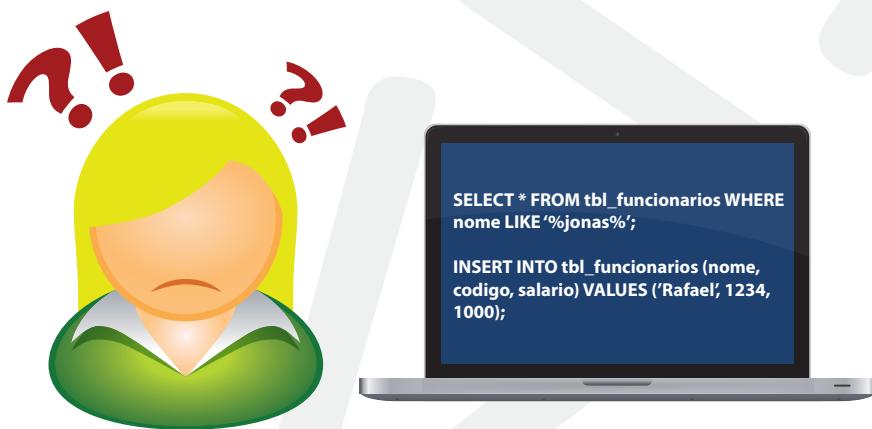


Figura 2.1: Usuários comuns não possuem conhecimento sobre SQL ou sobre o modelo relacional

Para resolver esse problema, podemos desenvolver aplicações com interfaces que não exijam conhecimentos técnicos de SQL ou do modelo relacional para serem utilizadas. Dessa forma, usuários comuns poderiam manipular as informações do banco de dados através dessas aplicações. Nessa abordagem, os usuários interagem com as aplicações e as aplicações interagem com os SGDBs.



Figura 2.2: Usuários comuns devem utilizar interfaces simples

Driver

As aplicações interagem com os SGDBs através de troca de mensagens. Os SGDBs definem o formato das mensagens. Para não sobrecarregar o canal de comunicação entre as aplicações e os SGDBs, as mensagens trocadas devem ocupar o menor espaço possível. Geralmente, protocolos binários são mais apropriados para reduzir o tamanho das mensagens e consequentemente diminuir a carga do canal de comunicação. Por isso, os SGDBs utilizam protocolos binários.

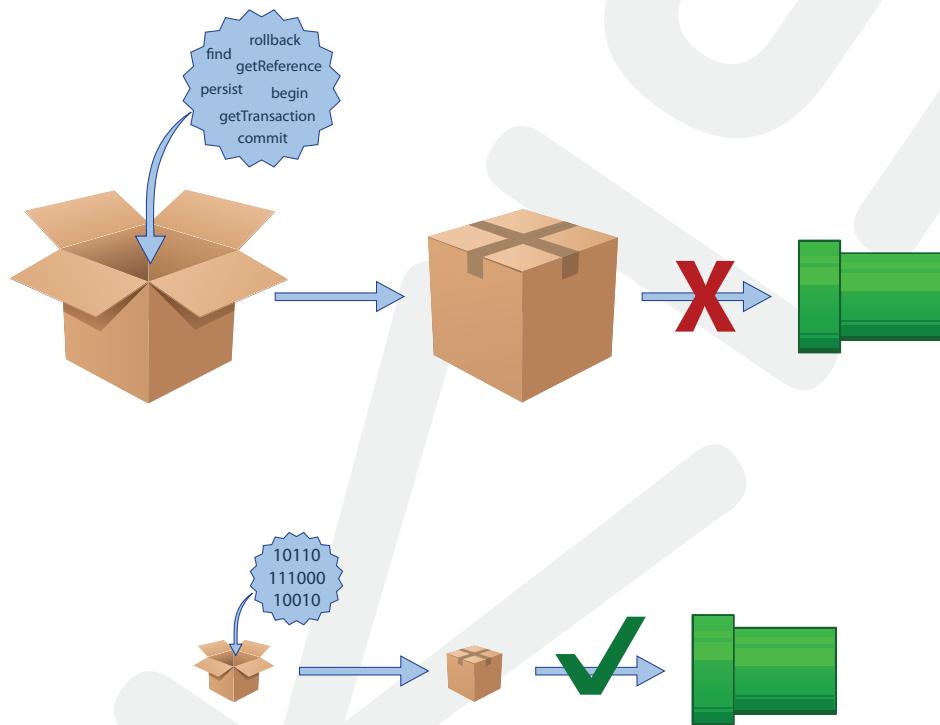


Figura 2.3: Diminuindo o tamanho das mensagens para não sobrecarregar o meio de comunicação

Mensagens binárias são facilmente interpretadas por computadores. Por outro lado, são complexas para um ser humano compreender. Dessa forma, o trabalho dos desenvolvedores seria muito complexo, aumentando o custo para o desenvolvimento e manutenção das aplicações.

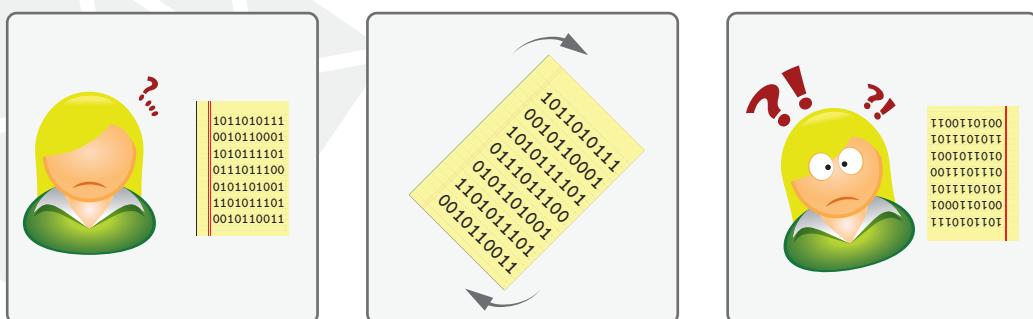


Figura 2.4: Mensagens binárias são altamente complexas para os seres humanos

Para resolver esse problema e facilitar o desenvolvimento das aplicações, as empresas proprietárias dos SGDBs, normalmente, desenvolvem e distribuem **drivers de conexão**. Um driver de conexão atua como um intermediário entre as aplicações e os SGDBs.

Os drivers de conexão são “tradutores” de comandos escritos em uma determinada linguagem de programação para comandos definidos de acordo com o protocolo de um SGDB. Utilizando um driver de conexão, os desenvolvedores das aplicações não manipulam diretamente as mensagens binárias trocadas entre as aplicações e os SGDBs.



Mais Sobre

Em alguns casos, o protocolo binário de um determinado SGDB é fechado. Consequentemente, a única maneira de se comunicar com ele é através de um driver de conexão oferecido pelo fabricante desse SGDB.

JDBC

Suponha que os drivers de conexão fossem desenvolvidos sem nenhum padrão. Cada driver teria sua própria interface, ou seja, seu próprio conjunto de instruções. Consequentemente, os desenvolvedores teriam de conhecer a interface de cada um dos drivers dos respectivos SGDBs que fossem utilizar.

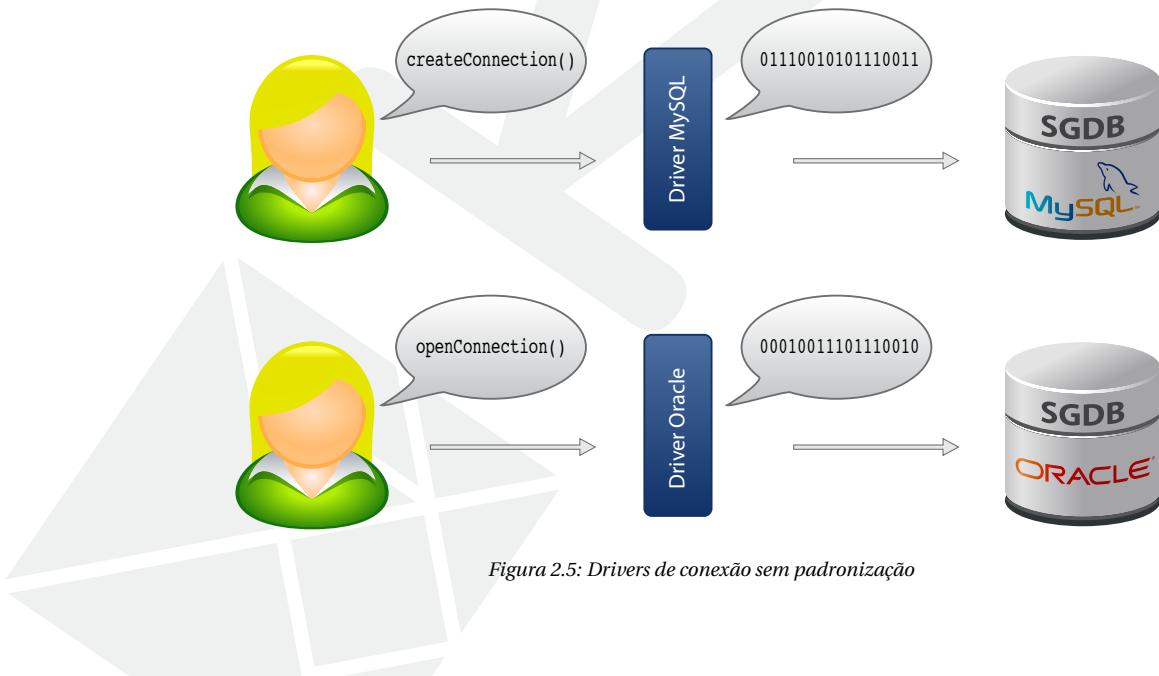


Figura 2.5: Drivers de conexão sem padronização

Para facilitar o trabalho do desenvolvedor da aplicação, a plataforma Java possui uma especificação que padroniza os drivers de conexão. A sigla dessa especificação é **JDBC** (*Java Database Connectivity*). Em geral, as empresas proprietárias dos SGBDs desenvolvem e distribuem drivers de conexão que seguem a especificação JDBC.

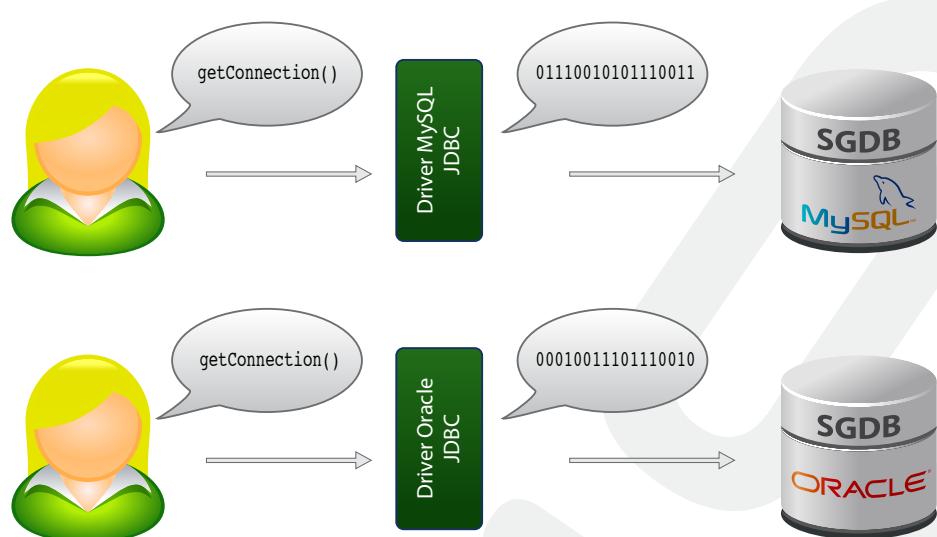


Figura 2.6: Drivers de conexão padronizados pela especificação JDBC

Instalando o Driver JDBC do MySQL Server

Podemos obter um driver de conexão JDBC para o MySQL Server na seguinte url:

<http://www.mysql.com/downloads/connector/j/>.

A instalação desse driver consiste em descompactar o arquivo obtido no site acima e depois incluir o arquivo **.jar** com o driver no **class path** da aplicação.

Criando uma conexão

Com o driver de conexão JDBC adicionado à aplicação, já é possível criar uma conexão. Abaixo, estão as informações necessárias para a criação de uma conexão JDBC.

- Nome do driver JDBC.
- Endereço (IP e porta) do SGDB.
- Nome da base de dados.
- Um usuário do SGBD.
- Senha do usuário.

O nome do driver JDBC, o endereço do SGDB e nome da base de dados são definidos na **string de conexão** ou **url de conexão**. Veja o exemplo abaixo:

```
1 String stringDeConexao = "jdbc:mysql://localhost/livraria";
```

Código Java 2.1: String de conexão

A classe responsável pela criação de uma conexão JDBC é a `DriverManager` do pacote `java.sql`. A string de conexão, o usuário e a senha devem ser passados ao método estático `getConnection()` da classe `DriverManager` para que ela possa criar uma conexão JDBC.

```

1 String urlDeConexao = "jdbc:mysql://localhost/livraria";
2 String usuario = "root";
3 String senha = "";
4 try {
5     Connection conexao = DriverManager.getConnection(urlDeConexao, usuario, senha);
6 } catch (SQLException e) {
7     e.printStackTrace();
8 }
```

Código Java 2.2: Criando uma conexão JDBC

Inserindo registros

Após estabelecer uma conexão JDBC, podemos executar operações. A primeira operação que realizaremos é a inserção de registros em uma tabela. O primeiro passo para executar essa operação é definir o código SQL correspondente.

```
1 String sql = "INSERT INTO Editora (nome, email) VALUES ('K19',' contato@k19.com.br');";
```

Código Java 2.3: Código SQL correspondente à operação de inserção

O código SQL correspondente à operação que desejamos executar deve ser passado como parâmetro para o método `prepareStatement()` de uma conexão JDBC. Esse método criará um objeto que representa a operação que será executada. A operação poderá ser executada posteriormente através do método `execute()`.

```

1 // criando um prepared statement
2 PreparedStatement comando = conexao.prepareStatement(sql);
3
4 // executando o prepared statement
5 comando.execute();
6 comando.close();
```

Código Java 2.4: Criando um prepared statement



Importante

A mesma conexão pode ser reaproveitada para executar várias operações. Quando não houver mais operações a serem executadas, devemos finalizar a conexão JDBC através do método `close()`. Finalizar as conexões JDBC que não são mais necessárias é importante pois libera recursos no SGBD.

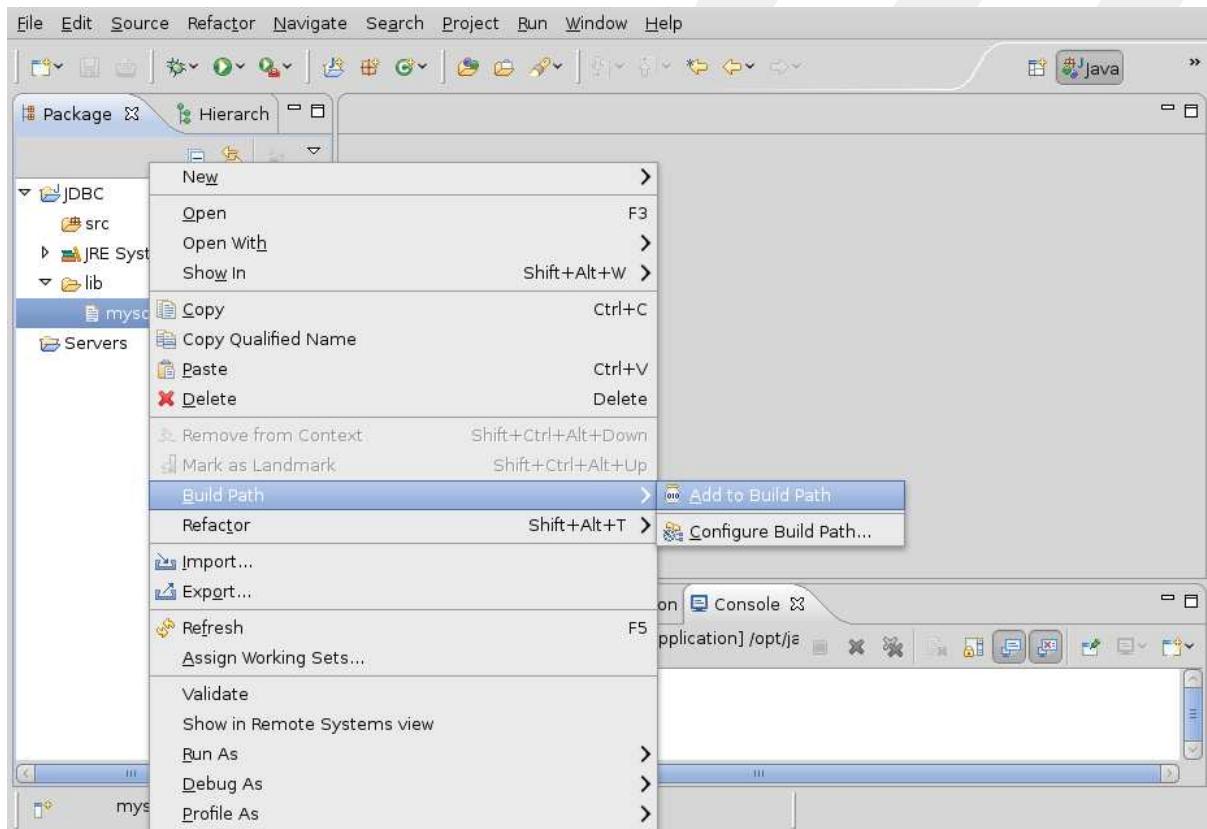
```
1 conexao.close();
```

Código Java 2.5: Finalizando uma conexão JDBC



Exercícios de Fixação

- 1 No Eclipse, crie um projeto Java chamado **JDBC**.
- 2 Crie uma pasta chamada **lib** no projeto **JDBC**.
- 3 Entre na pasta **K19-Arquivos/MySQL-Connector-JDBC** da Área de Trabalho e copie o arquivo **mysql-connector-java-5.1.13-bin.jar** para a pasta **lib** do projeto **JDBC**.
- 4 Adicione o arquivo **mysql-connector-java-5.1.13-bin.jar** ao **build path**. Veja a imagem abaixo.



- 5 Crie uma classe chamada **InsereEditora**, e adicione o seguinte conteúdo ao arquivo:

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.PreparedStatement;
4 import java.util.Scanner;
5
6 public class InsereEditora {
7     public static void main(String[] args) {
8         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
9         String usuario = "root";
10        String senha = "root";
11
12        Scanner entrada = new Scanner(System.in);
13
14        try {
15            System.out.println("Abrindo conexão...");
16            Connection conexao =
17                DriverManager.getConnection(stringDeConexao, usuario, senha);
18
19            System.out.println("Digite o nome da editora: ");
```

```

20     String nome = entrada.nextLine();
21
22     System.out.println("Digite o email da editora: ");
23     String email = entrada.nextLine();
24
25     String sql = "INSERT INTO Editora (nome, email) " +
26         "VALUES ('" + nome + "', '" + email + "')";
27
28     PreparedStatement comando = conexao.prepareStatement(sql);
29
30     System.out.println("Executando comando...");
31     comando.execute();
32
33     System.out.println("Fechando conexão...");
34     conexao.close();
35 } catch (Exception e) {
36     e.printStackTrace();
37 }
38 }
39 }
```

Código Java 2.6: InsereEditora.java

Execute e verifique se o registro foi inserido com sucesso na base de dados.



Exercícios Complementares

- 1 Crie uma classe chamada **InsereLivros** para cadastrar livros na base de dados.

SQL Injection

A implementação da inserção de registros feita anteriormente possui uma falha grave. Os dados obtidos do usuário através do teclado não são tratados antes de serem enviados para o SGDB.

Esses dados podem conter caracteres especiais. Se esses caracteres não são tratados, o comportamento esperado da operação é afetado. Eventualmente, registros não são inseridos como deveriam ou brechas de segurança podem se abrir.

Por exemplo, considere a classe **InsereEditora** do exercício de fixação. Se o usuário digitar “O'Reilly” e “oreilly@email.com”, o código SQL gerado pela aplicação seria:

```
1 INSERT INTO Editora (nome, email) VALUES ('O'Reilly', 'oreilly@email.com')
```

Observe que o caractere aspas simples aparece cinco vezes no código SQL acima. O SGDB não saberia dizer onde de fato termina o nome da editora. Ao tentar executar esse código, um erro de sintaxe é lançado pelo MySQL Server. Para resolver esse problema manualmente, devemos adicionar o caractere “\” antes do caractere aspas simples que faz parte do nome da editora. Na sintaxe do MySQL Server, o caractere “\” deve ser acrescentado imediatamente antes de todo caractere especial que deve ser tratado como um caractere comum.

```
1 INSERT INTO Editora (nome, email) VALUES ('O\'Reilly', 'oreilly@email.com')
```

Os valores recebidos dos usuários devem ser analisados e os caracteres especiais contidos nesses

valores devem ser tratados. Esse processo é extremamente trabalhoso, pois o conjunto de caracteres especiais e a forma de tratá-los é diferente em cada SGDB.

A responsabilidade do tratamento dos caracteres especiais contidos nos valores de entrada dos usuários pode ser repassada para os drivers JDBC. Dessa forma, o código das aplicações se torna independente das particularidades desse processo para cada SGDB.



Mais Sobre

O processo de tratamento dos caracteres especiais das entradas dos usuários é denominado **sanitize**.

```
1 // lendo as entradas do usuário
2 System.out.println("Digite o nome da editora: ");
3 String nome = entrada.nextLine();
4
5 System.out.println("Digite o email da editora: ");
6 String email = entrada.nextLine();
7
8 // código sql com marcadores para as entradas do usuário
9 String sql = "INSERT INTO Editora (nome, email) VALUES (?, ?)";
10
11 // criando um comando a partir do código SQL
12 PreparedStatement comando = conexao.prepareStatement(sql);
13
14 // adicionando as entradas do usuários no comando
15 // o processo de sanitização ocorre aqui
16 comando.setString(1, nome);
17 comando.setString(2, email);
```

Código Java 2.10: “Sanitizando” as entradas dos usuários

Observe que o código SQL foi definido com parâmetros através do caractere “?”. Antes de executar o comando, é necessário determinar os valores dos parâmetros. Essa tarefa pode ser realizada através do método `setString()`, que recebe o índice (posição) do parâmetro no código SQL e o valor correspondente. Esse método faz o tratamento dos caracteres especiais contidos nos valores de entrada do usuário de acordo com as regras do SGDB utilizado.



Exercícios de Fixação

- 6 Provoque um erro de SQL Injection na classe `InsereEditoras`. (Dica: tente entradas com aspas simples.)
- 7 Altere o código para eliminar o problema do SQL Injection.

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.PreparedStatement;
4 import java.util.Scanner;
5
6 public class InsereEditora {
7     public static void main(String[] args) {
8
9         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
10        String usuario = "root";
11        String senha = "root";
```

```

12 Scanner entrada = new Scanner(System.in);
13
14 try {
15     System.out.println("Abrindo conexão...");
16     Connection conexao =
17         DriverManager.getConnection(stringDeConexao, usuario, senha);
18
19     System.out.println("Digite o nome da editora: ");
20     String nome = entrada.nextLine();
21
22     System.out.println("Digite o email da editora: ");
23     String email = entrada.nextLine();
24
25     String sql = "INSERT INTO Editora (nome, email) VALUES (?, ?)";
26
27     PreparedStatement comando = conexao.prepareStatement(sql);
28     comando.setString(1, nome);
29     comando.setString(2, email);
30
31     System.out.println("Executando comando...");
32     comando.execute();
33
34     System.out.println("Fechando conexão...");
35     conexao.close();
36 } catch (Exception e) {
37     e.printStackTrace();
38 }
39
40 }
41

```

Código Java 2.11: InsereEditora.java

Exercícios Complementares

- 2** Provoque um erro de SQL Injection na classe `InsereLivros`. (Dica: tente entradas com aspas simples.)
- 3** Altere o código para eliminar o problema do SQL Injection.
- 4** Agora tente causar novamente o problema de SQL Injection ao inserir novos livros.

Listando registros

O processo para executar um comando de consulta é bem parecido com o processo de inserir registros. O primeiro passo é definir a consulta em SQL.

```

1 String sql = "SELECT * FROM Editora;";
2
3 PreparedStatement comando = conexao.prepareStatement(sql);
4
5 System.out.println("Executando comando...");
6 ResultSet resultado = comando.executeQuery();

```

Código Java 2.13: Realizando uma consulta.

A diferença é que para executar um comando de consulta é necessário utilizar o método `executeQuery()` ao invés do `execute()`. Esse método devolve um objeto da interface `java.sql.ResultSet`.

Set, que é responsável por armazenar os resultados da consulta.

Os dados contidos no ResultSet podem ser acessados através de métodos, como o getString, getInt, getDouble, etc, de acordo com o tipo do campo. Esses métodos recebem como parâmetro uma string referente ao nome da coluna correspondente.

```
1 int id = resultado.getInt("id"),
2 String nome = resultado.getString("nome"),
3 String email = resultado.getString("email");
```

Código Java 2.14: Recuperando os dados de um ResultSet

O código acima mostra como os campos do primeiro registro da consulta são recuperados. Para recuperar os dados dos outros registros é necessário avançar o ResultSet através do método next().

```
1 int id1 = resultado.getInt("id"),
2 String nome1 = resultado.getString("nome"),
3 String email1 = resultado.getString("email");
4
5 resultado.next();
6
7 int id2 = resultado.getInt("id"),
8 String nome2 = resultado.getString("nome"),
9 String email2 = resultado.getString("email");
```

Código Java 2.15: Avançando o ResultSet

O próprio método next() devolve um valor booleano para indicar se o ResultSet conseguiu avançar para o próximo registro. Quando esse método devolver false significa que não há mais registros para serem consultados.

```
1 while(resultado.next()) {
2     int id = resultado.getInt("id"),
3     String nome = resultado.getString("nome"),
4     String email = resultado.getString("email");
5 }
```

Código Java 2.16: Iterando os registros do ResultSet



Exercícios de Fixação

- 8 Insira algumas editoras utilizando a classe InsereEditora que você criou anteriormente.
- 9 Adicione uma nova classe ao projeto chamada ListaEditoras. O objetivo é listar as editoras que foram salvos no banco.

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.PreparedStatement;
4 import java.sql.ResultSet;
5
6 public class ListaEditoras {
7     public static void main(String[] args) {
8         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
9         String usuario = "root";
10        String senha = "root";
11
12        try {
```

```

13     System.out.println("Abrindo conexão...");
14     Connection conexao =
15         DriverManager.getConnection(stringDeConexao, usuario, senha);
16
17     String sql = "SELECT * FROM Editora;";
18
19     PreparedStatement comando = conexao.prepareStatement(sql);
20
21     System.out.println("Executando comando...");
22     ResultSet resultado = comando.executeQuery();
23
24     System.out.println("Resultados encontrados: \n");
25     while (resultado.next()) {
26         System.out.printf("%d : %s - %s\n",
27             resultado.getInt("id"),
28             resultado.getString("nome"),
29             resultado.getString("email"));
30     }
31
32     System.out.println("\nFechando conexão...");
33     conexao.close();
34 } catch (Exception e) {
35     e.printStackTrace();
36 }
37
38 }
```

Código Java 2.17: ListaEditoras.java



Exercícios Complementares

- 5 Crie uma classe para listar os livros cadastrados na base de dados.

Connection Factory

Você deve ter percebido que em diversos pontos diferentes da nossa aplicação precisamos de uma conexão JDBC. Se a url de conexão for definida em cada um desses pontos, teremos um problema de manutenção. Imagine que o driver do banco seja atualizado ou que o IP do SGBD seja alterado. Teríamos que alterar o código da nossa aplicação em muitos lugares. Mais precisamente, cada ocorrência da url de conexão precisaria ser modificada. A probabilidade de algum ponto não ser corrigido é grande.

Para diminuir o trabalho de manutenção, podemos implementar uma classe responsável pela criação e distribuição de conexões. A url de conexão deve ser definida nessa classe e somente nela. Consequentemente, alterações nas informações contidas na url de conexão afetariam apenas uma classe da aplicação.

```

1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.SQLException;
4
5 public class ConnectionFactory {
6     public static Connection createConnection() {
7         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
8         String usuario = "root";
9         String senha = "root";
10    }
```

```

11     Connection conexao = null;
12
13     try {
14         conexao = DriverManager.getConnection(stringDeConexao, usuario, senha);
15     } catch (SQLException e) {
16         e.printStackTrace();
17     }
18     return conexao;
19 }
20 }
```

Código Java 2.19: ConnectionFactory.java

Agora, podemos obter uma nova conexão apenas chamando o método `createConnection()`. O resto do sistema não precisa mais conhecer os detalhes sobre a criação das conexões com o banco de dados, diminuindo o acoplamento da aplicação.



Exercícios de Fixação

- 10** Adicione uma nova classe chamada `ConnectionFactory` com o código abaixo.

```

1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.SQLException;
4
5 public class ConnectionFactory {
6     public static Connection createConnection() {
7         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
8         String usuario = "root";
9         String senha = "root";
10
11     Connection conexao = null;
12
13     try {
14         conexao = DriverManager.getConnection(stringDeConexao, usuario, senha);
15     } catch (SQLException e) {
16         e.printStackTrace();
17     }
18     return conexao;
19 }
20 }
```

Código Java 2.20: ConnectionFactory.java

- 11** Altere as classes `InsereEditora` e `ListaEditoras` para que elas utilizem a fábrica de conexão. Depois, execute-as novamente.

```

1 import java.sql.Connection;
2 import java.sql.PreparedStatement;
3 import java.util.Scanner;
4
5 public class InsereEditora {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8
9         try {
10             System.out.println("Abrindo conexão...");
11             Connection conexao = ConnectionFactory.createConnection();
12
13             System.out.println("Digite o nome da editora: ");
14             String nome = entrada.nextLine();
15 }
```

```

16     System.out.println("Digite o email da editora: ");
17     String email = entrada.nextLine();
18
19     String sql = "INSERT INTO Editora (nome, email) " +
20         "VALUES (?, ?)";
21
22     PreparedStatement comando =conexao.prepareStatement(sql);
23     comando.setString(1, nome);
24     comando.setString(2, email);
25
26     System.out.println("Executando comando...");
27     comando.execute();
28
29     System.out.println("Fechando conexão...");
30     conexao.close();
31 } catch (Exception e) {
32     e.printStackTrace();
33 }
34 }
35 }
```

Código Java 2.21: InsereEditora.java

```

1 import java.sql.Connection;
2 import java.sql.PreparedStatement;
3 import java.sql.ResultSet;
4
5 public class ListaEditoras {
6     public static void main(String[] args) {
7         try {
8             System.out.println("Abrindo conexão...");
9             Connection conexao = ConnectionFactory.createConnection();
10
11             String sql = "SELECT * FROM Editora;";
12
13             PreparedStatement comando = conexao.prepareStatement(sql);
14
15             System.out.println("Executando comando...");
16             ResultSet resultado = comando.executeQuery();
17
18             System.out.println("Resultados encontrados: \n");
19             while (resultado.next()) {
20                 System.out.printf("%d : %s - %s\n",
21                     resultado.getInt("id"),
22                     resultado.getString("nome"),
23                     resultado.getString("email"));
24             }
25
26             System.out.println("\nFechando conexão...");
27             conexao.close();
28         } catch (Exception e) {
29             e.printStackTrace();
30         }
31     }
32 }
```

Código Java 2.22: ListaEditoras.java

Exercícios Complementares

- 6** Altere as classes `InsereLivro` e `ListaLivros` para que elas utilizem a fábrica de conexão. Depois, execute-as novamente.



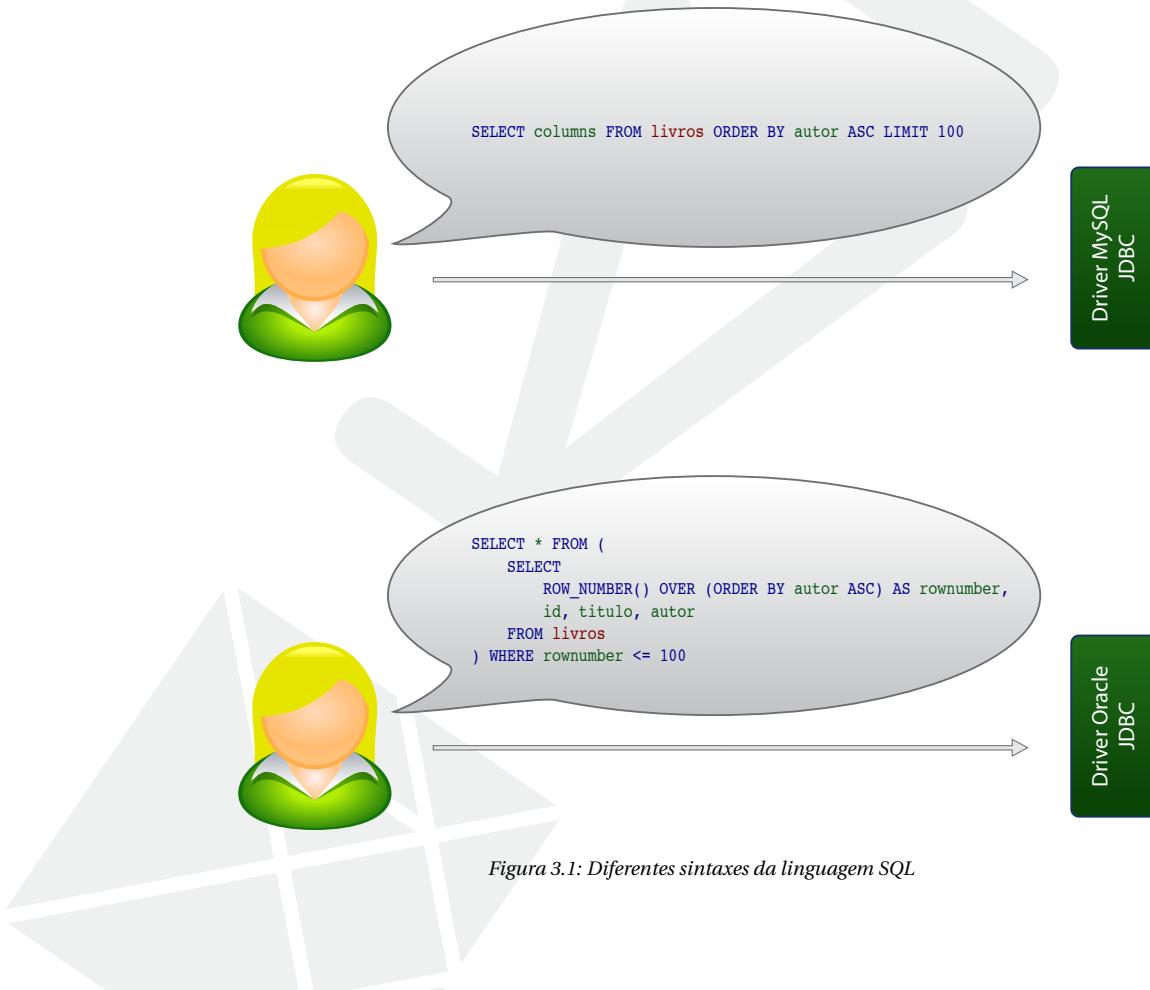
Desafios

- 1 Implemente a remoção de editoras pelo id.
- 2 Implemente a alteração dos dados das editoras pelo id.

JPA 2 E HIBERNATE

Múltiplas sintaxes da linguagem SQL

No capítulo anterior, utilizamos a especificação JDBC para fazer uma aplicação Java interagir com os SGDBs. Nessa interação, as consultas são definidas com a linguagem SQL. A sintaxe dessa linguagem é diferente em cada SGDB. Dessa forma, a complexidade do trabalho dos desenvolvedores aumenta. Para resolver esse problema, as consultas deveriam ser definidas através de um mecanismo independente da linguagem SQL.



Orientação a Objetos VS Modelo Relacional

Outro problema na comunicação entre uma aplicação Java e um SGDB é o conflito de paradigmas. Os SGDBs são organizados seguindo o modelo relacional. Por outro lado, as aplicações Java

utilizam o modelo orientado a objetos.

A transição de dados entre o modelo relacional e o modelo orientado a objetos não é simples. Para realizar essa transição, é necessário definir um mapeamento entre os conceitos desses dois paradigmas. Por exemplo, classes podem ser mapeadas para tabelas, objetos para registros, atributos para campos e referência entre objetos para chaves estrangeiras.

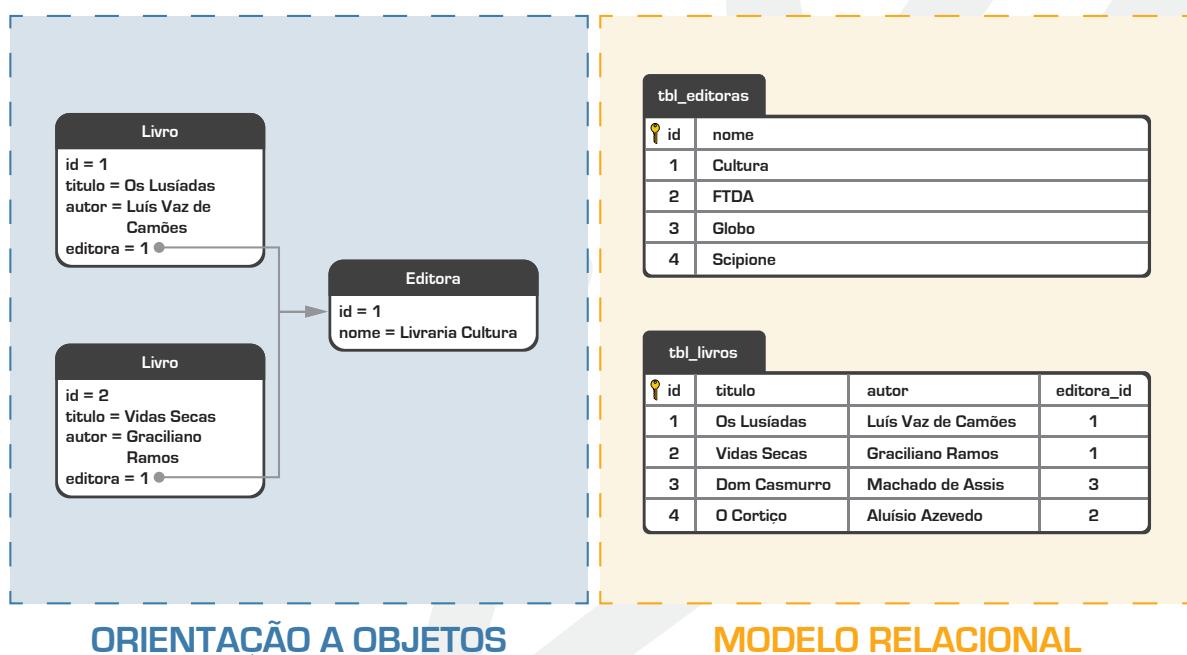


Figura 3.2: Modelo Orientado a Objetos vs Modelo Relacional

Ferramentas ORM

Para facilitar a comunicação entre aplicações Java que seguem o modelo orientado a objetos e os SGDBs que seguem o modelo relacional, podemos utilizar ferramentas que automatizam a transição de dados entre as aplicações e os SGDBs. Essas ferramentas são conhecidas como ferramentas **ORM** (*Object Relational Mapper*).

As ferramentas ORM oferecem mecanismos de consultas independentes da linguagem SQL. Dessa forma, o acoplamento entre as aplicações e os SGDBs diminui drasticamente. A principal ferramenta ORM para Java utilizada no mercado é o Hibernate. Mas, existem outras que possuem o mesmo objetivo.



Figura 3.3: Transformação dos dados do Modelo Relacional para o Modelo Orientado a Objetos



Figura 3.4: Transformação dos dados do Modelo Orientado a Objetos para o Modelo Relacional

O que é JPA e Hibernate?

Após o sucesso do Hibernate, a especificação **JPA** (*Java Persistence API*) foi criada com o objetivo de padronizar as ferramentas ORM para aplicações Java e consequentemente diminuir a complexidade do desenvolvimento. Atualmente, essa especificação está na sua segunda versão.

Ela especifica um conjunto de classes e métodos que as ferramentas ORM devem implementar. Veja que a JPA é apenas uma especificação. Ela não implementa nenhum código. Para isso, utilizamos alguma das diversas implementações da JPA. Neste curso, utilizaremos o Hibernate como implementação de JPA. As outras implementações de JPA mais conhecidas são **EclipseLink** e **OpenJPA**. Optamos por utilizar o Hibernate por ele ser o mais antigo e mais utilizado atualmente.

Caso você queira utilizar outra ferramenta ORM, poderá aplicar os conceitos aqui aprendidos justamente porque eles seguem a mesma especificação. Assim, podemos programar voltado à especificação e substituir uma implementação pela outra, sem precisar reescrever o código da nossa aplicação. Claro que teríamos que alterar alguns arquivos de configuração, mas o código da aplicação permaneceria o mesmo.

Bibliotecas

Antes de começar a utilizar o Hibernate, é necessário baixar do site oficial o **bundle** que inclui os

jar's do hibernate e todas as suas dependências. Neste curso, utilizaremos a versão 3.5.1. A url do site oficial do Hibernate é <http://www.hibernate.org/>.

Configuração

Para configurar o Hibernate em uma aplicação, devemos criar um arquivo chamado **persistence.xml**. O conteúdo desse arquivo possuirá informações sobre o banco de dados, como a url de conexão, usuário e senha, além de dados sobre a implementação de JPA que será utilizada.

O arquivo **persistence.xml** deve estar em uma pasta chamada **META-INF**, que deve estar no classpath da aplicação. Veja abaixo um exemplo de configuração para o **persistence.xml**.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.0"
3   xmlns="http://java.sun.com/xml/ns/persistence"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
6     ns/persistence/persistence_2_0.xsd">
7 
8   <persistence-unit name="K19-PU" transaction-type="RESOURCE_LOCAL">
9     <provider>org.hibernate.ejb.HibernatePersistence</provider>
10    <properties>
11      <property name="hibernate.dialect" value="org.hibernate.dialect.←
12        MySQL5InnoDBialect"/>
13 
14      <property name="hibernate.hbm2ddl.auto" value="create"/>
15 
16      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
17 
18      <property name="javax.persistence.jdbc.user" value="usuario"/>
19 
20      <property name="javax.persistence.jdbc.password" value="senha"/>
21 
22      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/←
23        K19-DB"/>
24    </properties>
25  </persistence-unit>
26 </persistence>
```

Código XML 3.1: *persistence.xml*

A propriedade **hibernate.dialect** permite que a aplicação escolha qual sintaxe de SQL deve ser utilizada pelo Hibernate.



Mais Sobre

Consulte o artigo da K19 sobre configuração do Hibernate e MySQL na seguinte url:
<http://www.k19.com.br/artigos/configurando-hibernate-com-mysql/>.

Mapeamento

Um dos principais objetivos das ferramentas ORM é estabelecer o mapeamento entre os conceitos do modelo orientado a objetos e os conceitos do modelo relacional. Esse mapeamento pode ser definido através de XML ou de maneira mais prática com anotações Java.

A seguir, veremos as principais anotações Java de mapeamento do JPA. Essas anotações estão no pacote **javax.persistence**.

@Entity É a principal anotação do JPA. Ela deve aparecer antes do nome de uma classe e deve ser definida em todas as classes que terão objetos persistidos no banco de dados.

As classes anotadas com **@Entity** são mapeadas para tabelas. Por convenção, as tabelas possuem os mesmos nomes das classes. Mas, podemos alterar esse comportamento utilizando a anotação **@Table**.

Os atributos declarados em uma classe anotada com **@Entity** são mapeados para colunas na tabela correspondente à classe. Outra vez, por convenção, as colunas possuem os mesmos nomes dos atributos. Novamente, podemos alterar esse padrão utilizando a anotação **@Column**.

@Id Utilizada para indicar qual atributo de uma classe anotada com **@Entity** será mapeado para a chave primária da tabela correspondente à classe. Geralmente o atributo anotado com **@Id** é do tipo **Long**.

@GeneratedValue Geralmente vem acompanhado da anotação **@Id**. Serve para indicar que o atributo é gerado pelo banco, no momento em que um novo registro é inserido.

@Table Utilizada para alterar o nome padrão da tabela. Ela recebe o parâmetro **name** para indicar qual nome deve ser utilizado na tabela. Veja o exemplo:

```

1 @Table(name="Publisher")
2 @Entity
3 public class Editora {
4     // ...
5 }
```

Código Java 3.1: *Editora.java*

@Column Utilizado para alterar o nome da coluna que será usado na tabela. Caso você esteja utilizando um banco de dados legado, no qual os nomes das colunas já foram definidos, você pode mudar através dessa anotação. Além disso, podemos estabelecer certas restrições, como determinar se o campo pode ou não ser nulo.

```

1 @Entity
2 public class Editora {
3     @Column(name="publisher_name", nullable=false)
4     private String nome;
5 }
```

Código Java 3.2: *Editora.java*

@Transient Serve para indicar que um atributo não deve ser persistido, ou seja, os atributos anotados com **@Transient** não são mapeados para colunas.

@Lob Utilizado para atributos que armazenam textos muito grandes, ou arquivos binários contendo imagens ou sons que serão persistidos no banco de dados. O tipo do atributo deve ser **String**, **Byte[]**, **byte[]** ou **java.sql.Blob**.

@Temporal Utilizado para atributos do tipo **Calendar** ou **Date**. Por padrão, tanto data quanto hora são armazenados no banco de dados. Mas, com a anotação **@Temporal**, podemos mandar persistir somente a data ou somente a hora.

```
1 @Entity  
2 public class Livro {  
3     @Temporal(TemporalType.DATE)  
4     private Calendar publicacao;  
5     // ...  
6 }  
7 }
```

Código Java 3.3: Livro.java

Gerando Tabelas

Uma das vantagens de se utilizar o Hibernate é que ele é capaz de gerar as tabelas do banco para a nossa aplicação. Ele faz isso de acordo com as anotações colocadas nas classes e as informações presentes no `persistence.xml`.

As tabelas são geradas através de um método da classe `Persistence`, o `createEntityManagerFactory(String entityUnit)`. O parâmetro `entityUnit` permite escolher, pelo nome, uma unidade de persistência definida no `persistence.xml`.

A política de criação das tabelas pode ser alterada modificando o valor a propriedade `hibernate.hbm2ddl.auto` no arquivo `persistence.xml`. Podemos, por exemplo, fazer com que o Hibernate sempre sobrescreva as tabelas existentes, bastando definir a propriedade `hibernate.hbm2ddl.auto` com o valor `create-drop`.

```
1 <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
```

Uma outra opção é configurar o Hibernate para simplesmente atualizar as tabelas de acordo com as mudanças nas anotações sem removê-las. Nesse caso, o valor da propriedade `hibernate.hbm2ddl.auto` deve ser `update`.

```
1 <property name="hibernate.hbm2ddl.auto" value="update"/>
```



Exercícios de Fixação

- 1 Crie um projeto no Eclipse chamado JPA2-Hibernate e feche o projeto JDBC para não gerar confusão na hora de manipular os arquivos.
- 2 Crie uma pasta chamada `lib` dentro do projeto JPA2-Hibernate.
- 3 Entre na pasta K19-Arquivos/Hibernate da Área de Trabalho e copie os jar's do Hibernate para a pasta `lib` do projeto JPA2-Hibernate.
- 4 Entre na pasta K19-Arquivos/MySQL-Connector-JDBC da Área de Trabalho e copie o arquivo `mysql-connector-java-5.1.13-bin.jar` para pasta `lib` do projeto JPA2-Hibernate.
- 5 Entre na pasta K19-Arquivos/SLF4J da Área de Trabalho e copie os jar's para pasta `lib` do

projeto JPA2-Hibernate.

- 6 Entre na pasta K19-Arquivos/Log4J da Área de Trabalho e copie o arquivo log4j-1.2.16.jar para pasta lib do projeto JPA2-Hibernate.
- 7 Adicione os jar's da pasta lib ao build path do projeto JPA2-Hibernate. Você deve selecionar os arquivos e adicioná-los no build path.
- 8 Crie uma pasta chamada META-INF na pasta src no projeto JPA2-Hibernate.
- 9 Crie o arquivo de configurações persistence.xml na pasta META-INF.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.0"
3   xmlns="http://java.sun.com/xml/ns/persistence"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
6     ns/persistence/persistence_2_0.xsd">
7
8   <persistence-unit name="livraria-pu" transaction-type="RESOURCE_LOCAL">
9     <provider>org.hibernate.ejb.HibernatePersistence</provider>
10    <properties>
11      <property name="hibernate.dialect" value="org.hibernate.dialect.←
12        MySQL5InnoDBialect"/>
13
14      <property name="hibernate.hbm2ddl.auto" value="create"/>
15
16      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
17
18      <property name="javax.persistence.jdbc.user" value="root"/>
19
20      <property name="javax.persistence.jdbc.password" value="root"/>
21
22      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/←
23        livraria"/>
</properties>
</persistence-unit>
</persistence>
```

Código XML 3.4: persistence.xml

- 10 Crie uma classe para modelar as editoras da nossa livraria e acrescente as anotações necessárias para fazer o mapeamento. Obs: As anotações devem ser importadas do pacote javax.persistence.

```

1 @Entity
2 public class Editora {
3   @Id @GeneratedValue
4   private Long id;
5
6   private String nome;
7
8   private String email;
9
10  // GETTERS AND SETTERS
11 }
```

Código Java 3.4: Editora.java

- 11 Apague a tabela Livro e depois a Editora.
- 12 Configure o Log4J criando um arquivo chamado log4j.properties na pasta src do projeto JPA2-Hibernate.

```

1 log4j.rootCategory=INFO, CONSOLE
2 log4j.appenders.CONSOLE=org.apache.log4j.ConsoleAppender
3 log4j.appenders.CONSOLE.layout=org.apache.log4j.PatternLayout
4 log4j.appenders.CONSOLE.layout.ConversionPattern=%r [%t] %-5p %c - %m%n

```

Arquivo de Propriedades 3.1: log4j.properties

- 13 Gere as tabelas através da classe Persistence. Para isso, crie uma classe com método main. Obs: As classes devem ser importadas do pacote javax.persistence.

```

1 public class GeraTabelas {
2     public static void main(String[] args) {
3         EntityManagerFactory factory =
4             Persistence.createEntityManagerFactory("livraria-pu");
5
6         factory.close();
7     }
8 }

```

Código Java 3.5: GeraTabelas.java

Através do MySQL Query Browser, verifique se a tabela Editora foi criada corretamente.

Manipulando entidades

Para manipular as entidades da nossa aplicação, devemos utilizar um EntityManager que é obtido através de uma EntityManagerFactory.

```

1 EntityManagerFactory factory =
2     Persistence.createEntityManagerFactory("K19");
3
4 EntityManager manager = factory.createEntityManager();

```

Código Java 3.6: Obtendo um EntityManager

Persistindo

Para armazenar as informações de um objeto no banco de dados, o primeiro passo é utilizar o método persist() do EntityManager.

```

1 Editora novaEditora = new Editora();
2 novaEditora.setNome("K19 - Livros")
3 novaEditora.setEmail("contato@k19.com.br");
4
5 manager.persist(novaEditora);

```

Código Java 3.7: Marcando um objeto para ser persistido

É importante destacar que o método persist() apenas marca os objetos que devem ser armazenados no banco de dados. Os objetos serão armazenados após a chamada do método commit(), como veremos adiante.

Buscando

Para obter um objeto que contenha informações do banco de dados, podemos utilizar o método `find()` ou o método `getReference()` do `EntityManager`.

```
1 Editora editorial1 = manager.find(Editora.class, 1L);
2 Editora editorial2 = manager.getReference(Editora.class, 2L);
```

Código Java 3.8: Obtendo objetos com informações do banco de dados

Há uma diferença entre os dois métodos básicos de busca `find()` e `getReference()`. O método `find()` recupera os dados desejados imediatamente. Já o método `getReference()` posterga essa tarefa até a primeira chamada de um método `get` do objeto.

Removendo

Para remover o registro correspondente a um objeto, devemos utilizar o método `remove()` do `EntityManager`.

```
1 Editora editorial1 = manager.find(Editora.class, 1L);
2 manager.remove(editorial1);
```

Código Java 3.9: Marcando um objeto para ser removido

Atualizando

Para alterar os dados do registro correspondente a um objeto, basta utilizar os próprios métodos setters desse objeto.

```
1 Editora editora = manager.find(Editora.class, 1L);
2 editora.setNome("K19 - Livros e Publicações");
```

Código Java 3.10: Alterando os dados de um registro

Listando

Para obter uma listagem com todos os objetos referentes aos registros de uma tabela, podemos utilizar a linguagem de consulta do JPA, a **JPQL**, que é muito parecida com a linguagem SQL. A principal vantagem do JPQL em relação ao SQL é que a sintaxe do JPQL não depende do SGDB utilizado.

```
1 Query query = manager.createQuery("SELECT e FROM Editora e");
2 List<Editora> editoras = query.getResultList();
```

Código Java 3.11: Obtendo uma lista de objetos com informações do banco de dados

Transações

As modificações realizadas nos objetos administrados por um EntityManager são mantidas em memória. Em certos momentos, é necessário sincronizar os dados da memória com os dados do banco de dados. Essa sincronização deve ser realizada através de uma transação JPA criada pelo EntityManager que administra os objetos que desejamos sincronizar.

Para abrir uma transação, utilizamos o método `begin()`.

```
1 manager.getTransaction().begin();
```

Código Java 3.12: Abrindo uma transação

Com uma transação aberta, podemos sincronizar os dados da memória com os dados do banco através do método `commit()`.

```
1 Editora editora = manager.find(Editora.class, 1L);
2 editora.setNome("K19 - Livros e Publicações");
3
4 manager.getTransaction().begin();
5 manager.getTransaction().commit();
```

Código Java 3.13: Sincronizando com o método commit()



Exercícios de Fixação

- 14 No arquivo de configurações `persistence.xml`, altere o valor da propriedade `hibernate.hbm2ddl.auto` para `update`. Assim, as tabelas não serão recriadas a cada execução e sim apenas atualizadas.
- 15 Crie um teste para inserir editoras no banco de dados.

```
1 public class InsereEditoraComJPA {
2
3     public static void main(String[] args) {
4         EntityManagerFactory factory =
5             Persistence.createEntityManagerFactory("livraria-pu");
6
7         EntityManager manager = factory.createEntityManager();
8
9         Editora novaEditora = new Editora();
10
11        Scanner entrada = new Scanner(System.in);
12
13        System.out.println("Digite o nome da editora: ");
14        novaEditora.setNome(entrada.nextLine());
15
16        System.out.println("Digite o email da editora: ");
17        novaEditora.setEmail(entrada.nextLine());
18
19        manager.persist(novaEditora);
20
21        manager.getTransaction().begin();
22        manager.getTransaction().commit();
23
24        manager.close();
25        factory.close();
26    }
27}
```

Código Java 3.14: InsereEditoraComJPA.java

- 16 Crie um teste para listar as editoras inseridas no banco de dados.

```

1 public class ListaEditorasComJPA {
2
3     public static void main(String[] args) {
4         EntityManagerFactory factory =
5             Persistence.createEntityManagerFactory("livraria-pu");
6
7         EntityManager manager = factory.createEntityManager();
8
9         Query query = manager.createQuery("SELECT e FROM Editora e");
10        List<Editora> editoras = query.getResultList();
11
12        for(Editora e : editoras) {
13            System.out.println("EDITORA: " + e.getNome() + " - " + e.getEmail());
14        }
15
16        manager.close();
17        factory.close();
18    }
19 }
```

Código Java 3.15: ListaEditorasComJPA.java

Repository

A interface EntityManager do JPA oferece recursos suficientes para que os objetos do domínio sejam recuperados ou persistidos no banco de dados. Porém, em aplicações com alta complexidade e grande quantidade de código, “espalhar” as chamadas aos métodos do EntityManager pode gerar dificuldades na manutenção e no entendimento do sistema.

Para melhorar a organização das nossas aplicações, diminuindo o custo de manutenção e aumentando a legibilidade do código, podemos aplicar o padrão Repository do DDD (*Domain Driven Design*).

Conceitualmente, um repositório representa o conjunto de todos os objetos de um determinado tipo. Ele deve oferecer métodos para recuperar e para adicionar elementos.

Os repositórios podem trabalhar com objetos prontos na memória ou reconstruí-los com dados obtidos de um banco de dados. O acesso ao banco de dados pode ser realizado através de ferramentas ORM como o Hibernate.

```

1 class EditoraRepository {
2     private EntityManager manager;
3
4     public EditoraRepository(EntityManager manager) {
5         this.manager = manager;
6     }
7
8     public void adiciona(Editora e) {
9         this.manager.persist(e);
10    }
11    public Editora busca(Long id) {
12        return this.manager.find(Editora.class, id);
13    }
14 }
```

```

14     public List<Editora> buscaTodas() {
15         Query query = this.manager.createQuery("SELECT e FROM Editora e");
16         return query.getResultList();
17     }
18 }
```

Código Java 3.16: EditoraRepository.java

```

1 EntityManagerFactory factory = Persistence.createEntityManagerFactory("K19");
2 EntityManager manager = factory.createEntityManager();
3 EditoraRepository editoraRepository = new EditoraRepository(manager);
4
5 List<Editora> editoras = editoraRepository.buscaTodas();
```

Código Java 3.17: Utilizando um repositório

Exercícios de Fixação

- 17** Implemente um repositório de editoras criando uma nova classe no projeto JPA2-Hibernate.

```

1 class EditoraRepository {
2     private EntityManager manager;
3
4     public EditoraRepository(EntityManager manager) {
5         this.manager = manager;
6     }
7
8     public void adiciona(Editora e) {
9         this.manager.persist(e);
10    }
11    public Editora busca(Long id) {
12        return this.manager.find(Editora.class, id);
13    }
14    public List<Editora> buscaTodas() {
15        Query query = this.manager.createQuery("SELECT e FROM Editora e");
16        return query.getResultList();
17    }
18 }
```

Código Java 3.18: EditoraRepository.java

- 18** Altere a classe InsereEditoraComJPA para que ela utilize o repositório de editoras.

```

1 public class InsereEditoraComJPA {
2
3     public static void main(String[] args) {
4         EntityManagerFactory factory =
5             Persistence.createEntityManagerFactory("livraria-pu");
6
7         EntityManager manager = factory.createEntityManager();
8
9         EditoraRepository editoraRepository = new EditoraRepository(manager);
10
11         Editora novaEditora = new Editora();
12
13         Scanner entrada = new Scanner(System.in);
14
15         System.out.println("Digite o nome da editora: ");
16         novaEditora.setNome(entrada.nextLine());
17 }
```

```
18     System.out.println("Digite o email da editora: ");
19     novaEditora.setEmail(entrada.nextLine());
20
21     editoraRepository.adiciona(novaEditora);
22
23     manager.getTransaction().begin();
24     manager.getTransaction().commit();
25
26     manager.close();
27     factory.close();
28 }
29 }
```

Código Java 3.19: InsereEditoraComJPA.java

- 19 Altere a classe ListaEditorasComJPA para que ela utilize o repositório de editoras.

```
1 public class ListaEditorasComJPA {
2
3     public static void main(String[] args) {
4         EntityManagerFactory factory =
5             Persistence.createEntityManagerFactory("livraria-pu");
6
7         EntityManager manager = factory.createEntityManager();
8
9         EditoraRepository editoraRepository = new EditoraRepository(manager);
10
11         List<Editora> editoras = editoraRepository.buscaTodas();
12
13         for(Editora e : editoras) {
14             System.out.println("EDITORA: " + e.getNome() + " - " + e.getEmail());
15         }
16
17         manager.close();
18         factory.close();
19     }
20 }
```

Código Java 3.20: ListaEditorasComJPA.java



WEB CONTAINER

Necessidades de uma aplicação web

HTTP

Os usuários de uma aplicação web utilizam navegadores (*browsers*) para interagir com essa aplicação. A comunicação entre navegadores e uma aplicação web é realizada através de requisições e respostas definidas pelo protocolo HTTP. Dessa forma, os desenvolvedores de aplicação web devem estar preparados para trabalhar com o protocolo HTTP.

Acesso simultâneo

Além disso, na grande maioria dos casos, as aplicações web devem ser acessadas por diversos usuários ao mesmo tempo. Consequentemente, os desenvolvedores web devem criar ou utilizar algum mecanismo eficiente que permita esse tipo de acesso.

Conteúdo dinâmico

As páginas de uma aplicação web devem ser geradas dinamicamente. Por exemplo, quando um usuário de uma aplicação de email acessa a sua caixa de entrada, ele deseja ver todos os emails enviados até aquele momento. A página contendo a lista de emails deve ser gerada novamente toda vez que essa página for requisitada. Consequentemente, os desenvolvedores web devem criar ou utilizar algum mecanismo eficiente que permita que o conteúdo das páginas das aplicações web seja gerado dinamicamente.

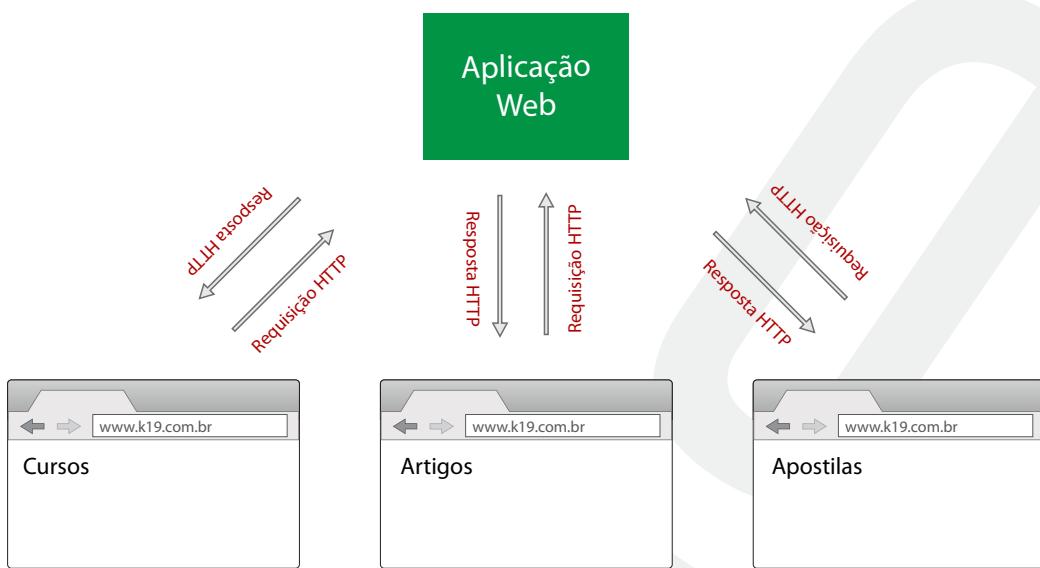


Figura 4.1: Necessidades de uma aplicação web

Solução

Resolver os três problemas apresentados tomaria boa parte do tempo de desenvolvimento, além de exigir conhecimentos técnicos extremamente específicos por parte dos desenvolvedores. Para facilitar o desenvolvimento de aplicações web, a plataforma Java oferece uma solução genérica que pode ser utilizada para desenvolver aplicações web. Conheceremos essa solução a seguir.

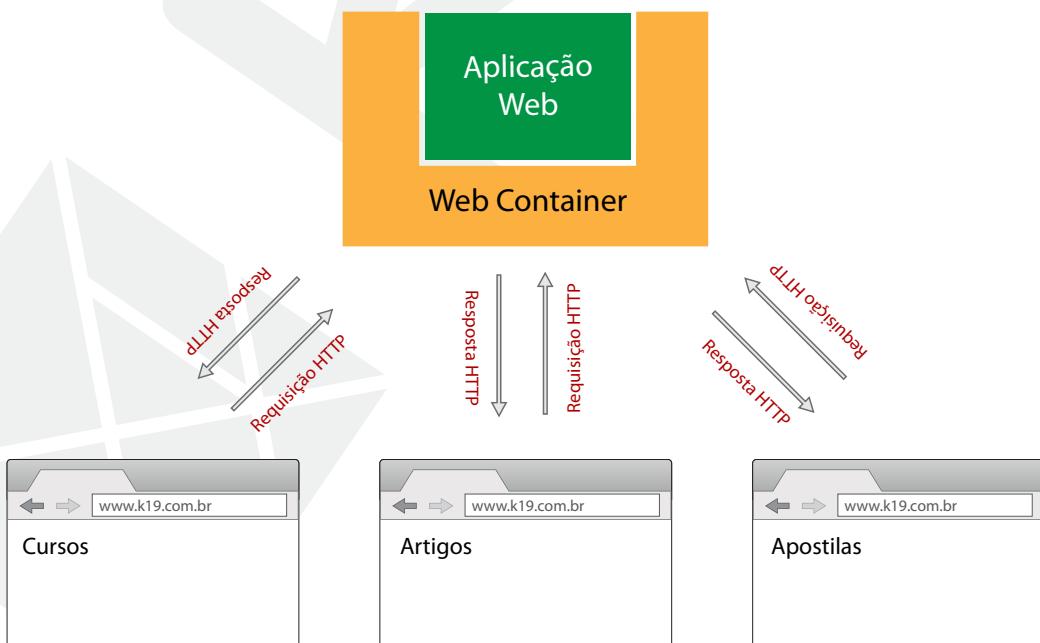


Figura 4.2: Web Container

Web Container

Uma aplicação web Java deve ser implantada em um **Web Container** para obter os recursos fundamentais que ela necessita. Um Web Container é responsável:

- Pelo envio e recebimento de mensagens HTTP.
- Por permitir que as aplicações sejam acessadas simultaneamente por vários usuários de uma maneira eficiente.
- Por permitir que as páginas de uma aplicação web sejam geradas dinamicamente.

Os dois Web Containers mais importantes do mercado são **Tomcat** e **Jetty**. Também podemos utilizar um servidor de aplicação Java EE como o **JBoss**, **Glassfish** ou **WebSphere**, pois eles possuem um Web Container internamente.

Servlet e Java EE

Como é comum na plataforma Java, foi definida uma especificação para padronizar a interface dos recursos oferecidos pelos Web Containers. Essa especificação é chamada **Servlet** e atualmente está na versão 3. Para consultá-la, acesse <http://jcp.org/en/jsr/detail?id=315>.

A especificação Servlet faz parte do **Java EE**. O Java EE é uma especificação que agrupa diversas outras especificações. Para consultá-la, acesse <http://jcp.org/en/jsr/detail?id=316>.

Apesar das especificações, os Web Containers possuem algumas diferenças nas configurações que devem ser realizadas pelos desenvolvedores. Dessa forma, não há 100% de portabilidade. Contudo, a maior parte das configurações e do modelo de programação é padronizado. Sendo assim, se você conhece bem um dos Web Containers, também conhece bastante dos outros.

Neste treinamento, optamos pela utilização do servidor de aplicação Glassfish 3.0. Esse servidor segue a especificação Java EE 6 e portanto contém um Web Container. Mostraremos, a seguir, a sua instalação e configuração.



Mais Sobre

Consulte os artigos da K19 sobre instalação e configuração do Glassfish para mais detalhes.

<http://www.k19.com.br/artigos/installando-glassfish/>

<http://www.k19.com.br/artigos/configurando-o-glassfish-no-ide-eclipse/>



Exercícios de Fixação

- 1 Na Área de Trabalho, entre na pasta K19-Arquivos e copie o arquivo glassfish-3.0.1-with-hi-

bernate.zip para o seu Desktop. Descompacte-o na própria Área de Trabalho.

- 2 Ainda na Área de Trabalho, entre na pasta glassfishv3/glassfish/bin e execute o script startserv para iniciar o Glassfish.
- 3 Verifique se o Glassfish está em execução, acessando <http://localhost:8080> através de um navegador.
- 4 Finalize o Glassfish executando o script stopserv, que está na mesma pasta do script startserv.
- 5 No Eclipse, abra a view “Servers” e clique com o botão direito no corpo dela. Escolha a opção “new” e configure o Glassfish.
- 6 Inicialize o Glassfish pela view “Servers” e verifique se ele está funcionando, acessando <http://localhost:8080>.
- 7 Finalize o Glassfish pela view “Servers”.

Aplicação Web Java

Para que uma aplicação web Java possa ser implantada em um Web Container, a estrutura de pastas precisa seguir algumas regras.

```
▷ K19-App/
  ▷ WEB-INF/
    ▷ classes/
    ▷ lib/
  ▷ web.xml
```

A pasta K19-App é a raiz da aplicação. Ela pode ter qualquer nome. A pasta WEB-INF deve ser criada dentro da pasta raiz. O conteúdo da pasta WEB-INF não pode ser acessado diretamente pelos usuários da aplicação. Por outro lado, os arquivos dentro da pasta raiz da aplicação mas fora da pasta WEB-INF podem ser acessados diretamente através de um navegador.

As pastas classes e lib devem ser criadas dentro da pasta WEB-INF. O código compilado da aplicação deve ser salvo na pasta classes. Os jar's das bibliotecas extras que serão utilizadas devem ser colocados na pasta lib. O arquivo web.xml contém configurações do Web Container e deve ser criado na pasta WEB-INF.

Em geral, as IDEs criam toda a estrutura de pastas exigidas pelos Web Containers. Então, na prática, não temos o trabalho de criar esses diretórios manualmente.



Mais Sobre

Consulte o artigo da K19 sobre criação de projetos web utilizando o Eclipse.

<http://www.k19.com.br/artigos/criando-um-dynamic-web-project/>



Exercícios de Fixação

- 8 No Eclipse, crie um projeto do tipo *Dynamic Web Project* chamado **K19-App**. Escolha “Glassfish” como opção para “Target runtime”. Na última tela de criação do projeto, selecione a opção **Generate web.xml deployment descriptor**.
- 9 Adicione o projeto K19-App no Glassfish através da view “Servers”. Clique com o botão direito do mouse no Glassfish e selecione “Add and Remove...”.
- 10 Inicialize o Glassfish através da view “Servers”. Clique com o botão direito do mouse sobre o Glassfish e escolha a opção “Start”.
- 11 Verifique o funcionamento da nossa aplicação acessando <http://localhost:8080/K19-App/> através de um navegador.

Processando requisições

Após implantar a nossa aplicação web Java em um Web Container, as requisições e respostas HTTP já estão sendo processadas pelo Web Container, que também já permite o acesso de múltiplos usuários à nossa aplicação.

Em seguida, devemos definir como o conteúdo das páginas da aplicação é gerado. Para isso, podemos criar uma **Servlet**.

Servlet

Para criar uma Servlet, podemos seguir os seguintes passos:

1. Criar uma classe.
2. Herdar da classe `javax.servlet.http.HttpServlet`.
3. Reescrever o método `service()`.
4. Utilizar a anotação `@WebServlet` para definir a url que será utilizada para acessar a Servlet. Essa anotação existe após a especificação Servlet 3.0. Antes, essa configuração era realizada através do arquivo `web.xml`.

```

1 @WebServlet("/OlaMundo")
2 public class OlaMundo extends HttpServlet{
3
4     @Override
5     protected void service(HttpServletRequest req, HttpServletResponse resp)
6         throws ServletException, IOException {
7         // Lógica para processar as regras de negócio e gerar conteúdo
8     }
9 }
```

Código Java 4.1: *OlaMundo.java*

O método `service()` é executado toda vez que uma requisição HTTP é realizada para a url definida na anotação `@WebServlet`. Esse método recebe dois parâmetros. O primeiro é a referência do objeto da classe `HttpServletRequest` que armazena todos os dados da requisição. O segundo parâmetro é a referência do objeto da classe `HttpServletResponse` que armazenará o conteúdo gerado pela Servlet.

Inserindo conteúdo na resposta

Para inserir conteúdo texto na resposta HTTP que será enviada para o navegador do usuário, devemos utilizar os métodos `getWriter()` e `println()`. Em geral, o conteúdo inserido na resposta HTTP é texto HTML. Veja o código abaixo.

```
1 @WebServlet("/OlaMundo")
2 public class OlaMundo extends HttpServlet {
3
4     @Override
5     protected void service(HttpServletRequest req, HttpServletResponse resp)
6         throws ServletException, IOException {
7         PrintWriter writer = resp.getWriter();
8         writer.println("<html><body><h1>Olá Mundo</h1></body></html>");
9     }
10 }
```

Código Java 4.2: *OlaMundo.java*



Exercícios de Fixação

- 12 Crie um pacote chamado `servlets` no projeto K19-App.
- 13 Crie uma classe chamada `OlaMundo` no pacote `servlets`.

```
1 @WebServlet("/OlaMundo")
2 public class OlaMundo extends HttpServlet {
3
4     @Override
5     protected void service(HttpServletRequest req, HttpServletResponse resp)
6         throws ServletException, IOException {
7         PrintWriter writer = resp.getWriter();
8         writer.println("<html><body><h1>Olá Mundo</h1></body></html>");
9     }
10 }
```

Código Java 4.3: *OlaMundo.java*

- 14 Verifique o funcionamento da Servlet acessando a url abaixo através de um navegador.

<http://localhost:8080/K19-App/OlaMundo>

Frameworks

Hoje em dia, é improvável que uma empresa decida começar um projeto utilizando diretamente Servlets, pois a produtividade seria pequena e a manutenção muito custosa. Vários frameworks fo-

ram criados para facilitar o desenvolvimento e a manutenção de aplicações web. Apesar de serem baseados em Servlets, esses frameworks oferecem diversos recursos adicionais para as aplicações. Eis uma lista dos principais frameworks para aplicações web Java:

- JSF
- Struts 1.x
- Struts 2.x
- Spring MVC

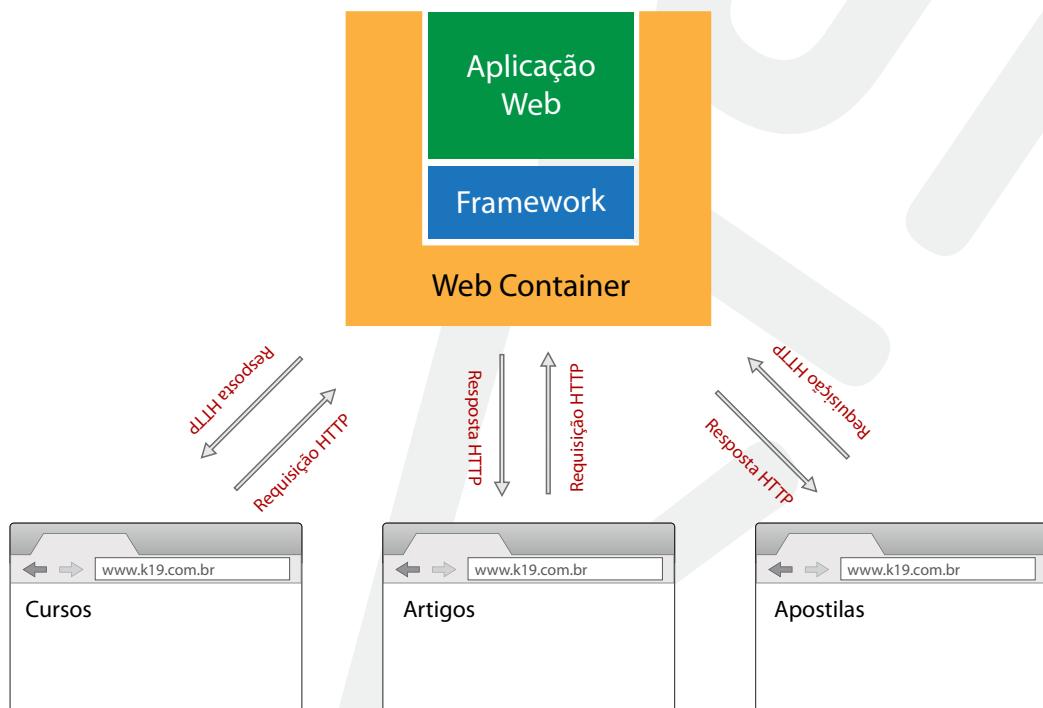


Figura 4.3: Framework para aplicações web

Nos próximos capítulos, mostraremos o funcionamento e explicaremos os conceitos relacionados ao framework JSF.



VISÃO GERAL DO JSF 2

Atualmente, o principal framework para desenvolvimento de aplicações web em Java é o JSF (*Java Server Faces*). A especificação desse framework pode ser obtida em <http://www.jcp.org/en/jsr/detail?id=314>. Além disso, recomendamos a consulta ao javadoc da API do JSF que pode ser obtido no mesmo link. O JSF é fortemente baseado nos padrões *MVC* e *Front Controller*.

MVC e Front Controller

O MVC (*model-view-controller*) é um padrão de arquitetura que tem por objetivo isolar a lógica de negócios da lógica de apresentação de uma aplicação.

Esse padrão (ou alguma variação) é amplamente adotado nas principais plataformas de desenvolvimento atuais. Em particular, ele é bastante utilizado na plataforma Java.

O padrão MVC divide uma aplicação em três tipos de componentes: modelo, visão e controlador.

Modelo: encapsula os dados e as funcionalidades da aplicação.

Visão: é responsável pela exibição de informações, cujos dados são obtidos do modelo.

Controlador: recebe as requisições do usuário e aciona o modelo e/ou a visão.

Para mais detalhes sobre o padrão MVC, uma boa referência é o livro *Pattern-Oriented Software Architecture Volume 1: A System of Patterns* (editora Wiley, 1996) dos autores Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal e Michael Stal.

No padrão *Front Controller*, todas as requisições do usuário são recebidas pelo mesmo componente. Dessa forma, tarefas que devem ser realizadas em todas as requisições podem ser implementadas por esse componente. Isso evita a repetição de código e facilita a manutenção do sistema.

Para mais informações sobre esse padrão, consulte, por exemplo, o livro *Core J2EE Patterns: Best Practices and Design Strategies* (editora Prentice Hall, 2003, segunda edição) dos autores Deepak Alur, Dan Malks e John Crupi.

Configurando uma aplicação JSF

Uma aplicação JSF deve respeitar a estrutura geral de uma aplicação web Java que foi descrita no capítulo anterior. Essa estrutura é definida pela especificação *Servlet* que está disponível em <http://www.jcp.org/en/jsr/detail?id=315>.

web.xml

A Faces Servlet deve ser configurada no arquivo WEB-INF/web.xml, indicando a classe que a implementa e o padrão de url que será associado a essa servlet.

Por exemplo, na configuração abaixo, todas as requisições cujas urls possuam o sufixo .xhtml serão processadas pela Faces Servlet.

```

1 <servlet>
2   <servlet-name>Faces Servlet</servlet-name>
3   <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
4 </servlet>
5 <servlet-mapping>
6   <servlet-name>Faces Servlet</servlet-name>
7   <url-pattern>*.xhtml</url-pattern>
8 </servlet-mapping>
```

Código XML 5.1: web.xml

faces-config.xml

Devemos adicionar um arquivo chamado faces-config.xml no diretório WEB-INF. Nesse arquivo, podemos alterar diversas configurações do JSF. Mas, a princípio, podemos deixá-lo apenas com a tag faces-config.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/→
5     javaee/web-facesconfig_2_0.xsd"
6   version="2.0">
7 </faces-config>
```

Código XML 5.2: faces-config.xml

Bibliotecas

Para utilizar os recursos do JSF, é necessário que a aplicação possua uma implementação JSF. Essa implementação pode ser adicionada manualmente no diretório WEB-INF/lib da aplicação.

Se um servidor de aplicação Java EE for utilizado, a inclusão de uma implementação JSF manualmente não é necessária, já que esse tipo de servidor possui, por padrão, uma implementação de JSF.

Managed Beans

Os managed beans são objetos fundamentais de uma aplicação JSF. Suas principais tarefas são:

1. Fornecer dados que serão exibidos nas telas.
2. Receber os dados enviados nas requisições.

3. Executar tarefas de acordo com as ações dos usuários.

Criando um Managed Bean

Um managed bean pode ser definido de duas maneiras. A primeira maneira é criar uma classe Java e registrá-la no arquivo `faces-config.xml`. Veja o exemplo abaixo.

```
1 package br.com.k19;
2
3 public class TesteBean {
4     ...
5 }
```

Código Java 5.1: TesteBean.java

```
1 ...
2 <managed-bean>
3     <managed-bean-name>testeBean</managed-bean-name>
4     <managed-bean-class>br.com.k19.TesteBean</managed-bean-class>
5     <managed-bean-scope>request</managed-bean-scope>
6 </managed-bean>
7 ...
```

Código XML 5.3: faces-config.xml

No registro de um managed bean, devemos definir o nome, a classe e o escopo do managed bean. O nome será utilizado para acessar esse managed bean nas páginas da aplicação. O escopo será explicado em detalhes no Capítulo 9.

A segunda forma é criar uma classe Java com a anotação `@ManagedBean` do pacote `javax.faces.bean`. Essa anotação só pode ser utilizada a partir da versão 2 do JSF. Observe o exemplo abaixo.

```
1 @ManagedBean
2 public class TesteBean {
3     ...
4 }
```

Código Java 5.2: TesteBean.java

Utilizando a anotação `@ManagedBean`, por padrão, o JSF assumirá que o nome do managed bean é o nome da classe com a primeira letra minúscula. Para o exemplo acima, o nome padrão do managed bean é `testeBean`. Além disso, o escopo `request` será assumido como padrão.

Propriedades

Considere o seguinte managed bean.

```
1 @ManagedBean
2 public class TesteBean {
3     private int numero;
4 }
```

Código Java 5.3: TesteBean.java

Para acessar o valor do atributo numero em uma tela JSF precisamos definir um método de leitura. Esse método deve seguir a convenção de nomenclatura do Java. Veja o exemplo abaixo:

```

1  @ManagedBean
2  public class TesteBean {
3      private int numero;
4
5      public int getNumero() {
6          return numero;
7      }
8 }
```

Código Java 5.4: TesteBean.java

Note que o nome do método começa com `get` e é seguido pelo nome do atributo com a primeira letra em caixa alta.

Para alterar o valor do atributo numero com valores obtidos através de uma tela JSF, precisamos definir um método de escrita.

```

1  @ManagedBean
2  public class TesteBean {
3      private int numero;
4
5      public int setNumero(int numero) {
6          this.numero = numero;
7      }
8
9      public int getNumero() {
10         return numero;
11     }
12 }
```

Código Java 5.5: TesteBean.java

O nome do método de escrita deve necessariamente começar com a palavra `set` e terminar com o nome do atributo com a primeira letra em caixa alta.

Com os métodos de acesso já implementados, podemos exibir o valor do atributo numero utilizando *expression language* (`#{}`). Veja o exemplo a seguir.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7  <h:head>
8      <title>K19 Treinamentos</title>
9  </h:head>
10
11 <h:body>
12     Valor: #{testeBean.numero}
13 </h:body>
14 </html>
```

Código XHTML 5.1: Exibindo o valor do atributo numero

Para alterar o valor do atributo numero do managed bean testeBean, podemos vinculá-lo, por exemplo, a uma caixa de texto em um formulário. Observe o código abaixo.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7  <h:head>
8   <title>K19 Treinamentos</title>
9  </h:head>
10
11 <h:body>
12
13   Valor: #{testeBean.numero}
14
15  <h:form>
16   <h:inputText value="#{testeBean.numero}" />
17   <h:commandButton value="Altera"/>
18  </h:form>
19 </h:body>
20 </html>

```

Código XHTML 5.2: Alterando o valor do atributo numero



Importante

É importante destacar que o vínculo com uma propriedade de um managed bean dá-se por meio dos nomes dos métodos getters e setters, e não pelo nome do atributo.

No exemplo acima, se mantivéssemos o nome do atributo do managed bean mas substituíssemos os nomes dos métodos getNumero() e setNumero() por getValor() e setValor(), respectivamente, então os trechos de código XHTML em destaque deveriam ser alterados para o seguinte:

```

1  Valor: #{testeBean.valor}
2  <h:inputText value="#{testeBean.valor}" />

```

Ações

Para implementar as lógicas que devem ser executadas assim que o usuário clicar em um botão ou link, basta criar métodos nas classes dos managed beans.

Seguindo o exemplo acima, podemos criar um método que incrementa o valor do atributo numero, como no código abaixo.

```

1  @ManagedBean
2  public class TesteBean {
3   private int numero;
4
5   public void incrementaNumero() {
6     this.numero = numero + 1;
7   }
8
9   // GETTERS E SETTERS
10 }

```

Código Java 5.6: TesteBean.java

Esses métodos podem ser void quando desejamos manter os usuários na mesma tela ou devolver String quando desejamos realizar uma navegação entre telas. Veremos mais detalhes em capítulos posteriores.

Para incrementar o valor do atributo numero do managed bean testeBean, podemos criar uma página que possui um botão para executar essa ação e associá-lo ao método incrementaNumero(), conforme o código abaixo.

```
1 <h:form>
2   <h:commandButton value="Incrementa" action="#{testeBean.incrementaNumero}" />
3 </h:form>
```

Código XHTML 5.4: Botão associado a um método de um managed bean



Mais Sobre

Todo managed bean possui um nome único que é utilizado para acessá-lo dentro dos trechos escritos com expression language. Quando utilizamos a anotação @ManagedBean, por padrão, o JSF assume que o nome do managed bean é o nome da classe com a primeira letra em caixa baixa. Porém, podemos alterar esse nome acrescentando um argumento na anotação.

```
1 @ManagedBean(name="teste")
2 class TesteBean {
3   ...
4 }
```

Código Java 5.7: Alterando o nome de um managed bean

Processamento de uma requisição

Em uma aplicação JSF, toda requisição realizada através de um navegador é recebida pela *Faces Servlet*. Ao receber uma requisição, a Faces Servlet executa os seguintes passos:

Restore View: Na primeira requisição de um usuário, uma nova árvore de componentes que representa a tela desse usuário é gerada. Nas demais requisições (*postback*) desse mesmo usuário, a árvore de componentes que representa a tela anteriormente enviada a ele é reconstruída.

Apply Request Values: Nesta etapa, a árvore construída na etapa anterior é percorrida e cada um dos seus componentes é “decodificado”. No processo de decodificação, cada componente extrai da requisição atual os dados associados a essa componente e se atualiza com essas informações.

Ainda nessa etapa, os eventos de ação (como um clique em um *link* ou em um botão, por exemplo) são identificados. Por padrão, esses eventos são adicionados às filas correspondentes para serem tratados posteriormente (na fase *Invoke Application*).

Process Validations: Nesta fase, os componentes que possuem valores submetidos pelo usuário através de formulários são convertidos e validados (caso haja algum validador registrado para esse componente).

Se ocorrer algum erro de conversão ou validação, mensagens de erro são adicionadas no contexto da requisição atual e o fluxo é redirecionado para a fase *Render Response*. Caso contrário, processo continua na fase *Update Model Values*.

Ainda nesta fase, os eventos de mudança de valor são identificados e tratados ao final da mesma.

Update Model Values: Os valores contidos em cada componente da árvore, já convertidos e validados na fase anterior, são armazenados em propriedades de objetos definidos pela aplicação (managed beans)

Invoke Application: Uma vez que os dados dos componentes já foram convertidos, validados e armazenados nos objetos do modelo, as tarefas correspondentes ao evento que disparou a requisição (normalmente um clique em um botão ou link) serão executadas.

Também nesta fase, a próxima tela a ser apresentada ao usuário é determinada pelo resultado do método que implementa a lógica de negócios executado nesta fase.

Render Response: Nesta etapa, a próxima tela é gerada e enviada ao navegador do usuário. Uma representação desta tela também é armazenada a fim de ser usada na fase *Restore View* da próxima requisição.

O diagrama abaixo ilustra a estrutura geral de uma aplicação JSF. O processamento de uma requisição enviada por um navegador começa na Faces Servlet logo após a sua chegada. A Faces Servlet controla a execução das seis etapas descritas acima.

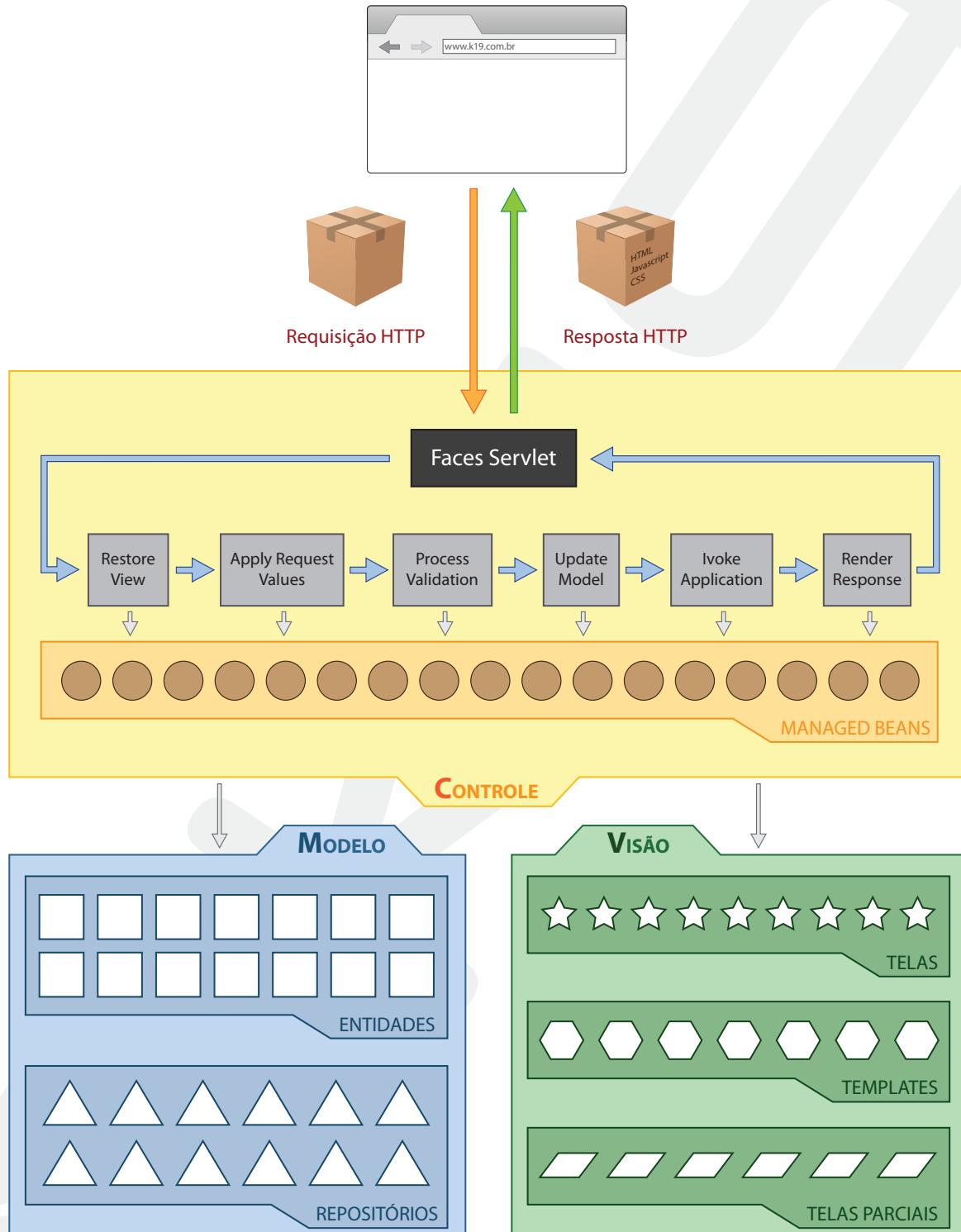


Figura 5.1: Estrutura geral de uma aplicação JSF

Os managed beans estão à disposição da Faces Servlet durante todo o processamento da requisição. Por exemplo, nas etapas *Render Response* e *Restore View*, a Faces Servlet aciona os managed beans para recuperar os dados que devem ser usados na construção ou reconstrução da árvore de componentes. Na etapa *Update Model*, a Faces Servlet armazena nos managed beans os dados já convertidos e validados. Na etapa *Invoke Application*, a Faces Servlet dispara um método em um managed bean responsável pelo processamento da regra de negócio correspondente à requisição atual.

Todas as regras de negócio são implementadas no modelo, que também administra os dados da aplicação. Os managed beans acionam o modelo para executar alguma regra de negócio, recuperar ou alterar os dados administrados pelo modelo.

As telas da aplicação são definidas na camada de visão. A Faces Servlet acessa essa camada toda vez que necessita construir ou reconstruir a árvore de componentes de uma determinada tela. Isso ocorre nas etapas *Restore View* e *Render Response*.

Exemplo Prático

Com as configurações já realizadas, implementaremos uma aplicação que mostra o funcionamento básico do JSF. Essa aplicação deverá receber um texto do usuário e exibi-lo em letras maiúsculas.

Managed Bean

Vamos começar criando um managed bean para armazenar o texto enviado pelo usuário e a lógica para transformá-lo.

```

1 import javax.faces.bean.ManagedBean;
2
3 @ManagedBean
4 public class TextoBean {
5     private String texto;
6
7     public String transformaEmCaixaAlta() {
8         this.texto = this.texto.toUpperCase();
9         return "resposta";
10    }
11
12    public String getTexto() {
13        return this.texto;
14    }
15
16    public void setTexto(String texto) {
17        this.texto = texto;
18    }
19 }
```

Código Java 5.8: TextoBean.java

A classe que implementa o managed bean deve ser anotada com `@ManagedBean`. O atributo `texto` armazenará o texto enviado pelo usuário e esse texto será modificado pelo método `transformaEmCaixaAlta()`. Esse método devolve uma string para indicar qual deve ser a próxima tela a ser enviada para o usuário.

A Faces Servlet utilizará o método `setTexto()` para armazenar o texto enviado pelo usuário no managed bean. Por outro lado, utilizará o método `getTexto()` para recuperar o texto e exibi-lo após a sua modificação.

Telas

Uma vez que o managed bean foi criado, podemos associá-lo a um formulário que receberá o texto do usuário.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14 <h:form>
15 <h:outputLabel value="Texto: " />
16 <h:inputTextarea value="#{textoBean.texto}" />
17 <h:commandButton value="Transformar" action="#{textoBean.transformaEmCaixaAlta}" />
18 </h:form>
19 </h:body>
20 </html>
```

Código XHTML 5.5: *formulario.xhtml*

Observe nas linhas em destaque a ligação entre essa tela e o managed bean. A caixa de entrada de texto foi associada à propriedade `texto` do managed bean `TextoBean`. O botão, por sua vez, foi associado ao método `transformaEmCaixaAlta()` do managed bean `TextoBean`.

Para exibir o texto transformado, podemos criar uma outra tela.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14 <h:outputText value="#{textoBean.texto}" />
15 </h:body>
16 </html>
```

Código XHTML 5.6: *resposta.xhtml*

Analogamente, a caixa de saída de texto está associada à propriedade `texto` do managed bean `TextoBean`.



Exercícios de Fixação

Como exercício, desenvolva uma aplicação JSF que (i) receba um número inteiro do usuário, (ii) produza um número aleatório entre zero e o número recebido, e (iii) exiba esse número na tela do navegador do usuário.

- 1 No Eclipse, crie um Dynamic Web Project chamado **K19-Visao-Geral**. Na primeira tela, devemos definir o nome do projeto, selecionar o target runtime e a configuração.

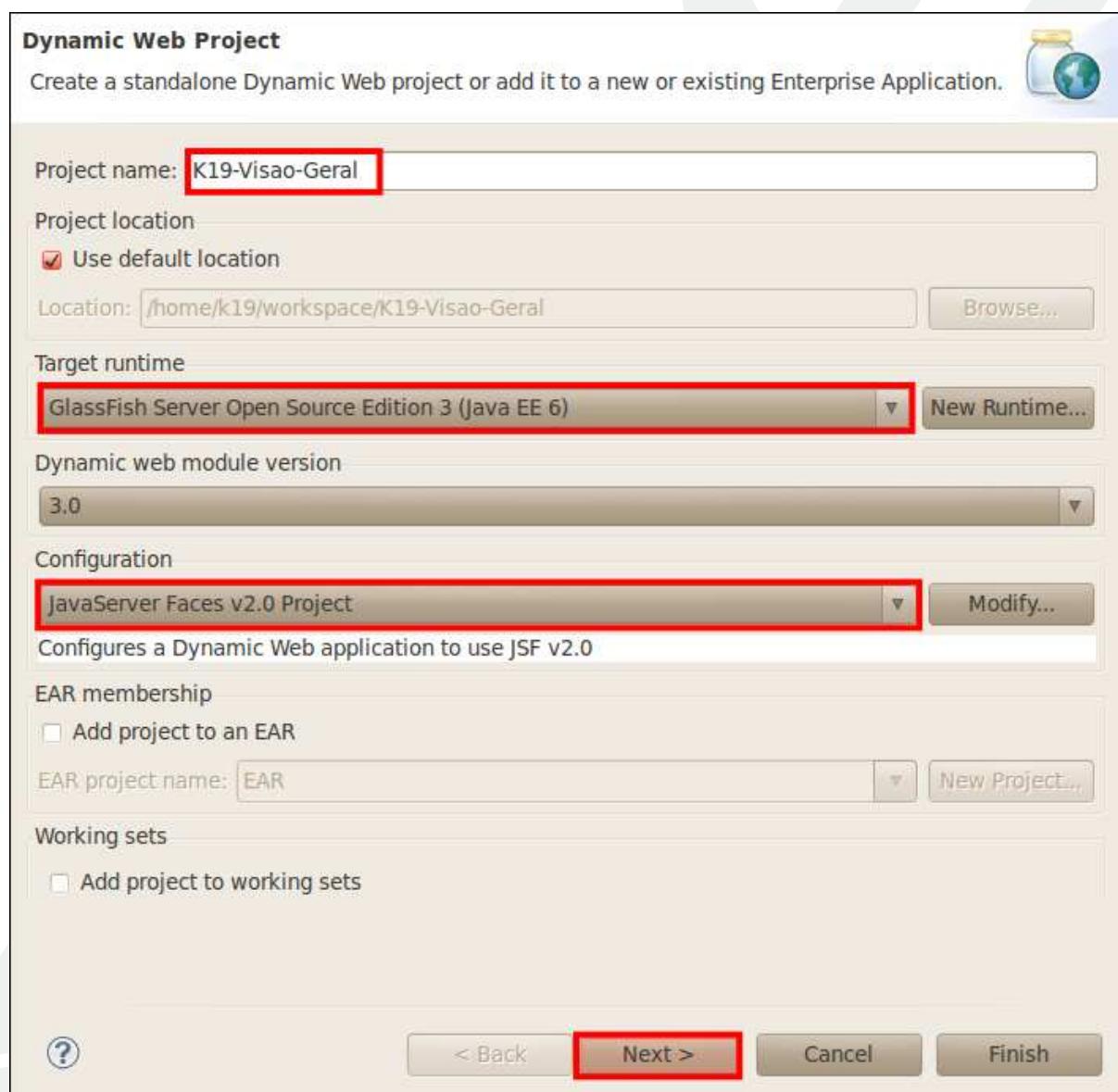


Figura 5.2: Criando um Dynamic Web Project

Na tela seguinte, apenas clique em “Next”.

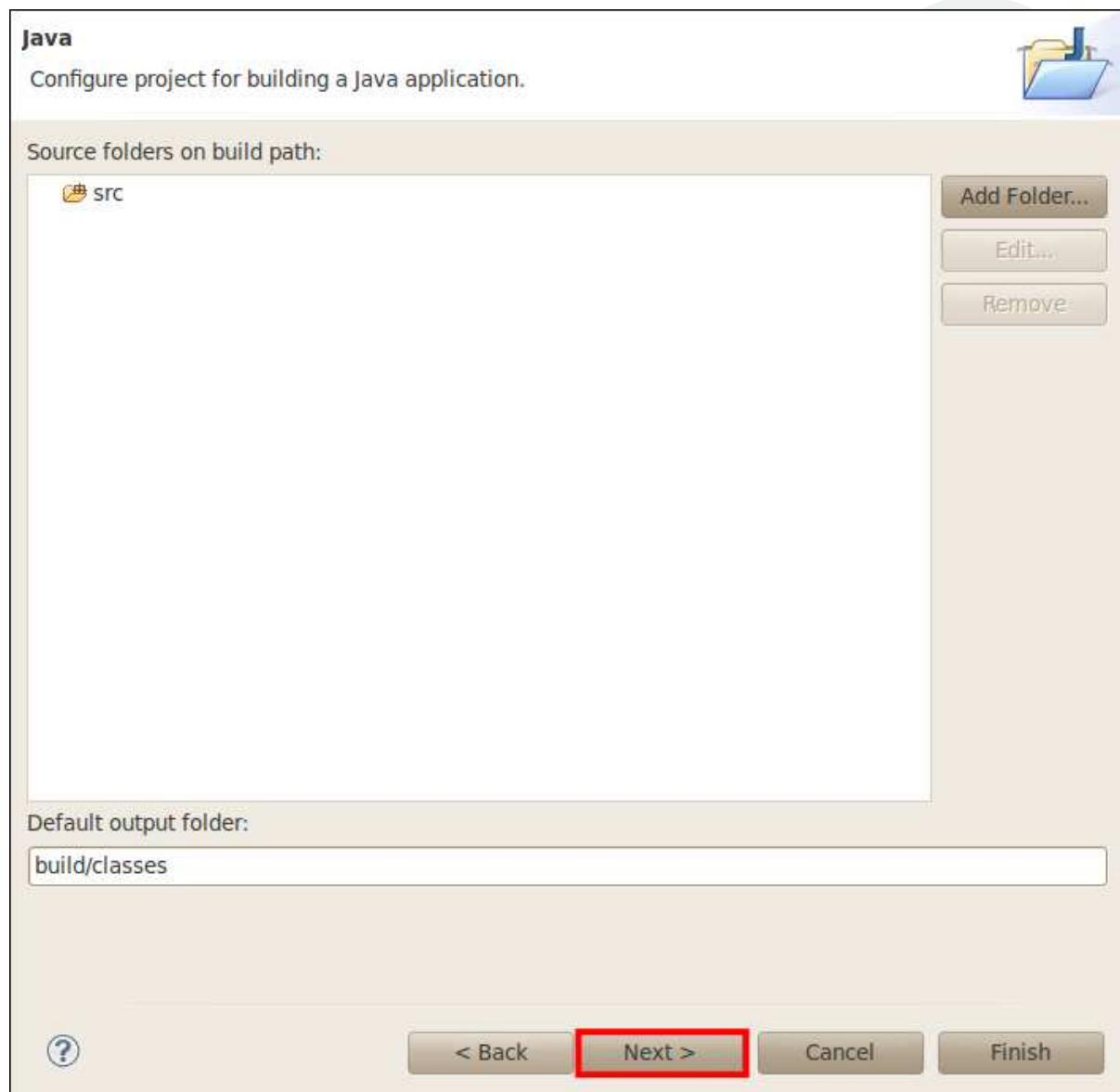


Figura 5.3: Criando um Dynamic Web Project

Novamente, clique em “Next”.

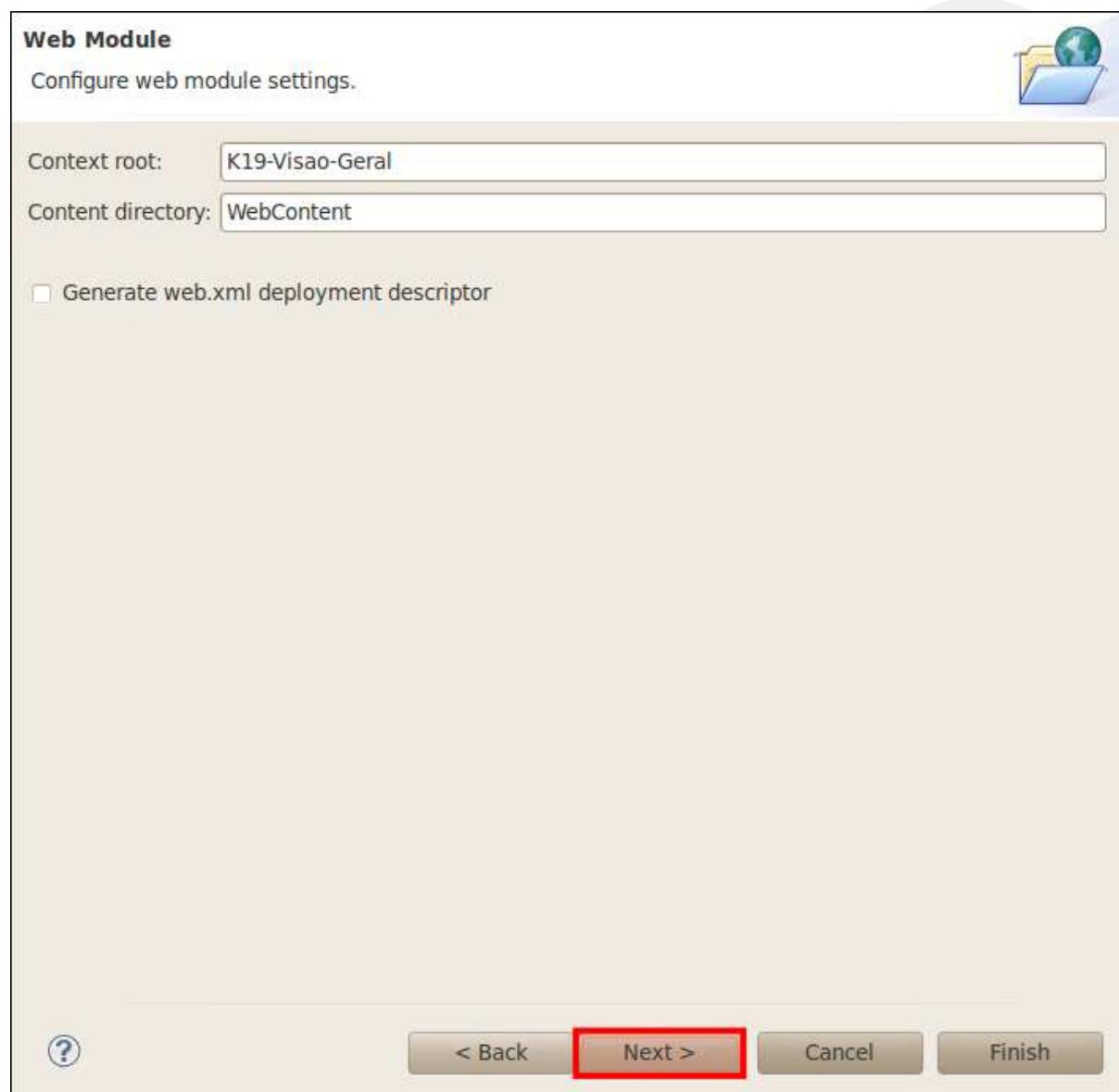


Figura 5.4: Criando um Dynamic Web Project

Na última tela, devemos desabilitar as configurações de biblioteca e mapear a Faces Servlet para as urls que seguem o padrão “*.xhtml”.

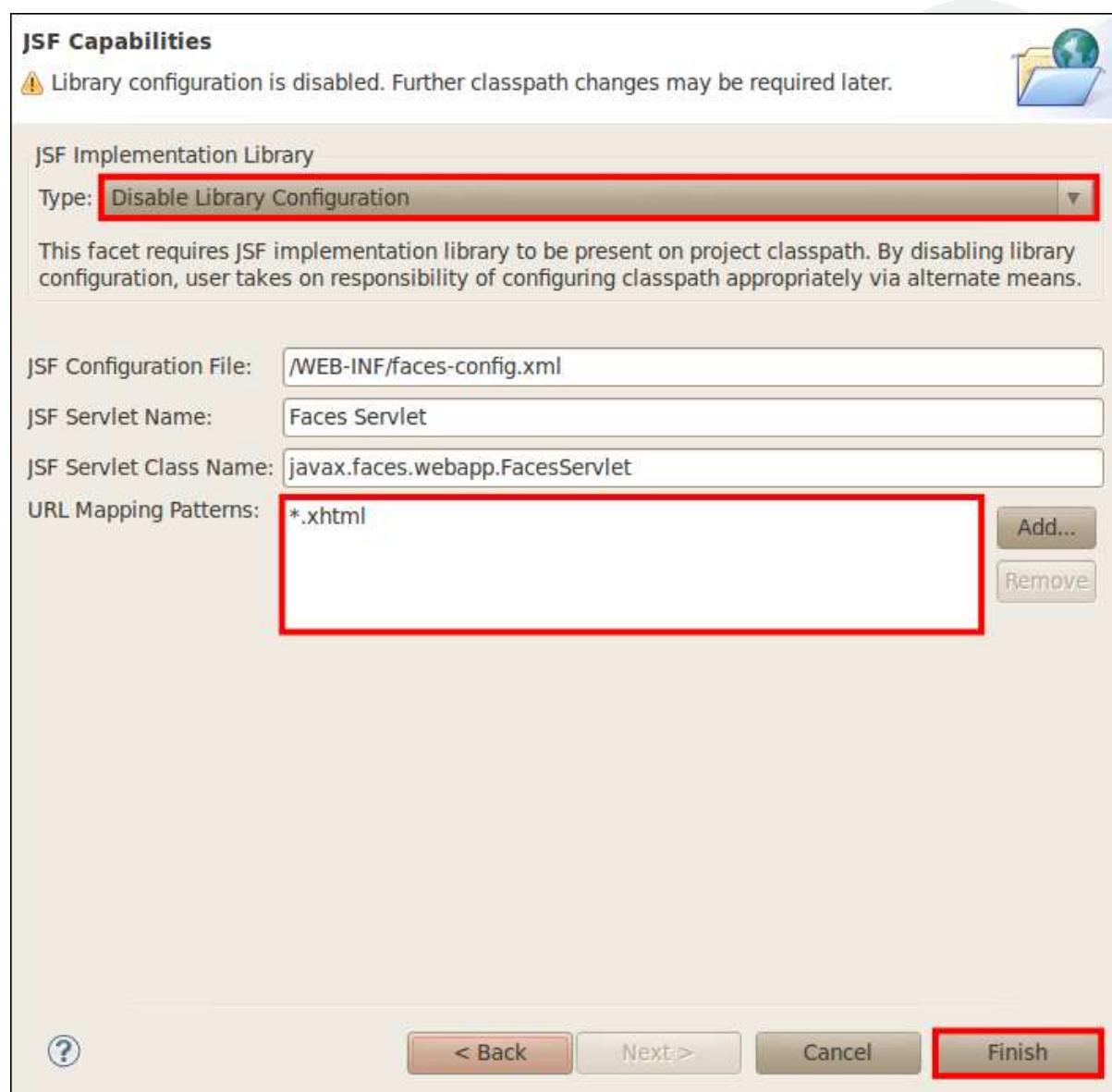


Figura 5.5: Criando um Dynamic Web Project

- 2 Na pasta src, crie um managed bean para armazenar o número inteiro n enviado pelo usuário, gerar um número aleatório entre zero e n e armazená-lo numa propriedade.

```

1 import javax.faces.bean.ManagedBean;
2
3 @ManagedBean
4 public class NumeroAleatorioBean {
5     private int maximo;
6     private int numeroAleatorio;
7
8     public String geraNumeroAleatorio() {
9         this.numeroAleatorio = (int) (Math.random() * this.maximo);
10        return "resposta";
11    }
12
13    // GETTERS E SETTERS
14 }
```

Código Java 5.9: NumeroAleatorioBean.java

- 3** Agora, na pasta WebContent, crie um formulário para que o usuário possa enviar o dado de entrada.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14   <h:form>
15     <h:outputLabel value="Número máximo: " />
16     <h:inputText value="#{numeroAleatorioBean.maximo}" />
17     <h:commandButton value="Gera número aleatório"
18       action="#{numeroAleatorioBean.geraNumeroAleatorio}" />
19   </h:form>
20 </h:body>
21 </html>
```

Código XHTML 5.7: formulario.xhtml

- 4** Também na pasta WebContent, defina uma tela para exibir o número gerado aleatoriamente.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14   <h:outputText value="#{numeroAleatorioBean.numeroAleatorio}" />
15 </h:body>
16 </html>
```

Código XHTML 5.8: resposta.xhtml

- 5** Adicione o projeto no GlassFish e teste-o, seguindo os passos abaixo.

1. Abra a aba “Servers”. Para isso, pressione “Ctrl+3”. Na janela que abrir, digite “Servers” e pressione “Enter”.
2. Na aba “Servers”, clique com o botão direito do mouse no GlassFish e selecione a opção “Add and Remove...”.
3. Selecione o projeto “K19-Visao-Geral” e clique em “Add >”. Em seguida, clique em “Finish”.

4. Inicialize o servidor. Para isso, clique mais uma vez no GlassFish com o botão direito e selecione “Start”.
5. Acesse a aplicação no endereço
<http://localhost:8080/K19-Visao-Geral/formulario.xhtml>.

COMPONENTES VISUAIS

No JSF 2, as telas são definidas em arquivos XHTML. Os componentes visuais que constituem as telas são adicionados por meio de tags. A especificação do JSF define uma grande quantidade de tags e as classifica em bibliotecas. As principais bibliotecas de tags do JSF são:

- Core (<http://java.sun.com/jsf/core>)
- HTML (<http://java.sun.com/jsf/html>)
- Facelets (<http://java.sun.com/jsf/facelets>)

A seguir, apresentaremos o funcionamento geral das principais tags dessas três bibliotecas. A documentação completa pode ser encontrada em <http://www.jcp.org/en/jsr/detail?id=314>.

Estrutura Básica de uma Página JSF

A estrutura básica de uma página JSF é muito semelhante à estrutura de uma página HTML. Observe o código abaixo.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10   <title>K19 Treinamentos</title>
11  </h:head>
12
13 <h:body>
14   <h:outputText value="Estrutura básica de uma tela JSF" />
15 </h:body>
16 </html>
```

Código XHTML 6.1: Estrutura Básica de uma Tela JSF

O conteúdo da página é definido no corpo da tag `<html>`. Esse conteúdo é dividido em duas partes: o cabeçalho, delimitado pela tag `<h:head>` (e não pela tag `<head>`), e o corpo, delimitado pela tag `<h:body>` (e não pela tag `<body>`).

As bibliotecas de tags que serão utilizadas para construir a página devem ser “importadas” através do pseudo-atributo `xmlns` aplicado à tag `<html>`. Observe no exemplo acima que as três principais bibliotecas do JSF foram importadas.



Mais Sobre

Devemos utilizar a declaração `<!DOCTYPE>` para informar aos navegadores qual é a versão do HTML utilizado na página. Com essa informação, os navegadores podem processar o documento corretamente.



Mais Sobre

Quando uma biblioteca de tags é “importada” através do pseudo-atributo `xmlns`, é possível definir um prefixo para essa biblioteca. Esse prefixo é utilizado, no documento, para acessar as tags da biblioteca.



Importante

A utilização das tags `<h:head>` e `<h:body>` é fundamental para o funcionamento completo das páginas HTML geradas pelo JSF. Ao processar essas tags, na etapa Render Response, o JSF adiciona recursos como scripts e arquivos de estilo na tela HTML que será enviada para o usuário. Esses recursos são necessários para o funcionamento correto dos componentes.

Formulários

Os formulários no JSF são definidos através da tag `<h:form>` (e não pela tag `<form>`). Um formulário é composto basicamente por caixas de texto, caixas de seleção, rótulos, botões e links. Ao ser processado, o componente `<h:form>` gera um formulário HTML configurado para o método POST do HTTP.

Nas seções a seguir, descreveremos os elementos de um formulário.

Caixas de Texto

O JSF define três tipos de caixas de texto para coletar dados do usuário.

- `<h:inputText>`
Permite que o usuário digite uma linha de texto.
- `<h:inputTextarea>`
Permite que o usuário digite várias linhas de texto.
- `<h:inputSecret>`
Igual ao `<h:inputText>` mas não exibe o que foi digitado.

Uma caixa de texto pode ser associada a uma propriedade de um managed bean através do atributo `value`. Esse vínculo é denominado **binding**. Considere o seguinte managed bean.

```
1 @ManagedBean
2 public class UsuarioBean {
3     private String nome;
```

```

4   public String getNome() {
5       return nome;
6   }
7
8   public void setNome(String nome) {
9       this.nome = nome;
10  }
11 }
12 }
```

Código Java 6.1: UsuarioBean.java

Devemos utilizar *expression language* (#{}) para estabelecer uma associação entre a propriedade nome a uma caixa de texto. Observe o código abaixo.

```
1 <h:inputText value="#{usuarioBean.nome}" />
```

Quando o JSF constrói a tela a ser enviada para o usuário, ele recupera o valor da propriedade nome através do método getNome() e guarda esse valor na caixa de texto correspondente.

Analogamente, no processamento de uma requisição, o JSF extrai o valor presente na caixa de texto e utiliza o método setNome() para armazenar esse valor na propriedade nome do managed bean UsuarioBean.

Rótulos

Rótulos são utilizados para indicar ao usuário o que ele deve preencher em determinada caixa de texto. Para acrescentar um rótulo em uma página, devemos utilizar o componente <h:outputLabel>. O texto do rótulo é definido pelo atributo value e pode ser associado a uma caixa de texto através do atributo for. O valor do atributo for deve ser igual ao valor do atributo id da caixa de texto que desejamos associar ao rótulo.

```
1 <h:outputLabel value="Nome: " for="nome" />
2 <h:inputText id="nome"/>
```

Código XHTML 6.3: Exemplo de uso do <h:outputLabel>

Exemplos

A seguir, alguns exemplos de utilização das caixas de texto serão apresentados. Mostraremos o código XHTML das telas, o código HTML produzido pelo JSF e as telas renderizadas por um navegador.

1. XHTML:

```
<h:outputLabel value="Nome: " for="nome" />
<h:inputText value="Jonas K. Hirata" id="nome" />
```

HTML gerado:

```
<label for="j_idt6:nome">Nome: </label>
<input id="j_idt6:nome" type="text" name="j_idt6:nome" value="Jonas K. Hirata" />
```

Resultado:

Nome: Jonas K. Hirata

2. XHTML:

```
<h:outputLabel value="Nome: " for="nome" />
<h:inputText value="#{usuarioBean.nome}" id="nome" />
```

HTML gerado:

```
<label for="j_idt6:nome">Nome: </label>
<input id="j_idt6:nome" type="text" name="j_idt6:nome" value="Jonas K. Hirata"/>
```

Resultado:

Nome: Jonas Hirata

3. XHTML:

```
<h:outputLabel value="Nome: " for="nome" />
<h:inputText value="#{usuarioBean.nome}" readonly="true" id="nome" />
```

HTML gerado:

```
<label for="j_idt6:nome">Nome: </label>
<input id="j_idt6:nome" type="text" name="j_idt6:nome" value="Jonas K. Hirata"
readonly="readonly" />
```

Resultado:

Nome: Jonas Hirata

4. XHTML:

```
<h:outputLabel value="CEP: " for="cep" />
<h:inputText value="#{usuarioBean.cep}" maxlength="9" id="cep" />
```

HTML gerado:

```
<label for="j_idt6:cep">Cep: </label>
<input id="j_idt6:cep" type="text" name="j_idt6:cep" value="01452-001" maxlength="9" />
```

Resultado:

CEP: 01452-001

5. XHTML:

```
<h:outputLabel value="Texto: " for="texto" />
<h:inputTextare value="Um texto de várias linhas" id="texto" />
```

HTML gerado:

```
<label for="j_idt6:texto">Texto: </label>
<textarea id="j_idt6:texto" name="j_idt6:texto">Um texto de várias linhas</textarea>
```

Resultado:

Texto: Um texto de várias linhas

6. XHTML:

```
<h:outputLabel value="Texto: " for="texto" />
<h:inputTextarea value="Um texto de várias linhas" cols="10" rows="3" id="texto"/>
```

HTML gerado:

```
<label for="j_idt6:texto">Texto: </label>
<textarea id="j_idt6:texto" name="j_idt6:texto" cols="10" rows="3">
  Um texto de várias linhas
</textarea>
```

Resultado:

Texto: Um texto de várias linhas

7. XHTML:

```
<h:outputLabel value="Senha: " for="senha" />
<h:inputSecret value="#{usuarioBean.senha}" id="senha" />
```

HTML gerado:

```
<label for="j_idt6:senha">Senha: </label>
<input id="j_idt6:senha" type="password" name="j_idt6:senha" value="" />
```

Resultado:

Senha: ●●●●●

Campos Ocultos

Podemos adicionar informações que são submetidas automaticamente quando um formulário é enviado ao servidor. Essas informações podem ser inseridas com o componente `<h:inputHidden>`. Essa tag possui dois atributos principais: `value` e `id`. Podemos vincular um campo oculto a uma propriedade de um managed bean, como no exemplo abaixo.

```
1  <h:inputHidden id="curso-id" value="#{cursosBean.curso.id}" />
```

Código XHTML 6.11: Exemplo de um campo oculto vinculado a uma propriedade de um managed bean

Caixas de Seleção

O JSF define sete tipos de caixas de seleção.

- <h:selectBooleanCheckbox>
Permite que o usuário faça seleções do tipo “sim ou não”.
- <h:selectManyCheckbox>
Permite que o usuário faça múltiplas seleções do tipo “sim ou não”.
- <h:selectOneRadio>, <h:selectOneMenu>, <h:selectOneListbox>
Permitem a seleção de uma única opção de um conjunto de opções. A principal diferença entre essas tags é modo como elas são apresentadas visualmente.
- <h:selectManyListbox>, <h:selectManyMenu>
Possibilita a seleção de diversas opções de um conjunto de opções. A diferença principal entre essas duas tags é modo como elas são apresentadas visualmente.

Podemos vincular uma caixa de seleção a uma propriedade de um managed bean através do atributo value. No exemplo abaixo, uma caixa de seleção é associada à propriedade aceito do managed bean ContratoBean.

```
1 <h:selectBooleanCheckbox value="#{contratoBean.aceito}" />
```

Código XHTML 6.12: Caixa de seleção vinculada à propriedade aceito do managed bean ContratoBean

Caixas de seleção do tipo <h:selectBooleanCheckbox> devem ser vinculadas a propriedades booleanas.

Opções Estáticas e Dinâmicas

As opções de uma caixa de seleção (com exceção da <h:selectBooleanCheckbox>) podem ser definidas estaticamente ou dinamicamente. Estaticamente, cada opção pode ser adicionada através da tag <f:selectItem>. Considere o seguinte exemplo.

```
1 <h:selectOneMenu value="#{cursosBean.siglaDoCursoEscolhido}">
2   <f:selectItem itemValue="K11" itemLabel="Orientação a Objetos em Java">
3   <f:selectItem itemValue="K12" itemLabel="Desenvolvimento Web com JSF2 e JPA2">
4 </h:selectOneMenu>
```

Código XHTML 6.13: Caixa de seleção com opções definidas estaticamente

O atributo itemValue define o que será enviado para a aplicação se a opção correspondente for selecionada. O atributo itemLabel define a descrição associada à opção correspondente. Essa descrição é exibida para o usuário.

Definir as opções estaticamente pode dificultar a adição ou remoção de opções. Por exemplo, o cadastramento ou a remoção de um curso no sistema implicaria em uma mudança no arquivo XHTML.

Seria interessante eliminar essa dependência fazendo com que o arquivo XHTML recupere dinamicamente todos os cursos cadastrados. Isso pode ser realizado através da tag <f:selectItems>.

```

1 <h:selectOneMenu value="#{cursosBean.siglaDoCursoEscolhido}">
2   <f:selectItems
3     value="#{cursosBean.cursos}"
4     var="curso"
5     itemValue="#{curso.sigla}"
6     itemLabel="#{curso.nome}">
7 </h:selectOneMenu>

```

Código XHTML 6.14: Caixa de seleção com opções definidas dinamicamente

A atributo `value` da tag `<f:selectItems>` deve ser associado a uma coleção de itens. Todos os itens serão percorridos e adicionados como opção na caixa de seleção. O atributo `var` é utilizado para definir a variável que armazenará, a cada rodada, o item atual. Essa variável permite que o `itemValue` e o `itemLabel` de cada opção sejam definidos.

Seleção Única ou Múltipla

Algumas caixas de seleção permitem que apenas uma opção seja selecionada, enquanto outras permitem que múltiplas opções sejam selecionadas. Considere um formulário que permita ao usuário escolher apenas um dos cursos de uma lista. Como vimos, esse formulário pode ter a seguinte estrutura:

```

1 <h:selectOneListbox value="#{cursosBean.siglaDoCursoEscolhido}">
2   <f:selectItems
3     value="#{cursosBean.cursos}"
4     var="curso"
5     itemValue="#{curso.sigla}"
6     itemLabel="#{curso.nome}" />
7 </h:selectOneListbox>

```

Código XHTML 6.15: Seleção de um único curso

```

1 @ManagedBean
2 public class CursosBean {
3   private String siglaDoCursoEscolhido;
4   private List<Curso> cursos;
5
6   // GETTERS E SETTERS
7 }

```

Código Java 6.2: CursosBean.java

```

1 public class Curso {
2   private String sigla;
3   private String nome;
4
5   // GETTERS E SETTERS
6 }

```

Código Java 6.3: Curso.java

A propriedade associada à caixa de seleção deve ser compatível com o tipo de dado utilizado no atributo `itemValue`. No exemplo acima, as siglas dos cursos são do tipo `String`. Consequentemente, a propriedade que armazenará a sigla do curso escolhido também deve ser do tipo `String`.

Agora, suponha que o usuário possa escolher mais do que um curso. Nesse caso, o managed bean deve estar preparado para guardar uma lista de siglas de cursos ao invés de uma única sigla. Para isso,

devemos adicionar uma propriedade no managed bean CursosBean, como mostra o código abaixo.

```

1  @ManagedBean
2  public class CursosBean {
3      private List<String> siglasDosCursosEscolhidos;
4      private List<Curso> cursos;
5
6      // GETTERS E SETTERS
7 }
```

Código Java 6.4: CursosBean.java

Para permitir a seleção múltipla, devemos utilizar uma caixa de seleção do tipo `<h:selectManyListbox>` ou `<h:selectManyMenu>`. Essa caixa deve então ser vinculada à nova propriedade do managed bean CursosBean. Por exemplo, se optarmos pela `<h:selectManyListbox>`, poderíamos ter o seguinte código.

```

1 <h:selectManyListbox value="#{cursosBean.siglasDosCursosEscolhidos}">
2     <f:selectItems
3         value="#{cursosBean.cursos}"
4         var="curso"
5         itemValue="#{curso.sigla}"
6         itemLabel="#{curso.nome}" />
7 </h:selectManyListbox>
```

Código XHTML 6.16: Seleção de diversos cursos

Caixa de Seleção com Pseudo-opção

Por padrão, a caixa de seleção do tipo `<h:selectOneMenu>` sempre tem uma opção selecionada. Contudo, em alguns casos, faz sentido para o usuário não selecionar nenhuma das opções disponíveis. Para resolver esse problema, podemos adicionar uma pseudo-opção na caixa de seleção. Considere o código abaixo.

```

1 <h:selectOneMenu value="#{cursosBean.siglaDoCursoEscolhido}">
2     <f:selectItem itemLabel="Nenhum" noSelectionOption="true"/>
3     <f:selectItems
4         value="#{cursosBean.cursos}"
5         var="curso"
6         itemValue="#{curso.sigla}"
7         itemLabel="#{curso.nome}" />
8 </h:selectOneMenu>
```

Código XHTML 6.17: Pseudo-opção

Para adicionar uma pseudo-opção, basta utilizar a tag `<f:selectItem>` com o atributo `noSelectionOption` igual a “true”. Se essa pseudo-opção estiver selecionada no momento em que o formulário for submetido, a propriedade correspondente receberá o valor null.

Muitas vezes, a pseudo-opção é utilizada simplesmente para exibir uma mensagem ao usuário, como “Escolha uma opção” ou “Selecione um item”.

Exemplos

A seguir, mostraremos exemplos de cada tipo de caixa de seleção.

1. XHTML:

```

1 <h:outputLabel value="Ex-aluno" for="exaluno"/>
2 <h:selectBooleanCheckbox value="#{cursosBean.exAluno}" id="exaluno"/>
```

HTML gerado:

```

1 <label for="j_idt6:exaluno">Ex-aluno</label>
2 <input id="j_idt6:exaluno" type="checkbox" name="j_idt6:exaluno" />
```

Resultado:

Ex-aluno

2. XHTML:

```

1 <h:selectManyCheckbox value="#{cursosBean.siglasDosCursosEscolhidos}"
2   layout="pageDirection">
3     <f:selectItems
4       value="#{cursosBean.cursos}"
5       var="curso"
6       itemValue="#{curso.sigla}"
7       itemLabel="#{curso.nome}" />
8   </h:selectManyCheckbox>
```

HTML gerado:

```

1 <table>
2   <tr>
3     <td>
4       <input name="j_idt6:j_idt27" id="j_idt6:j_idt27:0" value="K11" type="checkbox"/>
5       <label for="j_idt6:j_idt27:0" class="">Orientação a Objetos em Java</label>
6     </td>
7   </tr>
8   <tr>
9     <td>
10      <input name="j_idt6:j_idt27" id="j_idt6:j_idt27:1" value="K12" type="checkbox"/>
11      <label for="j_idt6:j_idt27:1" class="">
12        Desenvolvimento Web com JSF2 e JPA2
13      </label>
14    </td>
15  </tr>
16  <tr>
17    <td>
18      <input name="j_idt6:j_idt27" id="j_idt6:j_idt27:2" value="K51" type="checkbox"/>
19      <label for="j_idt6:j_idt27:2" class="">Design Patterns em Java</label>
20    </td>
21  </tr>
22 </table>
```

Resultado:

- Orientação a Objetos em Java
- Desenvolvimento Web com JSF2 e JPA2
- Design Patterns em Java

3. XHTML:

```

1 <h:selectOneRadio value="#{cursosBean.siglasDosCursosEscolhidos}" layout="lineDirection">
2   <f:selectItems
3     value="#{cursosBean.cursos}"
```

```

4     var="curso"
5     itemValue="#{curso.sigla}"
6     itemLabel="#{curso.nome}" />
7 </h:selectOneRadio>
```

HTML gerado:

```

1 <table>
2   <tr>
3     <td>
4       <input type="radio" name="j_idt6:j_idt30" id="j_idt6:j_idt30:0" value="K11"/>
5       <label for="j_idt6:j_idt30:0">Orientação a Objetos em Java</label>
6     </td>
7     <td>
8       <input type="radio" name="j_idt6:j_idt30" id="j_idt6:j_idt30:1" value="K12"/>
9       <label for="j_idt6:j_idt30:1">Desenvolvimento Web com JSF2 e JPA2</label>
10    </td>
11   <td>
12     <input type="radio" name="j_idt6:j_idt30" id="j_idt6:j_idt30:2" value="K51"/>
13     <label for="j_idt6:j_idt30:2">Design Patterns em Java</label>
14   </td>
15 </tr>
16 </table>
```

Resultado:

- Orientação a Objetos em Java
- Desenvolvimento Web com JSF2 e JPA2
- Design Patterns em Java

4. XHTML:

```

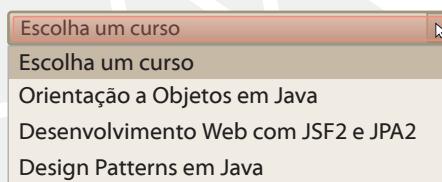
1 <h:selectOneMenu value="#{cursosBean.siglasDosCursosEscolhidos}">
2   <f:selectItem itemLabel="Escolha um curso" noSelectionOption="true"/>
3   <f:selectItems
4     value="#{cursosBean.cursos}"
5     var="curso"
6     itemValue="#{curso.sigla}"
7     itemLabel="#{curso.nome}" />
8 </h:selectOneMenu>
```

HTML gerado:

```

1 <select name="j_idt6:j_idt33" size="1">
2   <option value="">Escolha um curso</option>
3   <option value="K11">Orientação a Objetos em Java</option>
4   <option value="K12">Desenvolvimento Web com JSF2 e JPA2</option>
5   <option value="K51">Design Patterns em Java</option>
6 </select>
```

Resultado:



5. XHTML:

```

1 <h:selectOneListbox value="#{cursosBean.siglasDosCursosEscolhidos}">
2   <f:selectItems
3     value="#{cursosBean.cursos}"
```

```

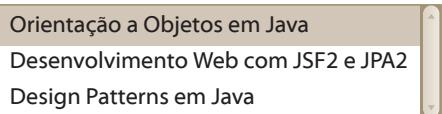
4     var="curso"
5     itemValue="#{curso.sigla}"
6     itemLabel="#{curso.nome}" />
7 </h:selectOneListbox>
```

HTML gerado:

```

1 <select name="j_idt6:j_idt37" size="3">
2   <option value="K11">Orientação a Objetos em Java</option>
3   <option value="K12">Desenvolvimento Web com JSF2 e JPA2</option>
4   <option value="K51">Design Patterns em Java</option>
5 </select>
```

Resultado:



6. XHTML:

```

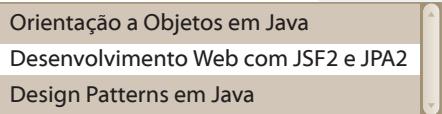
1 <h:selectManyListbox value="#{cursosBean.siglasDosCursosEscolhidos}">
2   <f:selectItems
3     value="#{cursosBean.cursos}"
4     var="curso"
5     itemValue="#{curso.sigla}"
6     itemLabel="#{curso.nome}" />
7 </h:selectManyListbox>
```

HTML gerado:

```

1 <select name="j_idt6:j_idt40" multiple="multiple" size="3">
2   <option value="K11">Orientação a Objetos em Java</option>
3   <option value="K12">Desenvolvimento Web com JSF2 e JPA2</option>
4   <option value="K51">Design Patterns em Java</option>
5 </select>
```

Resultado:



7. XHTML:

```

1 <h:selectManyMenu value="#{cursosBean.siglasDosCursosEscolhidos}" style="height: 70px;">
2   <f:selectItem itemLabel="Escolha um curso" noSelectionOption="true" />
3   <f:selectItems
4     value="#{cursosBean.cursos}"
5     var="curso"
6     itemValue="#{curso.sigla}"
7     itemLabel="#{curso.nome}" />
8 </h:selectManyMenu>
```

HTML gerado:

```

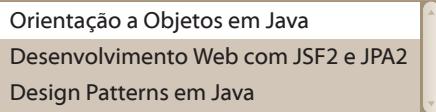
1 <select name="j_idt6:j_idt43" multiple="multiple" size="1" style="height: 70px;">
2   <option value="" selected="selected">Escolha um curso</option>
3   <option value="K11">Orientação a Objetos em Java</option>
```

```

4 <option value="K12">Desenvolvimento Web com JSF2 e JPA2</option>
5 <option value="K51">Design Patterns em Java</option>
6 </select>

```

Resultado:



Botões e Links

O JSF define cinco tipos de botões e links:

- <h:commandButton> e <h:commandLink>
Enviam os dados de um formulário HTML para o servidor através do método POST do HTTP.
- <h:button> e <h:link>
Realizam requisições HTTP do tipo GET. As URLs das requisições são geradas pelo JSF a partir do atributo outcome.
- <h:outputLink>
Também realiza requisições HTTP do tipo GET, mas exige que a URL de destino seja explicitamente especificada.

Os componentes <h:commandButton> e <h:commandLink> são usados para submeter formulários HTML por meio do método POST do HTTP. Esses dois componentes podem ser associados a métodos de ação de um managed bean através dos atributos action e actionListener. Esses atributos serão detalhados em outro capítulo.

```

1 <h:commandButton value="Adiciona curso" action="#{cursosBean.adicionaCurso}" />
2 <h:commandLink value="Remove curso" action="#{cursosBean.removeCurso}" />

```

Código XHTML 6.25: Exemplos de utilização do <h:commandButton> e do <h:commandLink>

A diferença básica entre o <h:commandButton> e o <h:commandLink> é o modo como eles são apresentados visualmente.

Os componentes <h:button> e <h:link>, por sua vez, são usados para realizar requisições através do método GET. O JSF utiliza o atributo outcome de um <h:button> ou <h:link> para definir a página de destino. Esses componentes não são utilizados para submeter formulários HTML.

```

1 <h:button value="Lista de cursos" outcome="lista-cursos" />
2 <h:link value="Home" outcome="home" />

```

Código XHTML 6.26: Exemplos de utilização do <h:button> e do <h:link>

Novamente, a diferença fundamental entre o <h:button> e o <h:link> é o modo como eles são exibidos visualmente.

O componente <h:outputLink> permite apenas a criação de links HTML que realizam requisições do tipo GET. Diferentemente dos componentes <h:button> e <h:link>, a URL da requisição é definida explicitamente no atributo value.

```
1 <h:outputLink value="http://www.k19.com.br"/>
```

Código XHTML 6.27: Um link para a página da K19

Exemplos

1. XHTML:

```
1 <h:commandButton value="Adiciona curso" action="#{cursosBean.adicionaCurso}" />
```

HTML gerado:

```
1 <input type="submit" name="j_idt59:j_idt60" value="Adiciona curso" />
```

Resultado:

Adiciona curso

2. XHTML:

```
1 <h:commandLink value="Remove curso" action="#{cursosBean.removeCurso}" />
```

HTML gerado:

```
1 <a href="#" onclick="mojarra.jsfcljs(document.getElementById('j_idt62'),{'j_idt62:j_idt66':'→
2   j_idt62:j_idt66'},');return false">
3   Remove curso
4 </a>
```

Resultado:

[Remove curso](#)

3. XHTML:

```
1 <h:button value="Lista de cursos" outcome="lista-cursos" />
```

HTML gerado:

```
1 <input
2   type="button"
3   onclick="window.location.href='/K19-Componentes-Visuais/lista-cursos.xhtml'; return false;←
4   value="Lista de cursos" />
```

Resultado:

Lista de cursos

4. XHTML:

```
1 <h:link value="Home" outcome="home" />
```

HTML gerado:

```
1 <a href="/K19-Componentes-Visuais/home.xhtml">Home</a>
```

Resultado:

[Home](#)

5. XHTML:

```
1 <h:outputLink value="http://www.k19.com.br">K19</h:outputLink>
```

HTML gerado:

```
1 <a href="http://www.k19.com.br">K19</a>
```

Resultado:

[K19](#)



Exercícios de Fixação

- 1 No Eclipse, crie um Dynamic Web Project chamado **K19-Componentes-Visuais** seguindo os passos vistos no exercício do Capítulo 5.
- 2 Crie uma classe chamada **UsuarioBean**. Essa classe deve possuir dois atributos: um atributo do tipo **String** para armazenar o nome do usuário e um atributo do tipo **int** para armazenar a idade do usuário.

```
1 @ManagedBean
2 public class UsuarioBean {
3     private String nome;
4     private int idade;
5
6     // GETTERS E SETTERS
7 }
```

Código Java 6.5: *UsuarioBean.java*

- 3 Crie um arquivo XHTML no diretório WebContent chamado **usuario.xhtml**. Nesse arquivo, crie um formulário para o usuário digitar e enviar o seu nome e a sua idade.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
```

```

11 <h:form>
12   <h:outputLabel value="Nome: " for="campo-nome" />
13   <h:inputText value="#{usuarioBean.nome}" id="campo-nome" />
14
15   <h:outputLabel value="Idade: " for="campo-idade" />
16   <h:inputText value="#{usuarioBean.idade}" id="campo-idade" />
17
18   <h:commandButton value="Enviar" />
19 </h:form>
20 </h:body>
21 </html>

```

Código XHTML 6.33: *usuario.xhtml*

- 4 Adicione a aplicação ao servidor e initialize-o. Acesse a aplicação em <http://localhost:8080/K19-Componentes-Visuais/usuario.xhtml>.
- 5 No arquivo *usuario.xhtml*, acrescente um trecho de código para exibir os dados do usuário.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h:form>
12     <h:outputLabel value="Nome: " for="campo-nome" />
13     <h:inputText value="#{usuarioBean.nome}" id="campo-nome" />
14
15     <h:outputLabel value="Idade: " for="campo-idade" />
16     <h:inputText value="#{usuarioBean.idade}" id="campo-idade" />
17
18     <h:commandButton value="Enviar" />
19 </h:form>
20
21   O nome do usuário é <b>#{usuarioBean.nome}</b>
22   e sua idade é <b>#{usuarioBean.idade}</b>.
23 </h:body>
24 </html>

```

Código XHTML 6.34: *usuario.xhtml*

- 6 Agora, crie um conversor monetário. Esse conversor deve receber três dados de entrada: as moedas de origem e destino, e o valor a ser convertido. Primeiramente, crie uma classe chamada *ConversorMonetarioBean* para implementar esse conversor.

```

1 @ManagedBean
2 public class ConversorMonetarioBean {
3   private String de;
4   private String para;
5   private Double valor;
6
7   // GETTERS E SETTERS
8 }

```

Código Java 6.6: *ConversorMonetarioBean.java*

- 7 Adicione um método para realizar a conversão e um atributo para armazenar o resultado.

```

1  @ManagedBean
2  public class ConversorMonetarioBean {
3      private String de;
4      private String para;
5      private Double valor;
6
7      private Double resultado;
8
9      private Map<String, Double> taxas = new LinkedHashMap<String, Double>();
10
11     public ConversorMonetarioBean() {
12         this.taxas.put("Real", 1.0);
13         this.taxas.put("Euro", 2.33);
14         this.taxas.put("Peso argentino", 0.42);
15         this.taxas.put("Dolar americano", 1.84);
16     }
17
18     public void converte() {
19         this.resultado = this.valor * this.taxas.get(this.de) / this.taxas.get(this.para);
20     }
21
22     // GETTERS E SETTERS
23 }
```

Código Java 6.7: ConversorMonetarioBean.java

- 8 No diretório WebContent, crie um arquivo chamado conversor-monetario.xhtml. Nesse arquivo, implemente um formulário para o usuário digitar os dados de entrada e ver o resultado da conversão.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5  xmlns:h="http://java.sun.com/jsf/html"
6  xmlns:f="http://java.sun.com/jsf/core">
7
8  <h:head>
9      <title>K19 Treinamentos</title>
10
11 <h:body>
12     <h:form>
13         <h:commandButton value="Converter" action="#{conversorMonetarioBean.converte}" />
14
15         <h:inputText value="#{conversorMonetarioBean.valor}" />
16
17         <h:outputLabel value="de " for="de" />
18
19         <h:selectOneMenu value="#{conversorMonetarioBean.de}" id="de">
20             <f:selectItems
21                 value="#{conversorMonetarioBean.taxas.keySet()}"
22                 var="moeda"
23                 itemValue="#{moeda}"
24                 itemLabel="#{moeda}" />
25         </h:selectOneMenu>
26
27         <h:outputLabel value="para " for="para" />
28
29         <h:selectOneMenu value="#{conversorMonetarioBean.para}" id="para">
30             <f:selectItems
31                 value="#{conversorMonetarioBean.taxas.keySet()}"
32                 var="moeda"
33                 itemValue="#{moeda}"
34                 itemLabel="#{moeda}" />
35         </h:selectOneMenu>
36     </h:form>
37
38     Resultado: #{conversorMonetarioBean.resultado}
```

```

39  </h:body>
40  </html>
41

```

Código XHTML 6.35: conversor-monetario.xhtml

- 9 Acesse a aplicação em <http://localhost:8080/K19-Componentes-Visuais/conversor-monetario.xhtml>



Exercícios Complementares

- 1 Considere o serviço de atendimento ao consumidor de uma loja virtual. Você deve criar um formulário para que o consumidor possa se comunicar com a loja. Esse formulário deve ter campos para o usuário informar o seu nome, e-mail, telefone, sexo e número do pedido. Além disso, deve existir um campo para o usuário selecionar um assunto e outro para ele escrever o seu comentário.

Textos e Imagens

Para inserir textos em uma página, podemos usar os componentes `<h:outputText>` e `<h:outputFormat>`. O texto exibido por tais componentes é definido pelo atributo `value`. Em particular, o `<h:outputFormat>` permite diversos tipos de formatação do texto que será exibido.

```

1 <h:outputFormat value="Preço do produto {0}: R$ {1}">
2   <f:param value="#{lojaBean.produto.nome}" />
3   <f:param value="#{lojaBean.produto.preco}" />
4 </h:outputFormat>

```

Código XHTML 6.37: Exemplo de utilização do `<h:outputFormat>`

O componente `<h:graphicImage>` é usado para inserir imagens. Podemos indicar o caminho da imagem através dos atributos `value` ou `url`. Esses dois atributos são exatamente iguais.

```

1 <h:graphicImage value="/imagens/k19-treinamentos.png" />

```

Código XHTML 6.38: Exemplo de inserção de imagem

Para padronizar a organização dos recursos da sua aplicação, o JSF 2 permite a criação de bibliotecas de imagens. Para criar uma biblioteca, basta adicionar um diretório na pasta `/resources` (na raiz da aplicação). Por exemplo, se criarmos o diretório `/resources/imagens-k19/` e adicionarmos a imagem `k19-treinamentos.png` nesse diretório, podemos inseri-la em uma página da seguinte forma:

```

1 <h:graphicImage library="imagens-k19" name="k19-treinamentos.png" />

```

Código XHTML 6.39: Exemplo de inserção de imagem usando o atributo `library`



Mais Sobre

Todos os componentes do JSF possuem um atributo booleano chamado `rendered`. Esse atributo indica se o componente deve ser renderizado durante a etapa *Render*.

Response do processamento de uma requisição. O valor padrão desse atributo é true, o que indica que o componente deve ser exibido.

O preço de alguns produtos vendidos pela Amazon.com, por exemplo, não são exibidos na página de apresentação do produto. Para um produto desse tipo, o usuário precisa adicioná-lo ao carrinho de compras para então poder ver o seu preço.

Como exemplo, suponha que um objeto do tipo Produto possua uma propriedade booleana chamada mostraPreco. Ela indica se o preço daquele produto deve ser exibido ou não. Assim, podemos associar o atributo rendered do componente responsável pela exibição do preço do produto a essa propriedade. Isso pode ser feito usando *expression language*, como no exemplo abaixo.

```
1 <h:outputFormat value="Preço do produto {0}: R$ {1}"
2   rendered="#{lojaBean.produto.mostraPreco}">
3   <f:param value="#{lojaBean.produto.nome}" />
4   <f:param value="#{lojaBean.produto.preco}" />
5 </h:outputFormat>
```

Código XHTML 6.40: O preço do produto é exibido somente se o valor da propriedade mostraPreco for true

Exemplos

1. XHTML:

```
1 <h:outputText value="Curso: #{curso.sigla} - #{curso.descricao}" />
```

HTML gerado:

```
1 Curso: K12 - Desenvolvimento Web com JSF2 e JPA2
```

2. XHTML:

```
1 <h:outputFormat value="{0} amava {1} que amava {2} que amava {3}
2   que amava {4} que amava {5} que não amava ninguém. {0} foi
3   para os Estados Unidos, {1} para o convento, {2} morreu de
4   desastre, {3} ficou para tia, {4} suicidou-se e {5} casou
5   com J. Pinto Fernandes que não tinha entrado na história.">
6
7   <f:param value="João"/>
8   <f:param value="Teresa"/>
9   <f:param value="Raimundo"/>
10  <f:param value="Maria"/>
11  <f:param value="Joaquim"/>
12  <f:param value="Lili"/>
13 </h:outputFormat>
```

HTML gerado:

```
1 João amava Teresa que amava Raimundo que amava Maria que amava Joaquim que amava
2 Lili que não amava ninguém. João foi para os Estados Unidos, Teresa para o
3 convento, Raimundo morreu de desastre, Maria ficou para tia, Joaquim suicidou-se
4 e Lili casou com J. Pinto Fernandes que não tinha entrado na história.
```

3. XHTML:

```
1 <h:graphicImage value="/imagens/k19-treinamentos.png" />
```

HTML gerado:

```
1 
```

Resultado:



Exercícios de Fixação

- 10 No arquivo conversor-monetario.xhtml do projeto K19-Componentes-Visuais, use a tag `<h:outputFormat>` para exibir o resultado do conversor monetário.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <h:form>
13     <h:commandButton value="Converter" action="#{conversorMonetarioBean.converte}" />
14
15     <h:inputText value="#{conversorMonetarioBean.valor}" />
16
17     <h:outputLabel value="de " for="de"/>
18
19     <h:selectOneMenu value="#{conversorMonetarioBean.de}" id="de">
20       <f:selectItems
21         value="#{conversorMonetarioBean.taxas.keySet()}"
22         var="moeda"
23         itemValue="#{moeda}"
24         itemLabel="#{moeda}" />
25   </h:selectOneMenu>
26
27     <h:outputLabel value="para " for="para"/>
28
29     <h:selectOneMenu value="#{conversorMonetarioBean.para}" id="para">
30       <f:selectItems
31         value="#{conversorMonetarioBean.taxas.keySet()}"
32         var="moeda"
33         itemValue="#{moeda}"
34         itemLabel="#{moeda}" />
35   </h:selectOneMenu>
36 </h:form>
37
38   <h:outputFormat value="{0} em {1} equivale a {2} em {3}" 
39   rendered="#{conversorMonetarioBean.resultado != null}">
40     <f:param value="#{conversorMonetarioBean.valor}" />
41     <f:param value="#{conversorMonetarioBean.de}" />
42     <f:param value="#{conversorMonetarioBean.resultado}" />
43     <f:param value="#{conversorMonetarioBean.para}" />

```

```

44  </h:outputFormat>
45  </h:body>
46  </html>

```

Código XHTML 6.44: conversor-monetario.xhtml

Componentes de Organização

O JSF define dois componentes que nos ajudam a organizar visualmente os elementos de uma página JSF

- `<h:panelGrid>`
Organiza os elementos em uma grade.
- `<h:panelGroup>`
Permite que diversos componentes sejam tratados como um único componente.

O `<h:panelGrid>` é usado para organizar componentes em forma de uma grade. O uso de um `<h:panelGrid>` é bastante simples. Basicamente, devemos configurar a quantidade de colunas que desejamos. Para isso, utilizamos o atributo `columns`.

```

1  <h:panelGrid columns="2">
2    <h:outputLabel value="Nome do curso: " for="nome"/>
3    <h:inputText value="#{cursosBean.curso.nome}" id="nome"/>
4
5    <h:outputLabel value="Descrição: " for="descricao"/>
6    <h:inputTextareal value="#{cursosBean.curso.descricao}" id="descricao"/>
7
8    <h:outputLabel value="Carga horária: " for="carga-horaria"/>
9    <h:inputText value="#{cursosBean.curso.cargaHoraria}" id="carga-horaria"/>
10
11   <h:commandButton action="#{cursosBean.cadastraCurso}" value="Cadastrar"/>
12 </h:panelGrid>

```

Código XHTML 6.45: Organização de um formulário com o <h:panelGrid>

Os componentes são inseridos na grade de acordo com a ordem em que estão apresentados no código. O primeiro elemento é inserido na primeira coluna da primeira linha. O segundo elemento é inserido na segunda coluna da primeira linha. Uma vez que uma linha foi completamente preenchida, o próximo componente é inserido na primeira coluna da próxima linha, e o processo se repete até o último componente.

Suponha que seja necessário inserir mais de um componente em uma mesma célula de um `<h:panelGrid>`. No exemplo acima, podemos adicionar um ícone de um relógio ao lado do rótulo “Carga horária:”. No entanto, queremos que a imagem e o rótulo sejam colocados na mesma célula. Para isso, podemos agrupá-los usando um `<h:panelGroup>`.

```

1  <h:panelGroup>
2    <h:outputLabel value="Carga horária: " for="carga-horaria"/>
3    <h:graphicImage library="imagens" name="relogio.png" />
4  </h:panelGroup>

```

Código XHTML 6.46: Agrupando componentes com o <h:panelGroup>

Exemplos

1. XHTML:

```

1 <h:panelGrid columns="2">
2   <h:outputLabel value="Nome do curso: " for="nome"/>
3   <h:inputText value="#{cursosBean.curso.nome}" id="nome"/>
4
5   <h:outputLabel value="Descrição: " for="descricao"/>
6   <h:inputTextarea value="#{cursosBean.curso.descricao}" id="descricao"/>
7
8   <h:panelGroup>
9     <h:outputLabel value="Carga horária: " for="carga-horaria"/>
10    <h:graphicImage library="imagens" name="relogio.png" />
11  </h:panelGroup>
12  <h:inputText value="#{cursosBean.curso.cargaHoraria}" id="carga-horaria"/>
13
14  <h:commandButton action="#{cursosBean.cadastraCurso}" value="Cadastrar"/>
15 </h:panelGrid>

```

HTML gerado:

```

1 <table>
2   <tbody>
3     <tr>
4       <td><label for="nome"> Nome do curso: </label></td>
5       <td><input id="nome" type="text" name="nome" /></td>
6     </tr>
7     <tr>
8       <td><label for="descricao"> Descrição: </label></td>
9       <td><textarea id="descricao" name="descricao"></textarea></td>
10    </tr>
11    <tr>
12      <td>
13        <label for="carga-horaria"> Carga horária: </label>
14        
15      </td>
16      <td><input id="carga-horaria" type="text" name="carga-horaria" /></td>
17    </tr>
18    <tr>
19      <td><input type="submit" name="j_idt98" value="Cadastrar" /></td>
20    </tr>
21  </tbody>
22 </table>

```

Resultado:

Nome do curso:

Descrição:

Carga horária: (1)

Cadastrar

Tabelas

O JSF fornece o componente `<h:dataTable>` para a criação de tabelas. Podemos associar uma lista de elementos a um `<h:dataTable>` através do atributo `value`. Automaticamente, esse componente gera uma linha para cada item da lista. Os itens da lista podem ser acessados através de uma variável definida pelo atributo `var`.

As colunas da tabela são definidas pelo componente `<h:column>`. Podemos acrescentar cabeçalhos e rodapés à tabela e a cada coluna usando o componente `<f:facet>`.

Veja um exemplo a seguir.

```

1 <h:dataTable value="#{cursosBean.cursos}" var="curso">
2
3   <f:facet name="header">Lista de cursos</f:facet>
4
5   <h:column>
6     <f:facet name="header">Sigla</f:facet>
7     #{curso.sigla}
8   </h:column>
9
10  <h:column>
11    <f:facet name="header">Nome</f:facet>
12    #{curso.nome}
13  </h:column>
14
15  <h:column>
16    <f:facet name="header">Descrição</f:facet>
17    #{curso.descricao}
18  </h:column>
19
20  <h:column>
21    <f:facet name="header">Adicionar turma</f:facet>
22    <h:commandLink
23      value="Adicionar turma"
24      action="#{cursosBean.adicionarTurma(curso)}" />
25  </h:column>
26 </h:dataTable>

```

Código XHTML 6.48: Exemplo de criação de tabela com o componente `<h:dataTable>`

Observe, no exemplo acima, que a tabela está associada à propriedade `cursos` do managed bean `cursosBean`. A variável `curso` é utilizada para acessar cada um dos elementos da propriedade `cursos`.

O cabeçalho “Lista de cursos” da tabela está definido com o componente `<f:facet>` assim como o cabeçalho de cada coluna (“Sigla”, “Nome”, “Descrição” e “Adicionar turma”).

O HTML gerado pelo JSF a partir do código acima seria mais ou menos assim:

```

1 <table>
2   <thead>
3     <tr>
4       <th colspan="4" scope="colgroup">Lista de cursos</th>
5     </tr>
6     <tr>
7       <th scope="col">Sigla</th>
8       <th scope="col">Nome</th>
9       <th scope="col">Descrição</th>
10      <th scope="col">Adicionar turma</th>
11    </tr>
12  </thead>
13  <tbody>
14    <tr>
15      <td>K11</td>

```

```

16    <td>Orientação a Objetos em Java</td>
17    <td>
18        Com este curso você vai obter uma base sólida de
19            conhecimentos de Java e de Orientação a Objetos
20        </td>
21        <td>
22            <input
23                type="submit" name="j_idt100:j_idt101:0:j_idt114"
24                value="Adicionar turma" />
25        </td>
26    </tr>
27    <tr>
28        <td>K12</td>
29        <td>Desenvolvimento Web com JSF2 e JPA2</td>
30        <td>
31            Depois deste curso, você estará apto a desenvolver
32            aplicações Web com os padrões da plataforma Java
33        </td>
34        <td>
35            <input
36                type="submit" name="j_idt100:j_idt101:1:j_idt114"
37                value="Adicionar turma" />
38        </td>
39    </tr>
40    <tr>
41        <td>K21</td>
42        <td>Persistência com JPA2 e Hibernate</td>
43        <td>
44            Neste curso de Java Avançado, abordamos de maneira profunda
45            os recursos de persistência do JPA2 e do Hibernate
46        </td>
47        <td>
48            <input
49                type="submit" name="j_idt100:j_idt101:2:j_idt114"
50                value="Adicionar turma" />
51        </td>
52    </tr>
53 </tbody>
54 </table>

```

Código HTML 6.24: Código HTML gerado pelo JSF

A figura abaixo mostra como essa tabela é apresentada no navegador.

Lista de cursos

Sigla	Nome	Descrição	Adicionar Turma
K11	Orientação a Objetos em Java	Com este curso você vai obter uma base sólida de conhecimentos de Java e de Orientação a Objetos.	Adiciona turma
K12	Desenvolvimento Web com JSF2 e JPA2	Depois deste curso, você estará apto a desenvolver aplicações Web com os padrões da plataforma Java	Adiciona turma
K21	Persistência com JPA2 e Hibernate	Neste curso de Java Avançado, abordamos de maneira profunda os recursos de persistência do JPA2 e do Hibernate	Adiciona turma

Figura 6.1: Representação do HTML no navegador



Exercícios de Fixação

- 11** No projeto K19-Componentes-Visuais, crie uma página que contenha um formulário para adicionar cursos e que exiba os cursos já adicionados. Primeiramente, crie uma classe chamada `Curso` para representar um curso. Essa classe deve ter dois atributos do tipo `String`: um para armazenar o nome e outro para armazenar a sigla do curso.

```

1 public class Curso {
2     private String nome;
3     private String sigla;
4
5     // GETTERS E SETTERS
6 }
```

Código Java 6.9: `Curso.java`

- 12** Crie uma classe chamada `CursosBean` para armazenar uma lista de cursos. Para que os cursos adicionados sejam mantidos nessa variável entre uma requisição e outra, marque a classe com a anotação `@SessionScoped`. No Capítulo 9 discutiremos sobre escopos de managed beans.

```

1 @ManagedBean
2 @SessionScoped
3 public class CursosBean {
4     private List<Curso> cursos = new ArrayList<Curso>();
5
6     // GETTER E SETTER
7 }
```

Código Java 6.10: `CursosBean.java`

- 13** Na classe `CursosBean`, adicione dois atributos do tipo `String` para armazenar o nome e a sigla de um curso e crie um método para adicionar um curso com esse nome e essa sigla.

```

1 @ManagedBean
2 @SessionScoped
3 public class CursosBean {
4     private List<Curso> cursos = new ArrayList<Curso>();
5     private Curso curso = new Curso();
6
7     public void adicionaCurso() {
8         this.cursos.add(this.curso);
9         this.curso = new Curso();
10    }
11
12    // GETTERS E SETTERS
13 }
```

Código Java 6.11: `CursosBean.java`

- 14** Crie um arquivo chamado `cursos.xhtml` no diretório `WebContent` e implemente um formulário para adicionar cursos à lista de cursos do managed bean `cursosBean`. Use a tag `<h:panelGrid>` para organizar o formulário.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:h="http://java.sun.com/jsf/html"
```

```

6   xmlns:f="http://java.sun.com/jsf/core">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <h:form>
13     <h:panelGrid columns="2">
14       <h:outputLabel value="Nome: " for="campo-nome" />
15       <h:inputText value="#{cursosBean.curso.nome}" id="campo-nome" />
16
17       <h:outputLabel value="Sigla: " for="campo-sigla" />
18       <h:inputText value="#{cursosBean.curso.sigla}" id="campo-sigla" />
19
20       <h:commandButton value="Adicionar" action="#{cursosBean.adicionaCurso}" />
21     </h:panelGrid>
22   </h:form>
23 </h:body>
24 </html>

```

Código XHTML 6.49: cursos.xhtml

- 15** No arquivo `cursos.xhtml`, insira um trecho de código para exibir os cursos já adicionados. Use a tag `<h:dataTable>` para apresentar os cursos. Você pode usar o atributo `rendered` desse componente para exibir a tabela apenas quando a lista de cursos não está vazia.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <h:form>
13     <h:panelGrid columns="2">
14       <h:outputLabel value="Sigla: " for="campo-sigla" />
15       <h:inputText value="#{cursosBean.curso.sigla}" id="campo-sigla" />
16
17       <h:outputLabel value="Nome: " for="campo-nome" />
18       <h:inputText value="#{cursosBean.curso.nome}" id="campo-nome" />
19
20       <h:commandButton value="Adicionar" action="#{cursosBean.adicionaCurso}" />
21     </h:panelGrid>
22   </h:form>
23
24   <h:dataTable value="#{cursosBean.cursos}" var="curso"
25     rendered="#{not empty cursosBean.cursos}">
26
27     <f:facet name="header">Lista de Cursos</f:facet>
28
29     <h:column>
30       <f:facet name="header">Sigla</f:facet>
31       #{curso.sigla}
32     </h:column>
33
34     <h:column>
35       <f:facet name="header">Nome</f:facet>
36       #{curso.nome}
37     </h:column>
38   </h:dataTable>
39 </h:body>
40 </html>

```

Código XHTML 6.50: cursos.xhtml

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Componentes-Visuais/cursos.xhtml>



Exercícios Complementares

- 2 Usando o componente <h:dataTable>, liste os produtos de uma loja virtual. A tabela deve mostrar o nome e o preço de cada produto. No diretório src, crie uma classe chamada Produto. Essa classe deve ter um atributo do tipo String para armazenar o nome do produto e um atributo do tipo Double para guardar o preço do produto. Crie uma classe chamada ProdutosBean para guardar uma lista de produtos. Implemente a listagem de produtos em um arquivo chamado lista-de-produtos.xhtml no diretório WebContent.
- 3 Na classe Produto, acrescente uma propriedade do tipo Boolean para indicar se o preço do produto deve ser exibido. Modifique a tabela de listagem dos produtos para considerar essa nova restrição. Se o preço de um produto não puder ser exibido, então o texto “Adicione o produto ao carrinho para ver o preço” deve aparecer no lugar do preço do produto.

Mensagens

Durante o processamento de uma requisição, podemos adicionar mensagens que podem ser exibidas na página de resposta. Uma mensagem pode ser adicionada, por exemplo, da seguinte forma:

```
1 FacesMessage mensagem = new FacesMessage("Turma adicionada");
2 FacesContext.getCurrentInstance().addMessage(null, mensagem);
```

Código Java 6.16: Adicionando uma mensagem

O primeiro parâmetro do método addMessage() define qual componente será associado à mensagem adicionada. Quando o valor null é passado nesse parâmetro, a mensagem é considerada global, ou seja, ela não será associada a nenhum componente específico.

Na tela, podemos exibir todas as mensagens geradas no processamento da requisição através do componente <h:messages>. Se desejarmos exibir apenas as mensagens globais, devemos utilizar o atributo globalOnly da seguinte forma:

```
1 <h:messages globalOnly="true" />
```

Código XHTML 6.53: Exibindo as mensagens globais

As mensagens são comumente utilizadas para indicar erros de conversão e validação. Veremos mais detalhes sobre isso no Capítulo 10.

Adicionando JavaScript e CSS

Podemos adicionar recursos como scripts e arquivos de estilos (CSS) usando diretamente as tags <script> e <link> do HTML. Contudo, o JSF define outra maneira de inserir esses recursos nas páginas da aplicação. Tais recursos podem ser inseridos com as tags <h:outputScript> e <h:out-

`putStylesheet`>. A utilização dessas tags facilita a criação de componentes visuais personalizados ou o gerenciamento de recursos que dependem de regionalidade.

```
1 <h:outputScript name="k19.js" library="javascript" target="head" />
2 <h:stylesheet name="k19.css" library="css" />
```

Código XHTML 6.54: Exemplo de adição de código JavaScript e arquivo CSS na página da aplicação

Outros Componentes

<ui:remove>

Para comentar (ou excluir) partes de código XHTML de uma aplicação, o JSF provê a tag `<ui:remove>`. Qualquer trecho de código dentro dessa tag é removido durante o processamento de uma página JSE. No exemplo abaixo, a caixa para inserir o sobrenome do usuário não será processada e, portanto, não será exibida na página gerada.

```
1 <h:outputText value="Nome: #{testeBean.nome}" />
2 <ui:remove>
3   <h:outputText value="Sobrenome: #{testeBean.sobrenome}" />
4 </ui:remove>
```

Código XHTML 6.55: Excluindo parte do código de uma página JSF

O trecho de código HTML correspondente, gerado pelo JSF, é o seguinte:

```
1 Nome: Jonas
```

Código HTML 6.25: Trecho de código HTML gerado pelo JSF

Alternativamente, podemos usar os delimitadores `<!--` e `-->` do XML para comentar o código. Nesse caso, contudo, o trecho de código comentado será processado pelo JSF. Por exemplo, a partir do código

```
1 <h:outputText value="Nome: #{testeBean.nome}" />
2 <!--
3 <h:outputText value="Sobrenome: #{testeBean.sobrenome}" />
4 -->
```

Código XHTML 6.56: Uso de delimitadores do XML para comentar código

o JSF produzirá o seguinte código HTML:

```
1 Nome: Jonas
2 <!--
3 <h:outputText value="Sobrenome: Hirata" />
4 -->
```

Código HTML 6.26: Trecho de código HTML gerado pelo JSF

Note que o código dentro do comentário foi processado pelo JSF. A string "Hirata" foi obtida da propriedade sobrenome do managed bean testeBean.

<ui:repeat>

A tag <ui:repeat> é usada para iterar sobre coleções. Ela possui dois atributos obrigatórios: value e var. O atributo value deve ser associado a uma coleção de objetos e o atributo var deve definir o nome da variável. Essa variável será usada para referenciar cada um dos elementos da coleção. No exemplo abaixo, temos um código para gerar uma lista HTML não ordenada a partir de uma coleção de cursos.

```

1 <h:outputText value="Alguns cursos da K19:" />
2 <ul>
3   <ui:repeat value="#{testeBean.cursos}" var="curso">
4     <li>
5       <h:outputText value="#{curso.sigla}: #{curso.nome}" />
6     </li>
7   </ui:repeat>
8 </ul>

```

Código XHTML 6.57: Iterando sobre uma lista de cursos

A partir desse código, o JSF gera o seguinte código HTML:

```

1 Alguns cursos da K19:
2 <ul>
3   <li>K11: Orientação a Objetos em Java</li>
4   <li>K12: Desenvolvimento Web com JSF2 e JPA2</li>
5   <li>K21: Persistência com JPA2 e Hibernate</li>
6   <li>K51: Design Patterns em Java</li>
7 </ul>

```

Código HTML 6.27: Código HTML gerado pelo JSF



Exercícios de Fixação

- 16 No projeto K19-Componentes-Visuais, crie um arquivo CSS para formatar a página que lista os cursos. No diretório WebContent, crie um diretório chamado resources. Dentro de resources, crie um diretório chamado css. Crie um arquivo chamado k19.css contendo o estilo desejado e adicione-o ao diretório WebContent/resources/css.

```

1 body {
2   font-family: arial, helvetica, sans-serif;
3   font-size: 14px;
4 }
5
6 h1 {
7   color: #006699;
8   font-size: 18px;
9 }
10
11 ul {
12   list-style-type: square;
13 }
14
15 input {
16   background-color: #E6E6FA;
17   border: solid 1px #000000;
18 }

```

Código CSS 6.1: k19.css

- 17** Use a tag <h:outputStylesheet> para incluir o arquivo de estilos na página definida por cursos.xhtml. Use a tag <ui:repeat> para exibir os cursos.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8
9  <h:head>
10   <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13
14  <h:outputStylesheet name="k19.css" library="css"/>
15
16 <h:form>
17   <h:panelGrid columns="2">
18     <h:outputLabel value="Sigla: " for="campo-sigla" />
19     <h:inputText value="#{cursosBean.curso.sigla}" id="campo-sigla" />
20
21     <h:outputLabel value="Nome: " for="campo-nome" />
22     <h:inputText value="#{cursosBean.curso.nome}" id="campo-nome" />
23
24     <h:commandButton value="Adicionar" action="#{cursosBean.adicionaCurso}" />
25   </h:panelGrid>
26 </h:form>
27
28 <h1>Alguns cursos da K19:</h1>
29 <ul>
30   <ui:repeat value="#{cursosBean.cursos}" var="curso">
31     <li>
32       <h:outputText value="#{curso.sigla}: #{curso.nome}" />
33     </li>
34   </ui:repeat>
35 </ul>
36 </h:body>
37 </html>
```

Código XHTML 6.58: cursos.xhtml



Exercícios Complementares

- 4** Considere novamente a página de adição e listagem de cursos. Modifique sua aplicação de forma que uma mensagem de confirmação seja exibida na tela após o usuário adicionar um novo curso.



TEMPLATES E MODULARIZAÇÃO

Certamente, você já ouviu alguém falar da importância da reutilização de código no desenvolvimento de software. Os objetivos principais dessa reutilização são diminuir o tempo e o custo do desenvolvimento e da manutenção das aplicações.

Com essa ideia em mente, um projeto chamado **Facelets** foi desenvolvido com o objetivo principal de facilitar todo o processo de desenvolvimento e manutenção das telas de uma aplicação JSF. O Facelets já faz parte do JSF 2 e é a engine padrão para o gerenciamento das telas de aplicações web JSF 2.

Templates

A reutilização do código das telas é realizada principalmente pelo uso de templates. A ideia é identificar um padrão em um determinado conjunto de telas e defini-lo através de um esqueleto (template). Esse esqueleto é formado por trechos estáticos e dinâmicos. O posicionamento desses trechos é sempre fixo. Contudo, ao contrário do conteúdo de um trecho estático, o conteúdo de um trecho dinâmico pode diferir de tela para tela.

Por exemplo, considere uma aplicação na qual todas as telas são divididas em três partes: cabeçalho, corpo e rodapé. Essas partes estão sempre posicionadas da mesma forma. O cabeçalho é posicionado no topo da página, o rodapé, no final e o corpo, entre os dois anteriores. O conteúdo do cabeçalho e do rodapé é sempre o mesmo em todas as telas da aplicação. Por outro lado, o conteúdo do corpo varia de tela para tela.

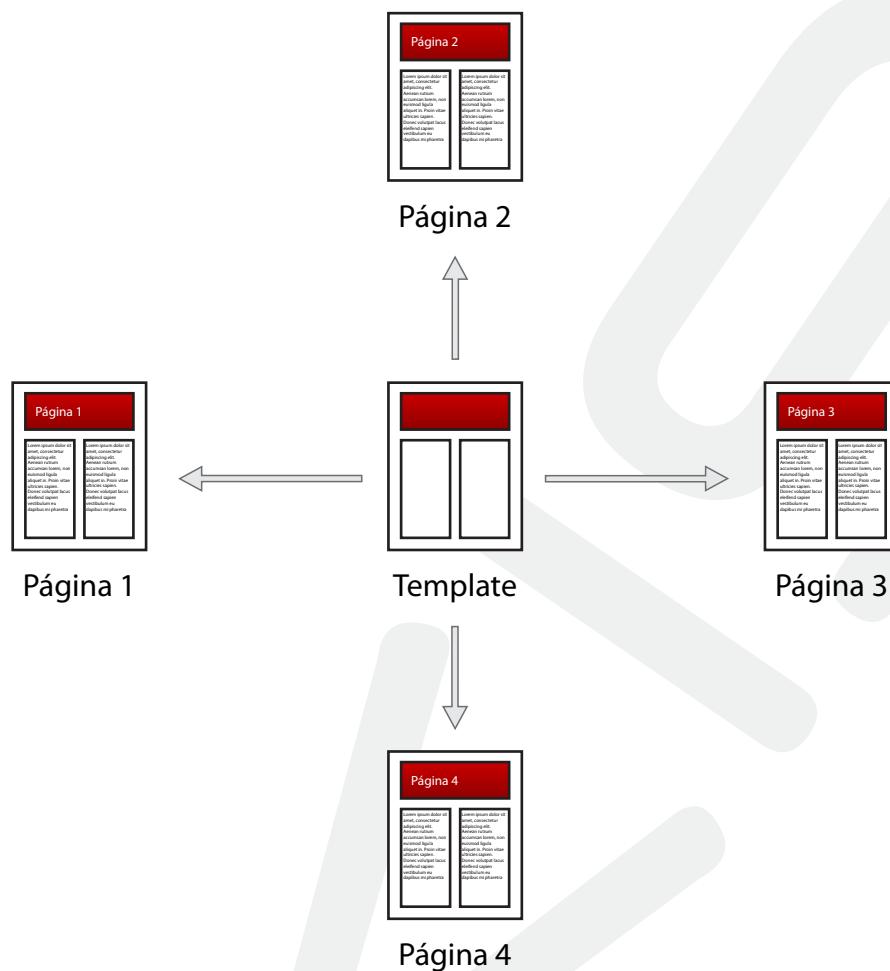


Figura 7.1: Template utilizado por diversas páginas

Criando um template

A criação de um template é simples. Basta criar um arquivo XHTML e definir o posicionamento dos trechos estáticos e dinâmicos. O conteúdo dos trechos estáticos também deve ser definido da forma usual dentro do template. Por outro lado, o conteúdo dos trechos dinâmicos só será definido nas telas. Para indicar o posicionamento dos trechos dinâmicos, devemos utilizar a tag `<ui:insert>`.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:ui="http://java.sun.com/jsf/facelets">
7
8  <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <div id="header">
13     
14     <hr />
15   </div>
16 
```

```

17 <ui:insert name="corpo-da-pagina"> Espaço para o conteúdo da tela </ui:insert>
18
19 <div id="footer" style="text-align: center">
20   <hr />
21   &copy; 2010 K19. Todos os direitos reservados.
22 </div>
23 </h:body>
24 </html>

```

Código XHTML 7.1: template.xhtml

O componente `<ui:insert>` é utilizado para posicionar os trechos dinâmicos. O atributo `name` desse componente é utilizado para nomear os trechos dinâmicos.

Normalmente, a declaração `<!DOCTYPE>` não varia de tela para tela dentro de uma aplicação. Dessa forma, podemos adicioná-la de forma estática no template, conforme o exemplo acima.

Vimos no Capítulo 6 que é fundamental a utilização dos componentes `<h:head>` e `<h:body>`. No exemplo acima, esses componentes foram definidos estaticamente no template. Assim, todas as telas que utilizam esse template não correm mais o risco de ficarem sem esses componentes.

Utilizando templates

Para criar uma tela que usa um determinado template, devemos criar um arquivo XHTML e adicionar a tag `<ui:composition>` a esse arquivo. O arquivo do template desejado deve ser indicado através do atributo `template` da tag `<ui:composition>`.

```

1 <ui:composition template="/template.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets">
4 </ui:composition>

```

Código XHTML 7.2: tela.xhtml

Todo conteúdo não contido na tag `<ui:composition>` será descartado pelo JSF no processo de construção da tela.

O conteúdo de um trecho dinâmico pode ser definido através do componente `<ui:define>`. Esse componente possui o atributo `name` que é utilizado para indicar qual trecho dinâmico do template queremos definir.

No exemplo acima, podemos definir o conteúdo do trecho dinâmico `corpo-da-pagina` da seguinte forma:

```

1 <ui:composition template="/template.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:ui="http://java.sun.com/jsf/facelets">
5
6   <ui:define name="corpo-da-pagina">
7     <h1>Lista de treinamentos</h1>
8     <h2>K11 - Orientação a Objetos em Java</h2>
9     <h2>K12 - Desenvolvimento Web com JSF2 e JPA2</h2>
10    </ui:define>
11 </ui:composition>

```

Código XHTML 7.3: tela.xhtml

Se o conteúdo de um trecho dinâmico não for definido, o JSF utilizará o conteúdo existente no corpo da tag <ui:insert> definido no template. No exemplo acima, caso o conteúdo do trecho dinâmico `corpo-da-pagina` não fosse definido no arquivo `tela.xhtml`, o JSF utilizaria o texto “Espaço para o conteúdo da tela”, que foi definido no arquivo `template.xhtml` como conteúdo padrão para esse trecho.



Pare para pensar...

Em geral, os templates não definem uma tela concreta da aplicação. Eles funcionam como uma base para a criação de telas concretas. Dessa forma, não seria interessante que os navegadores acessassem diretamente a tela correspondente a um template, pois, provavelmente, essa tela estaria incompleta. Para evitar esse comportamento indesejado, podemos colocar todos os templates dentro do diretório /WEB-INF.



Exercícios de Fixação

- 1** Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Templates-e-Modularizacao** seguindo os passos vistos no exercício do Capítulo 5.
- 2** Crie um diretório chamado `templates` na pasta WEB-INF. Crie um template chamado `template.xhtml` na pasta WEB-INF/templates. Copie o arquivo `k19-logo.png` da pasta K19-Arquivos/imagens, que está na Área de Trabalho, para a pasta WebContent do projeto K19-Templates-e-Modularizacao.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:h="http://java.sun.com/jsf/html"
6      xmlns:ui="http://java.sun.com/jsf/facelets">
7
8  <h:head>
9      <title>K19 Treinamentos</title>
10
11 <h:body>
12     <div id="header">
13     <hr />
14     </div>
15
16     <ui:insert name="conteudo"> Espaço para o conteúdo da tela </ui:insert>
17
18     <div id="footer" style="text-align: center">
19         <hr />
20         &copy; 2012 K19. Todos os direitos reservados.</div>
21
22 </h:body>
</html>
```

Código XHTML 7.4: `template.xhtml`

- 3** Monte uma tela que use o template criado no exercício anterior. O nome do arquivo deve ser `formulario.xhtml`.

```

1  <ui:composition template="/WEB-INF/templates/template.xhtml"
2      xmlns="http://www.w3.org/1999/xhtml"
3      xmlns:h="http://java.sun.com/jsf/html"
4      xmlns:ui="http://java.sun.com/jsf/facelets">
```

```

6  <ui:define name="conteudo">
7      <h:form>
8          <h:outputLabel value="Nome: " for="campo-nome" />
9          <h:inputText id="campo-nome" />
10         <h:commandButton value="Enviar" />
11     </h:form>
12 </ui:define>
13 </ui:composition>

```

Código XHTML 7.5: formulario.xhtml

Verifique o resultado acessando a url:

<http://localhost:8080/K19-Templates-e-Modularizacao/formulario.xhtml>

Modularização

Os trechos estáticos ou dinâmicos definidos em um template possuem posição fixa. Em determinadas situações, é necessário tornar flexível o posicionamento de um determinado trecho.

Por exemplo, considere uma aplicação com 50 páginas diferentes. Os usuários dessa aplicação podem enviar mensagens para o administrador do sistema quando identificarem algum problema. Essas mensagens são enviadas através de um formulário HTML. Esse formulário de contato deve ser exibido em todas as telas da aplicação.

Nesse caso, poderíamos pensar em criar um template e definir o formulário de contato como um trecho estático desse template. Contudo, há uma restrição importante que descarta essa abordagem. O formulário de contato deve ser exibido em posições diferentes nas telas da aplicação. Por exemplo, em algumas telas o formulário aparecerá no topo e em outras ele aparecerá no centro.

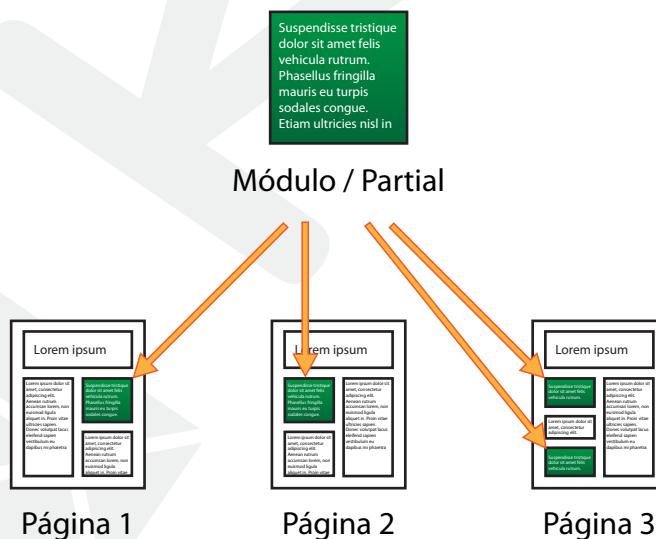


Figura 7.2: Módulo reaproveitado em diversas páginas

Nessa situação, a melhor abordagem é definir o formulário de contato, separadamente, em um arquivo XHTML. O código XHTML que define o formulário de contato deve ser inserido no corpo da tag `<ui:composition>`. Observe o código abaixo.

```

1 <ui:composition
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:ui="http://java.sun.com/jsf/facelets">
5
6   <h:form>
7     <h:panelGrid>
8       <h:outputLabel value="Mensagem: " for="mensagem"/>
9       <h:inputTextarea id="mensagem" value="#{mensagensBean.mensagem}"/>
10      <h:commandButton value="Enviar" action="#{mensagensBean.envia}"/>
11    </h:panelGrid>
12  </h:form>
13 </ui:composition>
```

Código XHTML 7.6: formulario-de-contato.xhtml

Com o formulário de contato definido em um arquivo separado, podemos utilizar a tag `<ui:include>` para adicioná-lo onde for necessário.

```

1 ...
2 <ui:include src="/formulario-de-contato.xhtml"/>
3 ...
```

Código XHTML 7.7: Inserindo o formulário de contato

Parâmetros

Em algumas situações é necessário passar parâmetros de um arquivo XHTML para outro. Por exemplo, considere a aplicação de uma livraria. Na primeira página dessa aplicação, uma lista com livros mais vendidos é exibida no canto direito. Em outra página, uma lista com os livros mais caros é exibida no canto esquerdo.

Podemos criar um arquivo XHTML e atribuir a ele a tarefa de exibir uma lista qualquer de livros independentemente de posicionamento.

```

1 <ui:composition
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:ui="http://java.sun.com/jsf/facelets">
5
6   <h:dataTable value="#{livros}" var="livro">
7     ...
8   </h:dataTable>
9 </ui:composition>
```

Código XHTML 7.8: lista-livros.xhtml

Observe, no código acima, que o componente `<h:dataTable>` foi vinculado a uma variável chamada `livros`. Essa variável é um parâmetro do arquivo `lista-livros.xhtml`.

Na primeira página da aplicação da livraria, podemos adicionar o arquivo `lista-livros.xhtml` com a tag `<ui:include>`, passando como parâmetro a lista dos livros mais vendidos. Para isso, devemos utilizar o componente `<ui:param>`.

```

1 ...
2 <ui:include src="/lista-livros.xhtml">
3   <ui:param name="livros" value="#{livrosBean.listaDosLivrosMaisVendidos}"/>
```

```

4  </ui:include>
5  ...

```

Código XHTML 7.9: Inserindo a lista dos livros mais vendidos

Em outra página da aplicação da livraria, podemos adicionar o arquivo `lista-livros.xhtml` passando como parâmetro a lista dos livros mais caros.

```

1  ...
2  <ui:include src="/lista-livros.xhtml">
3    <ui:param name="livros" value="#{livrosBean.listaDosLivrosMaisCaros}" />
4  </ui:include>
5  ...

```

Código XHTML 7.10: Inserindo a lista dos livros mais caros



Exercícios de Fixação

- 4 Vamos implementar uma listagem de instrutores no nosso projeto K19-Templates-e-Modularizacao. O primeiro passo é criar uma classe para modelar os instrutores. Crie um pacote chamado `model` no projeto K19-Templates-e-Modularizacao e adicione nele uma classe chamada `Instrutor` com seguinte código:

```

1 package model;
2
3 public class Instrutor {
4     private String nome;
5     private String dataDeNascimento;
6
7     // GETTERS E SETTERS
8 }

```

Código Java 7.1: Instrutor.java

- 5 Faça um managed bean que forneça uma lista de instrutores para uma tela de listagem de instrutores. Crie um pacote chamado `managedbeans` no projeto K19-Templates-e-Modularizacao e adicione nele uma classe chamada `InstrutorBean` com seguinte código:

```

1 package managedbeans;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import model.Instrutor;
6 import javax.faces.bean.ManagedBean;
7
8 @ManagedBean
9 public class InstrutorBean {
10
11     private List<Instrutor> instrutores = new ArrayList<Instrutor>();
12
13     public InstrutorBean() {
14         Instrutor rafael = new Instrutor();
15         rafael.setNome("Rafael Cosentino");
16         rafael.setDataDeNascimento("30/10/1984");
17
18         Instrutor marcelo = new Instrutor();
19         marcelo.setNome("Marcelo Martins");
20         marcelo.setDataDeNascimento("02/04/1985");
21     }

```

```

22     this.instrutores.add(rafael);
23     this.instrutores.add(marcelo);
24 }
25
26     public List<Instrutor> getInstrutores() {
27         return instrutores;
28     }
29
30     public void setInstrutores(List<Instrutor> instrutores) {
31         this.instrutores = instrutores;
32     }
33 }
```

Código Java 7.2: InstrutorBean.java

- 6** Crie uma tela parcial para mostrar os dados de apenas um instrutor dentro de um item de uma lista HTML. O arquivo deve ser adicionado na pasta WebContent do projeto K19-Templates-e-Modularizacao e se chamar `instrutor-info.xhtml`.

```

1 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:h="http://java.sun.com/jsf/html"
3   xmlns:ui="http://java.sun.com/jsf/facelets">
4
5 <li>
6   <h:outputText value="Nome: #{instrutor.nome}"/>
7   <br/>
8   <h:outputText value="Data Nascimento: #{instrutor.dataDeNascimento}"/>
9 </li>
10
11 </ui:composition>
```

Código XHTML 7.11: instrutor-info.xhtml

- 7** Faça a tela principal da listagem de instrutores. Crie um arquivo na pasta WebContent do projeto K19-Templates-e-Modularizacao com o nome `listagem-de-instrutores.xhtml` e com o seguinte código.

```

1 <ui:composition template="/WEB-INF/templates/template.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core"
5   xmlns:ui="http://java.sun.com/jsf/facelets">
6
7 <ui:define name="conteudo">
8   <ul>
9     <ui:repeat var="instrutor" value="#{instrutorBean.instrutores}">
10       <ui:include src="instrutor-info.xhtml">
11         <ui:param name="instrutor" value="#{instrutor}"/>
12       </ui:include>
13     </ui:repeat>
14   </ul>
15 </ui:define>
16 </ui:composition>
```

Código XHTML 7.12: listagem-de-instrutores.xhtml

Veja o resultado acessando a url:

<http://localhost:8080/K19-Templates-e-Modularizacao/listagem-de-instrutores.xhtml>



Exercícios Complementares

- 1 Usando templates, você deve criar uma página para exibir os detalhes dos produtos de uma loja virtual. A página de apresentação dos produtos deve ter o seguinte formato. Uma imagem do produto deve ser apresentada do lado esquerdo da página. O nome e o preço do produto devem ser exibidos do lado direito da imagem. Uma descrição do produto deve aparecer abaixo de tudo. Além disso, o logotipo da loja deve aparecer no topo da página e, logo abaixo, deve haver uma caixa de seleção para escolher o produto cujos detalhes serão exibidos.



NAVEGAÇÃO

Navegar entre as telas de uma aplicação web é preciso. O mecanismo de navegação do JSF é bem sofisticado e permite vários tipos de transições entre telas. A ideia é muito simples: no clique de um botão ou link, muda-se a tela apresentada ao usuário.

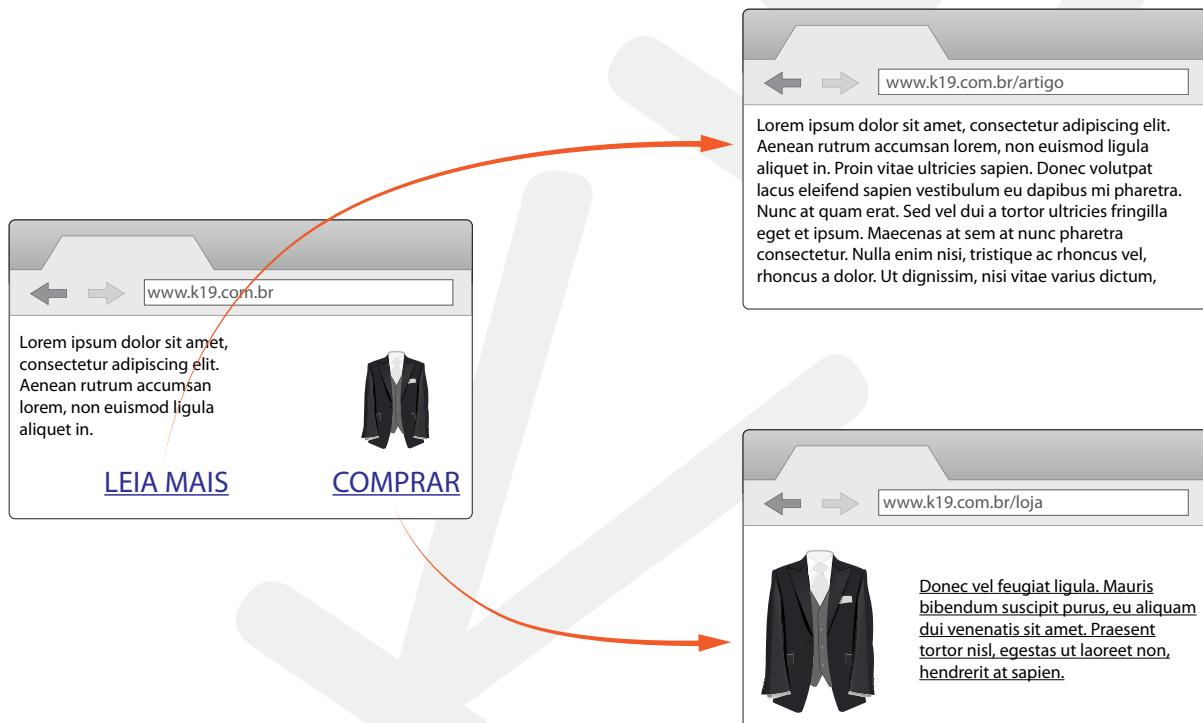


Figura 8.1: Navegação entre páginas

Navegação Implícita

Na navegação implícita, quando o usuário clica em algum botão ou link, um sinal (*outcome*) é enviado para o JSF. Esse sinal é uma string que será utilizada pelo tratador de navegação do JSF para definir a próxima tela que será apresentada ao usuário.

Considere, por exemplo, um link ou botão de uma tela (definida pelo arquivo `pagina1.xhtml`) que envia o outcome “`pagina2`”. Quando um usuário clicar nesse link ou botão, ele será redirecionado para a tela definida pelo arquivo `pagina2.xhtml`. Esse arquivo deve estar no mesmo diretório do arquivo `pagina1.xhtml`. Nesse caso, o outcome é igual ao nome do arquivo de resposta sem o sufixo `.xhtml`.

Quando utilizamos os componentes `<h:commandButton>` e `<h:commandLink>` para criar botões e links, devemos definir os outcomes através do atributo `action`. Por outro lado, quando utilizamos os componentes `<h:button>` e `<h:link>`, devemos definir os outcomes através do atributo `outcome`.

Veja como seria o código da `pagina1.xhtml` e `pagina2.xhtml`.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h1>K19 Página 1</h1>
12   <h:form>
13     <h:commandButton value="Página 2" action="pagina2"/>
14   </h:form>
15 </h:body>
16 </html>

```

Código XHTML 8.1: pagina1.xhtml

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h1>K19 Página 2</h1>
12   <h:link outcome="pagina1">
13     <h:outputText value="Página 1"/>
14   </h:link>
15 </h:body>
16 </html>

```

Código XHTML 8.2: pagina2.xhtml

Além do nome, os outcomes também podem determinar o caminho do arquivo de resposta. Se o outcome começar com caractere “/”, esse caminho será definido a partir do diretório raiz da aplicação web . Caso contrário, será definido a partir do diretório atual.

Considere, por exemplo, um arquivo chamado `pagina1.xhtml` no diretório `/diretorio1` e um arquivo chamado `pagina2.xhtml` no diretório `/diretorio1/diretorio2`. No arquivo `pagina1.xhtml`, podemos adicionar um botão para navegar para o arquivo `pagina2.xhtml` da seguintes formas:

```

1 ...
2 <h:commandButton value="Página 2" action="/diretorio1/diretorio2/pagina2"/>
3 ...

```

Código XHTML 8.3: pagina1.xhtml

```

1 ...
2 <h:commandButton value="Página 2" action="diretorio2/pagina2"/>
3 ...

```

Código XHTML 8.4: pagina1.xhtml

Note que, no primeiro código, o caminho é definido a partir do diretório raiz da aplicação web. Já no segundo código, o caminho é definido relativamente ao diretório atual.

Navegação Explícita

Na navegação explícita, podemos associar um outcome a um arquivo de resposta independentemente do nome ou do caminho desse arquivo. Essa associação deve ser registrada no arquivo de configuração do JSF, o faces-config.xml.

Para registrar uma navegação explícita, podemos adicionar três informações no arquivo de configuração do JSF:

- O caminho do arquivo que define a tela de origem.
- O outcome.
- O caminho do arquivo que define a tela de resposta.

Veja o exemplo abaixo.

```

1 <navigation-rule>
2   <from-view-id>/pagina1.xhtml</from-view-id>
3
4   <navigation-case>
5     <from-outcome>proxima</from-outcome>
6     <to-view-id>/pagina2.xhtml</to-view-id>
7   </navigation-case>
8 </navigation-rule>
```

Código XML 8.1: Registrando uma navegação explícita

Após o registro da navegação acima, podemos adicionar um botão ou link no arquivo pagina1.xhtml com o outcome proxima para navegar para o arquivo pagina2.xhtml.

```

1 ...
2 <h:commandButton value="Próxima tela" action="proxima"/>
3 ...
```

Código XHTML 8.5: pagina1.xhtml



Importante

O JSF utiliza a seguinte lógica para determinar a página de resposta. Primeiro, ele verifica se o outcome é compatível com alguma regra de navegação registrada no arquivo faces-config.xml. Caso seja, o JSF realizará uma navegação explícita processando essa regra. Caso contrário, o JSF tentará realizar uma navegação implícita, procurando um arquivo compatível com o outcome. Se esse arquivo não existir, a tela atual será reapresentada.



Mais Sobre

Podemos registrar uma regra de navegação para diversos arquivos utilizando wildcards. Considere o exemplo abaixo.

```

1 <navigation-rule>
2   <from-view-id>/K19/*</from-view-id>
3
4   <navigation-case>
5     <from-outcome>home</from-outcome>
6     <to-view-id>/home.xhtml</to-view-id>
7   </navigation-case>
8 </navigation-rule>

```

Código XML 8.2: Registrando uma navegação explícita

O caractere “*” na tag `<from-view-id>` indica que essa regra de navegação será aplicada para todo arquivo do diretório /K19.



Mais Sobre

Podemos criar navegações condicionais. Considere, por exemplo, uma aplicação em que os usuários podem escolher a versão da interface que desejam utilizar. As regras de navegação podem então considerar a preferência do usuário. Para isso, podemos adicionar a tag `<if>` na definição de uma regra de navegação. Veja o exemplo abaixo.

```

1 <navigation-rule>
2   <from-view-id>/login.xhtml</from-view-id>
3
4   <navigation-case>
5     <from-outcome>sucesso</from-outcome>
6     <if>
7       #{usuarioBean.preferencias['versao-da-interface'] == 'simples'}
8     </if>
9     <to-view-id>/home-simples.xhtml</to-view-id>
10   </navigation-case>
11
12   <navigation-case>
13     <from-outcome>sucesso</from-outcome>
14     <if>
15       #{usuarioBean.preferencias['versao-da-interface'] == 'avancada'}
16     </if>
17     <to-view-id>/home-avancada.xhtml</to-view-id>
18   </navigation-case>
19 </navigation-rule>

```

Código XML 8.3: Registrando uma navegação explícita condicional



Exercícios de Fixação

- 1 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Navegacao** seguindo os passos vistos nos exercícios de fixação do Capítulo 5.
- 2 Na pasta WebContent do projeto K19-Navegacao, crie três arquivos XHTML: `componentes-visuais.xhtml`, `templates-e-modularizacao.xhtml` e `navegacao.xhtml`. A partir de qualquer um deles, deve ser possível navegar para qualquer outro. Podemos criar uma espécie de menu para navegar entre as páginas:

1 ...

```

1 <h:link value="Componentes Visuais" outcome="componentes-visuais"/> -
2 <h:link value="Templates e Modularização" outcome="templates-e-modularizacao"/> -
3 <h:link value="Navegação" outcome="navegacao"/>
4 ...
5 ...

```

Código XHTML 8.6: Menu para navegar entre as páginas.

- 3 Para facilitar, podemos definir um template para criar essas três páginas. Crie um arquivo chamado `template.xhtml` no diretório `/WEB-INF/templates/`.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:ui="http://java.sun.com/jsf/facelets">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12
13   <div id="header" style="text-align: center">
14     <h1>K19 - Desenvolvimento Web com JSF2 e JPA2</h1>
15   </div>
16
17   <h:link value="Componentes Visuais" outcome="componentes-visuais"/> -
18   <h:link value="Templates e Modularização" outcome="templates-e-modularizacao"/> -
19   <h:link value="Navegação" outcome="navegacao"/>
20
21   <ui:insert name="corpo-da-pagina"> Espaço para o conteúdo da tela </ui:insert>
22
23   <div id="footer" style="text-align: center">
24     <hr />
25     &copy; 2012 K19. Todos os direitos reservados.
26   </div>
27
28 </h:body>
29 </html>

```

Código XHTML 8.7: template.xhtml

Agora, use esse template para criar as páginas `componentes-visuais.xhtml`, `templates-e-modularizacao.xhtml` e `navegacao.xhtml`

```

1 <ui:composition template="/WEB-INF/templates/template.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:ui="http://java.sun.com/jsf/facelets">
5
6   <ui:define name="corpo-da-pagina">
7     <h1>Componentes Visuais</h1>
8
9     No JSF 2, as telas são definidas em arquivos xhtml. Os componentes visuais que
10    constituem as telas são adicionados por meio de tags. A especificação do JSF
11    define uma grande quantidade de tags e as classifica em bibliotecas.
12    As principais bibliotecas de tags do JSF são:
13    <ul>
14      <li>Core (http://java.sun.com/jsf/core)</li>
15      <li>HTML (http://java.sun.com/jsf/html)</li>
16      <li>Facelets (http://java.sun.com/jsf/facelets)</li>
17    </ul>
18  </ui:define>
19 </ui:composition>

```

Código XHTML 8.8: componentes-visuais.xhtml

```

1 <ui:composition template="/WEB-INF/templates/template.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:ui="http://java.sun.com/jsf/facelets">
5
6   <ui:define name="corpo-da-pagina">
7     <h1>Templates e Modularização</h1>
8
9     A criação de um template é simples. Basta criar um arquivo XHTML e definir o
10    posicionamento dos trechos estáticos e dinâmicos. O conteúdo dos trechos
11    estáticos também deve ser definido da forma usual dentro do template. Por outro
12    lado, o conteúdo dos trechos dinâmicos só será definido nas telas. Para indicar
13    o posicionamento dos trechos dinâmicos, devemos utilizar a tag ui:insert.
14   </ui:define>
15 </ui:composition>
```

Código XHTML 8.9: templates-e-modularizacao.xhtml

```

1 <ui:composition template="/WEB-INF/templates/template.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:ui="http://java.sun.com/jsf/facelets">
5
6   <ui:define name="corpo-da-pagina">
7     <h1>Navegação</h1>
8
9     Navegar entre as telas de uma aplicação web é preciso. O mecanismo de
10    navegação do JSF é bem sofisticado e permite vários tipos de transições entre
11    telas. A ideia é muito simples: no clique de um botão ou link, muda-se a tela
12    apresentada ao usuário.
13   </ui:define>
14 </ui:composition>
```

Código XHTML 8.10: navegacao.xhtml

- 4** Navegue através dos links (ou botões) da url:

<http://localhost:8080/K19-Navegacao/componentes-visuais.xhtml>

- 5** Suponha que a sequência dos tópicos seja “Componentes Visuais”, “Templates e Modularização” e “Navegação”. Queremos adicionar dois botões às páginas: um para avançar para o próximo tópico e outro para voltar para o tópico anterior. Para isso, configure uma navegação explícita no arquivo faces-config.xml do diretório /WebContent/WEB-INF/.

```

1 ...
2 <navigation-rule>
3   <from-view-id>componentes-visuais.xhtml</from-view-id>
4
5   <navigation-case>
6     <from-outcome>proximo</from-outcome>
7     <to-view-id>templates-e-modularizacao.xhtml</to-view-id>
8   </navigation-case>
9 </navigation-rule>
10
11 <navigation-rule>
12   <from-view-id>templates-e-modularizacao.xhtml</from-view-id>
13
14   <navigation-case>
15     <from-outcome>proximo</from-outcome>
16     <to-view-id>navegacao.xhtml</to-view-id>
17   </navigation-case>
18
19   <navigation-case>
20     <from-outcome>anterior</from-outcome>
21     <to-view-id>componentes-visuais.xhtml</to-view-id>
```

```

22   </navigation-case>
23 </navigation-rule>
24
25 <navigation-rule>
26   <from-view-id>navegacao.xhtml</from-view-id>
27
28   <navigation-case>
29     <from-outcome>anterior</from-outcome>
30     <to-view-id>templates-e-modularizacao.xhtml</to-view-id>
31   </navigation-case>
32 </navigation-rule>
33 ...

```

Código XML 8.4: faces-config.xml

- 6** No arquivo /WEB-INF/templates/template.xhtml, adicione dois botões. O primeiro deve ser usado para navegar para o tópico anterior (logo, deve emitir o sinal “anterior”). O segundo botão deve ser usado para navegar para o próximo tópico (portanto, deve emitir o sinal “proximo”).

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:ui="http://java.sun.com/jsf/facelets">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12
13   <div id="header" style="text-align: center">
14     <h1>K19 - Desenvolvimento Web com JSF2 e JPA2</h1>
15   </div>
16
17   <h:form>
18     <h:commandButton value="Anterior" action="anterior"/>
19     <h:commandButton value="Próximo" action="proximo"/>
20   </h:form>
21
22   <h:link value="Componentes Visuais" outcome="componentes-visuais"/> -
23   <h:link value="Templates e Modularização" outcome="templates-e-modularizacao"/> -
24   <h:link value="Navegação" outcome="navegacao"/>
25
26   <ui:insert name="corpo-da-pagina"> Espaço para o conteúdo da tela </ui:insert>
27
28   <div id="footer" style="text-align: center">
29     <hr />
30     &copy; 2010 K19. Todos os direitos reservados.
31   </div>
32
33 </h:body>
34 </html>

```

Código XHTML 8.11: template.xhtml

- 7** Navegue através dos links e botões da url:

<http://localhost:8080/K19-Navegacao/componentes-visuais.xhtml>

Navegações Estática e Dinâmica

Até agora, utilizamos apenas navegações estáticas, pois os outcomes foram definidos nos botões e links. Dessa forma, toda vez que um botão ou link é clicado, ele emite sempre o mesmo outcome. Por exemplo, o botão abaixo sempre emite o outcome “k19”.

```
1 <h:commandButton value="K19" action="k19"/>
```

Código XHTML 8.12: Botão com navegação estática

Podemos tornar esse comportamento dinâmico, fazendo com que os outcomes sejam diferentes a cada clique. Para isso, os botões e links devem ser associados a métodos de ação de managed beans através de expression language. Veja o exemplo abaixo.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h1>K19 Cara ou Coroa</h1>
12   <h:form>
13     <h:commandButton value="Lançar Moeda" action="#{caraOuCoroaBean.lanca}" />
14   </h:form>
15 </h:body>
16 </html>
```

Código XHTML 8.13: cara-ou-coroa.xhtml

```
1 @ManagedBean
2 public class CaraOuCoroaBean {
3   public String lancia() {
4     if(Math.random() < 0.5) {
5       return "cara";
6     } else {
7       return "coroa";
8     }
9   }
10 }
```

Código Java 8.1: CaraOuCoroaBean.java

Toda vez que o botão “Lançar Moeda” for clicado, o método `lancia()` do managed bean `caraOuCoroaBean` será chamado. A string devolvida por esse método será considerada o outcome a ser utilizado na navegação.



Mais Sobre

Vimos que a página de resposta depende tanto da página de origem quanto do outcome. No caso da navegação dinâmica, a página de resposta também pode depender do método de ação que produziu o outcome. Para isso, basta acrescentar a tag `<from-action>` na definição da regra de navegação. Veja o exemplo abaixo.

```
1 <navigation-rule>
2   <from-view-id>/index.xhtml</from-view-id>
3
4   <navigation-case>
```

```

5   <from-outcome>lista</from-outcome>
6   <from-action>#{cursosBean.lista}</from-action>
7   <to-view-id>/lista-cursos.xhtml</to-view-id>
8   </navigation-case>
9
10  <navigation-case>
11    <from-outcome>lista</from-outcome>
12    <from-action>#{alunosBean.lista}</from-action>
13    <to-view-id>/lista-alunos.xhtml</to-view-id>
14  </navigation-case>
15 </navigation-rule>

```

Código XML 8.5: faces-config.xml

```

1 ...
2 <h:commandLink value="Lista cursos" action="#{cursosBean.lista}"/>
3 <h:commandLink value="Lista alunos" action="#{alunosBean.lista}"/>
4 ...

```

Código XHTML 8.14: index.xhtml

Se o link “Lista cursos” for clicado, o método `lista()` do managed bean `cursosBean` será acionado. Suponha que esse método devolva “lista”. De acordo com a regra de navegação definida no `faces-config.xml`, o usuário será direcionado à página definida pelo arquivo `/lista-cursos.xhtml`.

Por outro lado, se o usuário clicar no link “Lista alunos”, o método `lista()` do managed bean `alunosBean` será invocado. Se esse método devolver “lista”, o usuário será direcionado à página definida pelo arquivo `/lista-alunos.xhtml`, conforme a regra de navegação definida no `faces-config.xml`.



Exercícios de Fixação

- 8** Implemente um managed bean que, de forma aleatória, escolha entre dois outcomes. Crie um pacote chamado `managedbeans` no projeto K19-Navegacao e adicione uma classe chamada `CaraOuCoroaBean`.

```

1 @ManagedBean
2 public class CaraOuCoroaBean {
3     public String proxima() {
4         if(Math.random() < 0.5) {
5             return "cara";
6         } else {
7             return "coroa";
8         }
9     }
10 }

```

Código Java 8.2: CaraOuCoroaBean.java

- 9** Crie uma tela principal com um botão que chama o managed bean do exercício anterior para escolher o outcome que deve ser emitido para o JSF. Para isso, faça um arquivo chamado `cara-ou-coroa.xhtml` na pasta `WebContent` do projeto K19-Navegacao.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h1>K19 Cara ou Coroa</h1>
12   <h:form>
13     <h:commandButton value="Lançar Moeda" action="#{caraOuCoroaBean.proxima}" />
14   </h:form>
15 </h:body>
16 </html>

```

Código XHTML 8.15: cara-ou-coroa.xhtml

- 10** Crie os arquivos de saída.

O arquivo cara.xhtml:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h1>Deu Cara!</h1>
12   <h:form>
13     <h:commandButton value="voltar" action="cara-ou-coroa"/>
14   </h:form>
15 </h:body>
16 </html>

```

Código XHTML 8.16: cara.xhtml

O arquivo coroa.xhtml:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h1>Deu Coroa!</h1>
12   <h:form>
13     <h:commandButton value="voltar" action="cara-ou-coroa"/>
14   </h:form>
15 </h:body>
16 </html>

```

Código XHTML 8.17: coroa.xhtml

- 11** Navegue através dos links e botões da url:

<http://localhost:8080/K19-Navegacao/cara-ou-coroa.xhtml>



Exercícios Complementares

- 1 Considere uma aplicação em que o usuário pode escolher entre acessar uma versão simples ou avançada da aplicação. O usuário pode alterar essa preferência em uma página específica da aplicação. Crie uma aplicação web que tenha três páginas. Uma das páginas deve permitir que o usuário altere suas preferências. As outras duas devem ser a página principal da aplicação, sendo uma para a versão simples e outra para a versão avançada. Cada uma das páginas deve ter um menu para que o usuário possa navegar entre a página principal e a página de configurações. O link para a página principal deve levar em consideração a preferência do usuário. Use navegação **implícita**.

Dica: Para armazenar a preferência do usuário, você pode usar um managed bean. Anote a classe que implementará esse managed bean com `@SessionScoped` para que a preferência do usuário seja mantida entre uma requisição e outra da aplicação.

- 2 Repita o exercício anterior usando navegação **explícita**.



ESCOPOS

Os managed beans são instanciados pelo JSF, ou seja, os desenvolvedores definem as classes e o JSF cuida do “new”. Porém, podemos determinar quando os managed beans devem ser criados e descartados. O tempo de vida de uma instância afeta principalmente a durabilidade dos dados que ela armazena. Por isso, precisamos escolher qual escopo queremos utilizar em cada managed bean. Eis alguns dos escopos do JSF 2:

- Request.
- View.
- Session.
- Application.

Request

No escopo *request*, as instâncias dos managed beans são criadas durante o processamento de uma requisição assim que forem necessárias e descartadas no final desse mesmo processamento. Assim, os dados não são mantidos de uma requisição para outra.

A partir do JSF 2, os managed beans podem ser registrados através da anotação @ManagedBean. O JSF utiliza o escopo *request* como padrão quando managed beans são registrados usando essa anotação. Mesmo sendo o padrão, podemos deixar explícito a escolha do escopo *request* através da anotação @RequestScoped.

```
1 package br.com.k19;
2
3 @ManagedBean
4 @RequestScoped
5 class TesteBean {
6     ...
7 }
```

Código Java 9.1: TesteBean.java

Por outro lado, se o managed bean for registrado da maneira tradicional, no arquivo faces-config.xml, a escolha de um escopo é obrigatória e realizada através da tag managed-bean-scope.

```
1 ...
2 <managed-bean>
3     <managed-bean-name>testeBean</managed-bean-name>
4     <managed-bean-class>br.com.k19.TesteBean</managed-bean-class>
5     <managed-bean-scope>request</managed-bean-scope>
6 </managed-bean>
7 ...
```

Código XML 9.1: faces-config.xml

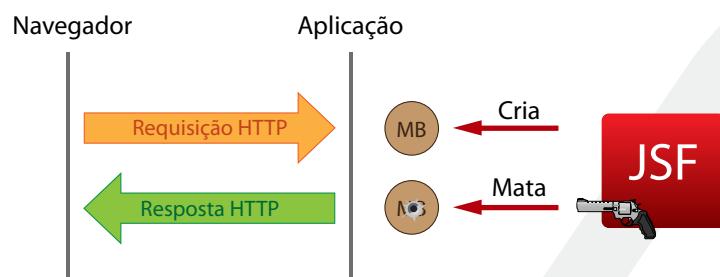


Figura 9.1: Escopo de request

View

O escopo *view* foi adicionado no JSF 2. A ideia é manter determinados dados enquanto o usuário não mudar de tela. As instâncias dos managed beans em escopo *view* são eliminadas somente quando há uma navegação entre telas.

Analogamente, para escolher o escopo *view*, devemos utilizar a anotação `@ViewScoped` ou a tag `managed-bean-scope`.

```

1 package br.com.k19;
2
3 @ManagedBean
4 @ViewScoped
5 class TesteBean {
6     ...
7 }
```

Código Java 9.2: TesteBean.java

```

1 ...
2 <managed-bean>
3     <managed-bean-name>testeBean</managed-bean-name>
4     <managed-bean-class>br.com.k19.TesteBean</managed-bean-class>
5     <managed-bean-scope>view</managed-bean-scope>
6 </managed-bean>
7 ...
```

Código XML 9.2: faces-config.xml

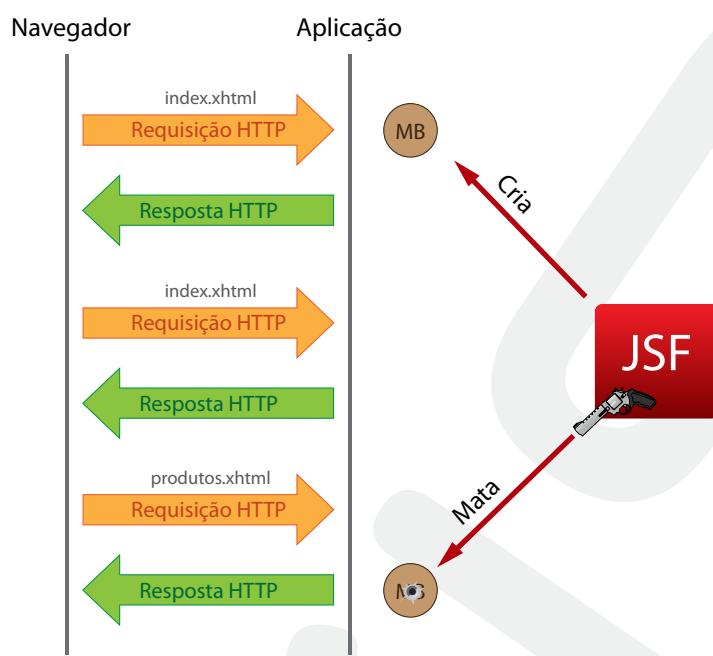


Figura 9.2: Escopo de view

Session

Certas informações devem ser mantidas entre as requisições de um determinado usuário em um determinado navegador. Por exemplo, considere uma aplicação que utiliza carrinho de compras. Um usuário faz diversas requisições para escolher os produtos e colocá-los no seu carrinho. Durante todo esse tempo, a aplicação deve manter a informação de quais produtos já foram escolhidos por esse usuário.

Daí surge o escopo *session*. Cada usuário possui um espaço na memória do servidor que é chamado de *session*. Tecnicamente, é possível existir duas ou mais *sessions* de um mesmo usuário, por exemplo, se ele estiver utilizando dois ou mais navegadores.

As instâncias dos managed beans configurados com o escopo *session* são criadas quando necessárias durante o processamento de uma requisição e armazenadas na *session* do usuário que fez a requisição.

Essas instâncias são eliminadas basicamente em duas situações: a própria aplicação decide por algum motivo específico apagar a *session* de um usuário (por exemplo, quando o usuário faz logout) ou o servidor decide apagar a *session* de um usuário quando esse usuário não faz requisições por um determinado período de tempo. Esse tempo pode ser configurado com o Web Container.

Para escolher o escopo *session*, devemos utilizar a anotação `@SessionScoped` ou a tag `managed-bean-scope`.

```

1 package br.com.k19;
2
3 @ManagedBean
4 @SessionScoped
5 class TesteBean {
```

```

6 ...
7 }

```

Código Java 9.3: *TesteBean.java*

```

1 ...
2 <managed-bean>
3   <managed-bean-name>testeBean</managed-bean-name>
4   <managed-bean-class>br.com.k19.TesteBean</managed-bean-class>
5   <managed-bean-scope>session</managed-bean-scope>
6 </managed-bean>
7 ...

```

Código XML 9.3: *faces-config.xml*

Temos que tomar um cuidado maior ao utilizar o escopo *session*, pois podemos acabar sobre-carregando o servidor. Portanto, a dica é evitar utilizar o escopo *session* quando possível. Para não consumir excessivamente os recursos de memória do servidor, o escopo *request* é mais apropriado.

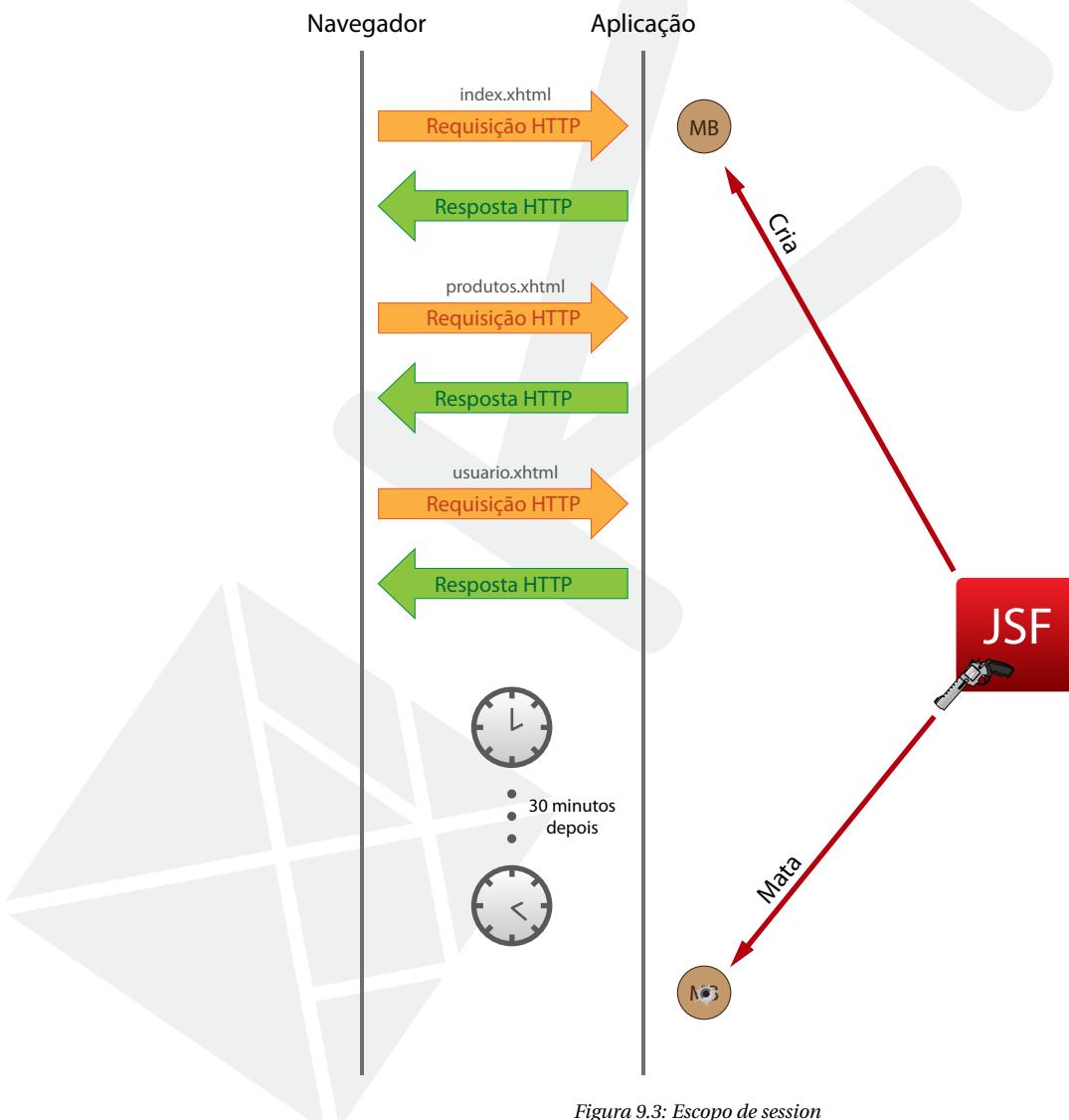


Figura 9.3: Escopo de session

Application

As instâncias dos managed beans configurados com escopo *application* são criadas no primeiro momento em que elas são utilizadas e mantidas até a aplicação ser finalizada.

Ao contrário do que ocorre com escopos discutidos anteriormente, as instâncias dos managed beans registrados com escopo *application* são compartilhadas por todos os usuários da aplicação. O JSF cria apenas uma instância de cada managed bean em escopo de *application*.

Analogamente, para escolher o escopo *application*, devemos utilizar a anotação `@ApplicationScoped` ou a tag `managed-bean-scope`.

```

1 package br.com.k19;
2
3 @ManagedBean
4 @ApplicationScoped
5 class TesteBean {
6     ...
7 }
```

Código Java 9.4: *TesteBean.java*

```

1 ...
2 <managed-bean>
3     <managed-bean-name>testeBean</managed-bean-name>
4     <managed-bean-class>br.com.k19.TesteBean</managed-bean-class>
5     <managed-bean-scope>application</managed-bean-scope>
6 </managed-bean>
7 ...
```

Código XML 9.4: *faces-config.xml*



Mais Sobre

Por padrão, a instância de um managed bean em escopo *application* é criada no momento em que ela é usada pela primeira vez. Podemos alterar esse comportamento, fazendo com que essa instância seja criada quando a aplicação é inicializada. Para isso, basta acrescentar a propriedade `eager` com o valor `true` na anotação `@ManagedBean`. Veja o exemplo abaixo.

```

1 package br.com.k19;
2
3 @ManagedBean(eager=true)
4 @ApplicationScoped
5 class TesteBean {
6     ...
7 }
```

Código Java 9.5: *TesteBean.java*



Exercícios de Fixação

- 1 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Escopos** seguindo os passos vistos no exercício do Capítulo 5.
- 2 Vamos criar uma página para adicionar carros e visualizá-los. Na pasta `src`, faça um pacote chamado `model`.
- 3 No pacote `model`, crie uma classe chamada `Carro` para modelar os carros. Essa classe deve ter dois atributos do tipo `String`: um para armazenar a marca e outro para guardar o modelo do carro.

```

1 package model;
2
3 public class Carro {
4     private String marca;
5     private String modelo;
6
7     // GETTERS E SETTERS
8 }
```

Código Java 9.6: Carro.java

- 4 Na pasta `src`, faça um pacote chamado `managedbeans`.
- 5 No pacote `managedbeans`, crie uma classe chamada `CarrosBean` para armazenar uma lista de carros.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class CarrosBean {
5     private List<Carro> carros = new ArrayList<Carro>();
6
7     // GETTER E SETTER
8 }
```

Código Java 9.7: CarrosBean.java

- 6 Na classe `CarrosBean`, acrescente um atributo do tipo `Carro` para armazenar o carro a ser adicionado. Implemente também um método chamado `adicionaCarro()` para inserir um novo carro na lista.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class CarrosBean {
5
6     private List<Carro> carros = new ArrayList<Carro>();
7     private Carro carro = new Carro();
8
9     public void adicionaCarro() {
10         this.carros.add(this.carro);
11         this.carro = new Carro();
12     }
13
14     // GETTERS E SETTERS
15 }
```

Código Java 9.8: CarrosBean.java

- 7 Defina o escopo do managed bean carrosBean como sendo *request*. Para isso, use a anotação @RequestScoped.

```

1 package managedbeans;
2
3 @ManagedBean
4 @RequestScoped
5 public class CarrosBean {
6     ...
7 }
```

Código Java 9.9: CarrosBean.java

- 8 No diretório WebContent, crie um arquivo XHTML chamado carros.xhtml. Implemente um formulário para adicionar um novo carro.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8
9 <h:head>
10   <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13   <h:form>
14     <h:panelGrid columns="2">
15       <h:outputLabel value="Marca: " for="campo-marca" />
16       <h:inputText value="#{carrosBean.carro.marca}" id="campo-marca" />
17
18       <h:outputLabel value="Modelo: " for="campo-modelo" />
19       <h:inputText value="#{carrosBean.carro.modelo}" id="campo-modelo" />
20
21       <h:commandButton value="Adicionar" action="#{carrosBean.adicionaCarro}" />
22     </h:panelGrid>
23   </h:form>
24 </h:body>
25 </html>
```

Código XHTML 9.1: carro.xhtml

- 9 No arquivo carros.xhtml, adicione um trecho de código para exibir os carros adicionados.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8
9 <h:head>
10   <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13   <h:form>
14     <h:panelGrid columns="2">
15       <h:outputLabel value="Marca: " for="campo-marca" />
16       <h:inputText value="#{carrosBean.carro.marca}" id="campo-marca" />
17
18       <h:outputLabel value="Modelo: " for="campo-modelo" />
19       <h:inputText value="#{carrosBean.carro.modelo}" id="campo-modelo" />
```

```

21      <h:commandButton value="Adicionar" action="#{carrosBean.adicionaCarro}" />
22  </h:panelGrid>
23 </h:form>
24
25 <h:panelGroup rendered="#{not empty carrosBean.carros}">
26   <h1>Lista de carros:</h1>
27   <ul>
28     <ui:repeat value="#{carrosBean.carros}" var="carro">
29       <li><h:outputText value="#{carro.marca} #{carro.modelo}" /></li>
30     </ui:repeat>
31   </ul>
32 </h:panelGroup>
33 </h:body>
34 </html>

```

Código XHTML 9.2: carros.xhtml

- 10** Acesse a aplicação no endereço:

<http://localhost:8080/K19-Escopos/carros.xhtml>

Adicione alguns carros e observe a lista de carros. O que acontece?

- 11** Mude o escopo do managed bean carrosBean de *request* para *view*. Para isso, use a anotação `@ViewScoped`.

```

1 package managedbeans;
2
3 @ManagedBean
4 @ViewScoped
5 public class CarrosBean {
6   ...
7 }

```

Código Java 9.10: CarrosBean.java

- 12** Novamente, acesse a aplicação no endereço:

<http://localhost:8080/K19-Escopos/carros.xhtml>

Adicione alguns carros e observe a lista de carros. O que acontece?

- 13** No diretório WebContent, crie um arquivo chamado menu.xhtml. Esse arquivo deve definir uma página que possua um link para a página definida por carros.xhtml.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10 <h:body>
11   <h:link value="Adicionar carros" outcome="carros" />
12 </h:body>
13 </html>

```

Código XHTML 9.3: menu.xhtml

- 14** No arquivo carros.xhtml, adicione um link para a página definida por menu.xhtml.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8
9  <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13 <h:link value="Menu" outcome="menu"/>
14
15 <h:form>
16   <h:panelGrid columns="2">
17     <h:outputLabel value="Marca: " for="campo-marca" />
18     <h:inputText value="#{carrosBean.carro.marca}" id="campo-marca" />
19
20     <h:outputLabel value="Modelo: " for="campo-modelo" />
21     <h:inputText value="#{carrosBean.carro.modelo}" id="campo-modelo" />
22
23     <h:commandButton value="Adicionar" action="#{carrosBean.adicionaCarro}" />
24   </h:panelGrid>
25 </h:form>
26
27 <h:panelGroup rendered="#{not empty carrosBean.carros}">
28   <h1>Lista de carros:</h1>
29   <ul>
30     <ui:repeat value="#{carrosBean.carros}" var="carro">
31       <li><h:outputText value="#{carro.marca} #{carro.modelo}" /></li>
32     </ui:repeat>
33   </ul>
34 </h:panelGroup>
35 </h:body>
36 </html>
```

Código XHTML 9.4: carros.xhtml

- 15 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Escopos/carros.xhtml>

Certifique-se de que o escopo do managed bean carrosBean seja *view*. Mais uma vez, adicione alguns carros e observe a lista de carros. Use os links criados para navegar entre as páginas definidas por carros.xhtml e menu.xhtml. O que acontece?

- 16 Mude o escopo do managed bean carrosBean de *view* para *session*. Para isso, use a anotação @SessionScoped.

```

1 package managedbeans;
2
3 @ManagedBean
4 @SessionScoped
5 public class CarrosBean {
6   ...
7 }
```

Código Java 9.11: CarrosBean.java

- 17 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Escopos/carros.xhtml>

Novamente, adicione alguns carros e observe a lista de carros. Use os links que você criou para navegar entre as páginas definidas por `carros.xhtml` e `menu.xhtml`. O que acontece?

- 18 Agora, acesse a aplicação a partir de navegadores diferentes. Adicione alguns carros a partir dos diferentes navegadores. O que você observa?
- 19 Mude o escopo do managed bean `carrosBean` de `session` para `application`. Para isso, use a anotação `@ApplicationScoped`.

```
1 package managedbeans;  
2  
3 @ManagedBean  
4 @ApplicationScoped  
5 public class CarrosBean {  
6     ...  
7 }
```

Código Java 9.12: `CarrosBean.java`

- 20 Novamente, usando navegadores diferentes, adicione alguns carros. O que você pode observar?

CONVERSÃO E VALIDAÇÃO

Conversão

Quando um usuário preenche um formulário, os valores preenchidos são enviados para uma aplicação. De acordo com o HTTP, protocolo de comunicação utilizado entre os navegadores e as aplicações web, esses dados não possuem tipagem. Eles são tratados como texto puro. Dessa forma, quando uma aplicação recebe valores preenchidos em formulários HTML, ela precisa realizar a conversão dos dados que deseja tratar de forma específica.

Por exemplo, considere um formulário que possui um campo para os usuários digitarem a sua idade. A informação digitada nesse campo é tratada como texto puro até chegar na aplicação, que deve converter esse dado para algum tipo adequado do Java como `int` ou `long`.

Eventualmente, os dados que são enviados para a aplicação não podem ser convertidos, pois não estão no formato esperado. Por exemplo, se o texto preenchido em um campo numérico possui caracteres não numéricos, a conversão falhará.

Podemos observar o processo de conversão de outro ponto de vista. Nem sempre o formato das informações que estão em uma aplicação web Java corresponde ao formato que desejamos que seja apresentado aos usuários. Novamente, os dados devem ser convertidos antes de serem enviados para os navegadores.

Felizmente, o JSF oferece um mecanismo automatizado de conversão de dados. Veremos a seguir o funcionamento desse mecanismo.

Conversores Padrão

Para facilitar o trabalho de desenvolvimento de uma aplicação, o JSF define um conjunto de conversores padrão. Alguns desses conversores são aplicados automaticamente. Outros conversores são aplicados apenas se forem explicitamente indicados.

O JSF aplica automaticamente conversores padrão para os seguintes tipos fundamentais do Java:

- `BigDecimal`
- `BigInteger`
- `Boolean` e `boolean`
- `Byte` e `byte`
- `Character` e `char`
- `Double` e `double`

- Float e float
- Integer e int
- Long e long
- Short e short

No exemplo abaixo, o conteúdo digitado na caixa de texto será convertido para o tipo double automaticamente, pois a propriedade numero é do tipo double.

```

1 @ManagedBean
2 public class TesteBean {
3
4     private double numero;
5
6     // GETTERS E SETTERS
7 }
```

Código Java 10.1: TesteBean.java

```

1 <!-- O valor digitado nesse campo será convertido para double -->
2 <h:inputText value="#{testeBean.numero}" />
```

Código XHTML 10.1: Caixa de texto vinculada à propriedade numero

Personalizando Conversores Padrão

Alguns conversores padrão podem ou precisam usar informações adicionais para realizar a conversão de dados. As tags <f:convertNumber> e <f:convertDateTime> são usadas para transmitir informações aos conversores de números (java.lang.Number) e datas (java.util.Date e java.sql.Date), respectivamente.

<f:convertNumber>

A tag <f:convertNumber> permite que conversões mais sofisticadas sejam feitas em valores numéricos.

Para estipular que um valor numérico seja exibido com um número máximo de casas decimais, podemos usar o atributo maxFractionDigits da tag <f:convertNumber>. No exemplo abaixo, a propriedade numero do managed bean testeBean é exibida com no máximo duas casas decimais.

```

1 <h:outputText value="#{testeBean.numero}" >
2     <f:convertNumber maxFractionDigits="2" />
3 </h:outputText>
```

Analogamente, podemos definir o número mínimo de casas decimais com o atributo minFractionDigits.

Podemos também definir a formatação de um número por meio de uma espécie de expressão regular. Isso pode ser feito usando o atributo pattern, como no exemplo abaixo.

```

1 <h:outputText value="#{testeBean.numero}" >
2     <f:convertNumber pattern="#0.000" />
```

```
3 </h:outputText>
```

Para apresentar dados na forma de porcentagem, podemos usar o atributo type e defini-lo com o valor percent.

```
1 <h:outputText value="#{testeBean.numero}" >
2   <f:convertNumber type="percent" />
3 </h:outputText>
```

Para personalizar a exibição de valores monetários, a tag <f:convertNumber> provê o atributo currencySymbol e locale. No código abaixo, exemplificamos o uso desses dois atributos.

```
1 <h:outputText value="#{testeBean.numero}" >
2   <f:convertNumber currencySymbol="R$" type="currency" />
3 </h:outputText>
4
5 <h:outputText value="#{testeBean.numero}" >
6   <f:convertNumber locale="pt_BR" type="currency" />
7 </h:outputText>
```

<f:convertDateTime>

A tag <f:convertDateTime> permite que conversões de datas sejam realizadas. Esse conversor pode ser aplicado em dados do tipo `java.util.Date` e `java.sql.Date`. Essa tag possui o atributo pattern, que permite a definição do formato da data que desejamos utilizar.

```
1 <h:outputText value="#{testeBean.data}">
2   <f:convertDateTime pattern="dd/MM/yyyy" />
3 </h:outputText>
```

Para mais detalhes sobre o uso do atributo pattern, consulte a página <http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html>.

Uma outra maneira de escolher o formato de data a ser utilizado é por meio do atributo locale. No exemplo abaixo, ao definir o valor do atributo locale como pt_BR, o padrão de data utilizado será "dd/MM/yyyy".

```
1 <h:outputText value="#{testeBean.data}">
2   <f:convertDateTime locale="pt_BR" />
3 </h:outputText>
```



Exercícios de Fixação

- 1 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Conversao-e-Validacao** seguindo os passos vistos no exercício do Capítulo 5.
- 2 No diretório src, crie um pacote chamado model.
- 3 Crie uma classe chamada Funcionario para modelar funcionários e adicione-a ao pacote model.

```

1 package model;
2
3 public class Funcionario {
4     private Double salario;
5     private Integer codigo;
6     private Date aniversario;
7
8     // GETTERS E SETTERS
9 }
```

Código Java 10.2: Funcionario.java

- 4** Acrescente um pacote na pasta src chamado managedbeans e adicione a seguinte classe a esse pacote:

```

1 package managedbeans;
2
3 @ManagedBean
4 public class FuncionarioBean {
5     private Funcionario funcionario = new Funcionario();
6
7     // GETTER E SETTER
8 }
```

Código Java 10.3: FuncionarioBean.java

- 5** Crie uma tela para cadastrar funcionários. Adicione um arquivo chamado cadastro.xhtml na pasta WebContent.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://java.sun.com/jsf/html"
6       xmlns:f="http://java.sun.com/jsf/core">
7 <h:head>
8     <title>K19 Treinamentos</title>
9 </h:head>
10
11 <h:body>
12     <h1>Cadastro de Funcionário</h1>
13     <h:form>
14         <h:panelGrid columns="3">
15             <h:outputLabel value="Salário: R$ " for="campo-salario"/>
16             <h:inputText id="campo-salario" value="#{funcionarioBean.funcionario.salario}">
17                 <f:convertNumber locale="pt_BR"/>
18             </h:inputText>
19             <h:message for="campo-salario"/>
20
21             <h:outputLabel value="Código: " for="campo-codigo"/>
22             <h:inputText id="campo-codigo" value="#{funcionarioBean.funcionario.codigo}">
23             <h:message for="campo-codigo"/>
24
25             <h:outputLabel value="Data: (dd/MM/yyyy)" for="campo-aniversario"/>
26             <h:inputText id="campo-aniversario"
27                 value="#{funcionarioBean.funcionario.aniversario}">
28                 <f:convertDateTime pattern="dd/MM/yyyy"/>
29             </h:inputText>
30             <h:message for="campo-aniversario"/>
31
32             <h:commandButton value="Cadastrar"/>
33         </h:panelGrid>
34         <h:messages/>
35     </h:form>
36 </h:body>
```

37 `</html>`

Código XHTML 10.8: cadastro.xhtml

- 6 Acesse a aplicação no endereço:

`http://localhost:8080/K19-Conversao-e-Validacao/cadastro.xhtml`

Preencha o formulário com valores inadequados diversas vezes e observe as mensagens de erros.



Exercícios Complementares

- 1 Crie um formulário para registrar a cotação do dólar. Esse formulário deve ter três campos obrigatórios. No primeiro, o usuário deve digitar o valor do dólar em reais. Esse campo deve estar associado a um conversor de números cujo tipo seja a moeda brasileira (use os atributos locale e type da tag `<f:convertNumber>`). O segundo campo deve guardar a variação do dólar, em porcentagem, com no máximo cinco dígitos na parte fracionária. No terceiro campo, o usuário deve colocar o horário e a data da cotação, cujo formato deve ser “HH:mm dd-MM-yyyy”.
- 2 Exiba na tela os valores que foram submetidos pelo formulário e convertidos.

Mensagens de Erro

Eventualmente, as informações preenchidas pelos usuários em formulários não são adequadas, impedindo a conversão dos dados. Nesses casos, geralmente, desejamos apresentar mensagens relacionadas aos erros no preenchimento das informações.

<h:message>

Para exibir erros relacionados a um determinado campo, podemos utilizar o componente `<h:message>`. Esse componente deve ser associado ao campo correspondente aos erros que desejamos exibir. Para estabelecer essa ligação, devemos definir o valor do atributo `for` do componente `<h:message>` igual ao valor do atributo `id` do campo em questão.

```
1 @ManagedBean
2 public class TesteBean {
3
4     private Double numero;
5
6     // GETTERS E SETTERS
7 }
```

Código Java 10.6: TesteBean.java

```
1 <h:inputText value="#{testeBean.numero}" id="campo-numero" />
2 <h:message for="campo-numero" />
```

Código XHTML 10.11: Exibindo mensagens de erro relacionadas a um campo de texto

As imagens abaixo exibem o campo de texto com o valor digitado pelo usuário e após o processamento da requisição.

Figura 10.1: Campo de texto associado a uma propriedade numérica de um managed bean

Figura 10.2: Mensagem de erro de conversão



Mais Sobre

Os textos padrão das mensagens de erro são definidos no *resource bundle* javax.faces.Messages.



Mais Sobre

Cada mensagem de erro possui duas versões: uma detalhada e outra resumida. Por padrão, apenas a mensagem detalhada é exibida quando a tag <h:message> é aplicada. Para modificar esse comportamento, podemos alterar os valores dos atributos showSummary e showDetail. O primeiro atributo indica se a mensagem resumida deve ser apresentada, enquanto o segundo indica se a mensagem detalhada deve ser exibida. Esses atributos devem assumir valores true ou false. Veja o exemplo abaixo.

```
1 <h:inputText value="#{testeBean.numero}" id="campo-numero"/>
2 <h:message for="campo-numero" showSummary="true" showDetail="false"/>
```

Código XHTML 10.12: Definindo o tipo de mensagem de erro a ser apresentada

Note que as mensagens detalhada e resumida podem ser exibidas ao mesmo tempo. Para isso, basta que esses dois atributos possuam o valor true.

<h:messages>

Como vimos, a tag <h:message> permite que os erros relacionados a um determinado campo sejam exibidos. Para exibir os erros de todos os campos, podemos incluir uma tag <h:message> para cada campo. Entretanto, podemos fazer o mesmo de uma forma mais prática. A tag <h:messages> exibe todas as mensagens.

```
1 <h:messages />
```

Código XHTML 10.13: Exibindo todas as mensagens

Ao contrário da tag <h:message>, os valores padrão dos atributos showSummary e showDetail da tag <h:messages> são true e false, respectivamente.

Alterando as Mensagens de Erro

O texto de cada mensagem de erro de conversão ou validação está definido na especificação do JSF 2 que pode ser obtida através da url <http://jcp.org/en/jsr/detail?id=314>. Normalmente, queremos personalizar essas mensagens. Para isso, são necessários dois passos:

1. Criar um arquivo de mensagens.
2. Registrar esse arquivo no faces-config.xml.

Criando um Arquivo de Mensagens

Um arquivo de mensagens é um conjunto de chaves e valores. As chaves são utilizadas para recuperar as mensagens, enquanto os valores definem os textos que serão exibidos.

O maior problema para definir um arquivo de mensagens no JSF é saber quais são as chaves que podemos utilizar. Para conhecer as chaves, devemos consultar a especificação do JSF que pode ser obtida através da url <http://jcp.org/en/jsr/detail?id=314>. Veja um exemplo de arquivo de mensagens:

```
1 javax.faces.converter.BooleanConverter.BOLEAN={1}: ''{0}'' must be 'true' or 'false'.
```

Código XML 10.1: Exemplo de arquivo de mensagens

Os nomes dos arquivos de mensagens devem possuir o sufixo .properties.

Registrando um Arquivo de Mensagens

Suponha que você tenha criado um arquivo de mensagens chamado Messages.properties num pacote chamado resources. Para registrá-lo, você deve acrescentar uma configuração no arquivo faces-config.xml.

```
1 ...
2 <application>
3   <message-bundle>resources.Messages</message-bundle>
4 </application>
5 ...
```

Código XML 10.2: faces-config.xml

Note que o sufixo .properties é omitido no faces-config.xml.



Mais Sobre

Suponha que uma aplicação tenha um arquivo de mensagens registrado no

`faces-config.xml`. Para obter uma mensagem de erro, o JSF buscará primeiramente nesse arquivo. Caso não a encontre, ele utilizará a mensagem padrão.



Mais Sobre

Podemos definir, de maneira específica, a mensagem que deve ser utilizada quando um erro de conversão ocorrer em um determinado campo. Para isso, basta utilizar o atributo `converterMessage`. Observe o seguinte exemplo.

```
1 <h:inputText value="#{testeBean.numero}" id="campo-numero"
2   converterMessage="Por favor, digite um número"/>
3 <h:message for="campo-numero"/>
```

Código XHTML 10.14: Definindo uma mensagem para um erro de conversão

No exemplo acima, se o usuário submeter o formulário com um valor não numérico no campo `campo-numero`, então a mensagem de erro “Por favor, digite um número” será exibida na tela de resposta.



Exercícios de Fixação

- 7 No projeto K19-Conversao-e-Validacao, faça um pacote chamado `resources` na pasta `src`. Adicione a esse pacote um arquivo de mensagens chamado `Messages.properties` com o seguinte conteúdo.

```
1 javax.faces.converter.NumberConverter.NUMBER=0 valor {0} não é adequado.
2 javax.faces.converter.NumberConverter.NUMBER_detail={0} não é número ou é inadequado.
3 javax.faces.converter.IntegerConverter.INTEGER=0 valor {0} não é adequado.
4 javax.faces.converter.IntegerConverter.INTEGER_detail={0} não é um número inteiro.
5 javax.faces.converter.DateTimeConverter.DATE=A data {0} não está correta.
6 javax.faces.converter.DateTimeConverter.DATE_detail= {0} não parece uma data.
```

Código XML 10.3: Messages.properties

- 8 Adicione a configuração necessária no `faces-config.xml` para utilizar o arquivo de mensagens criado no exercício anterior.

```
1 ...
2 <application>
3   <message-bundle>resources.Messages</message-bundle>
4 </application>
5 ...
```

Código XML 10.4: faces-config.xml

Observação: a tag `<application>` deve ser colocada dentro da tag `<faces-config>`.

- 9 Preencha o formulário várias vezes com valores inadequados e observe as novas mensagens.

Validação

Como vimos, um dado enviado através de uma requisição HTTP chega na aplicação no formato de texto. O processo de conversão consiste em transformar esse texto em algum tipo específico do Java. Após a conversão, podemos verificar se os valores obtidos respeitam determinadas restrições impostas pelas regras da aplicação.

Por exemplo, considere um campo para o usuário digitar sua idade. Como discutido anteriormente, o valor digitado será tratado como texto até chegar na aplicação. Em seguida, esse dado será convertido para um tipo numérico do Java (como `int` ou `long` por exemplo).

Obviamente, a idade de um usuário deve ser um valor não-negativo. Contudo, essa condição não é verificada na etapa de conversão. Dessa forma, depois da conversão dos dados, mais uma etapa é necessária para que essa verificação seja realizada. Essa etapa é conhecida como validação.

Validadores Padrão

Novamente, para facilitar o trabalho de desenvolvimento de uma aplicação, o JSF define um conjunto de validadores padrão. A seguir, veremos alguns detalhes sobre esses validadores.

Campo Obrigatório (Required)

A validação mais comum de todas é a de verificar se um determinado campo foi preenchido. Podemos aplicar essa validação utilizando o atributo `required`.

```
1 <h:inputText value="#{testeBean.nome}" id="campo-nome" required="true" />
2 <h:message for="campo-nome" />
```

Código XHTML 10.15: Campo de texto obrigatório

Uma outra forma de registrar um validador desse tipo é por meio da tag `<f:validateRequired>`, conforme o exemplo abaixo.

```
1 <h:inputText value="#{testeBean.nome}" id="campo-nome">
2   <f:validateRequired/>
3 </h:inputText>
4 <h:message for="campo-nome" />
```

Código XHTML 10.16: Outra forma de exigir que um campo seja obrigatório

<f:validateLongRange>

O validador `<f:validateLongRange>` é utilizado para verificar se um valor numérico inteiro pertence a um determinado intervalo de números. No exemplo abaixo, o validador irá verificar se o valor da propriedade `idade` está entre 10 e 130.

```
1 <h:inputText value="#{testeBean.idade}" id="campo-idade">
2   <f:validateLongRange minimum="10" maximum="130" />
3 </h:inputText>
4 <h:message for="campo-idade" />
```

Código XHTML 10.17: A propriedade `idade` deve receber um número entre 10 e 130

<f:validateDoubleRange>

O validador `<f:validateDoubleRange>` funciona de forma análoga ao `<f:validateLongRange>`. Ele é utilizado para verificar se um valor numérico real pertence a um determinado intervalo de números. No exemplo abaixo, o validador verificará se o valor da propriedade `preco` do managed bean `testeBean` está entre 20.57 e 200.56.

```
1 <h:inputText value="#{testeBean.preco}" id="campo-preco">
2   <f:validateDoubleRange minimum="20.57" maximum="200.56" />
3 </h:inputText>
4 <h:message for="campo-preco" />
```

Código XHTML 10.18: A propriedade `preco` deve receber um número entre 20.57 e 200.56

<f:validateLength>

O validador `<f:validateLength>` verifica se uma string possui uma quantidade mínima ou máxima de caracteres. Veja o exemplo abaixo.

```
1 <h:inputText value="#{testeBean.nome}" id="campo-nome">
2   <f:validateLength minimum = "6" maximum = "20"/>
3 </h:inputText>
4 <h:message for="campo-nome" />
```

Código XHTML 10.19: Restrição no número de caracteres da propriedade `nome`

Nesse exemplo, a restrição imposta é a de que a propriedade `nome` do managed bean `testeBean` seja uma string com pelo menos 6 e no máximo 20 caracteres.

<f:validateRegex>

O validador `<f:validateRegex>` é usado para verificar se um texto respeita uma determinada expressão regular. No exemplo abaixo, o validador verificará se a propriedade `codigo` tem pelo menos 6 e no máximo 20 caracteres, além de ser formada apenas por letras.

```
1 <h:inputText value="#{testeBean.codigo}" id="campo-codigo">
2   <f:validateRegex pattern="[a-zA-Z]{6,20}" />
3 </h:inputText>
4 <h:message for="campo-codigo" />
```

Código XHTML 10.20: Expressão regular para restringir o dado de entrada de um campo de texto



Exercícios de Fixação

- 10** Vamos acrescentar alguns validadores para os campos do formulário de cadastro de funcionários do projeto K19-Conversao-e-Validacao. Com base nas restrições abaixo, acrescente validadores apropriados.

- O salário de um funcionário deve ser não-negativo.
- O código de um funcionário deve ser um número inteiro maior ou igual a 5 e estritamente menor do que 20.
- Todos os campos são obrigatórios.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7  <h:head>
8   <title>K19 Treinamentos</title>
9  </h:head>
10
11 <h:body>
12   <h1>Cadastro de Funcionário</h1>
13   <h:form>
14     <h:panelGrid columns="3">
15       <h:outputLabel value="Salário: R$ " for="campo-salario"/>
16       <h:inputText id="campo-salario" value="#{funcionarioBean.funcionario.salario}"
17         required="true">
18         <f:convertNumber locale="pt_BR"/>
19         <f:validateDoubleRange minimum="0"/>
20       </h:inputText>
21       <h:message for="campo-salario"/>
22
23       <h:outputLabel value="Código: " for="campo-codigo"/>
24       <h:inputText id="campo-codigo" value="#{funcionarioBean.funcionario.codigo}"
25         required="true">
26         <f:validateLongRange minimum="5" maximum="19"/>
27       </h:inputText>
28       <h:message for="campo-codigo"/>
29
30       <h:outputLabel value="Data: (dd/MM/yyyy)" for="campo-aniversario"/>
31       <h:inputText id="campo-aniversario"
32         value="#{funcionarioBean.funcionario.aniversario}"
33         required="true">
34         <f:convertDateTime pattern="dd/MM/yyyy"/>
35       </h:inputText>
36       <h:message for="campo-aniversario"/>
37
38       <h:commandButton value="Cadastrar"/>
39     </h:panelGrid>
40     <h:messages/>
41   </h:form>
42 </h:body>
43 </html>
```

Código XHTML 10.21: *cadastro.xhtml*

- 11** Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/cadastro.xhtml>

Teste o formulário com diversos valores e observe o que acontece.



Exercícios Complementares

- 3 Crie um formulário para o cadastro de produtos. Um produto tem quatro atributos: nome, código, tamanho e volume. O nome deve ser formado apenas por letras e espaços. O código deve ter pelo menos quatro e no máximo dez caracteres. O primeiro caractere desse código deve ser uma letra maiúscula e os demais devem ser números. O tamanho deve ser um número inteiro entre 1 e 32. O volume deve ser um número real maior ou igual a 0.1. Todos os campos do formulário devem ser obrigatórios.

Bean Validation

Uma nova abordagem para definir validações foi adicionada no JSF 2. A ideia é declarar as regras de validação nas classes do modelo ao invés de inseri-las nos arquivos XHTML que definem as telas. A grande vantagem das validações definidas nas classes do modelo é que elas podem ser utilizadas em diversas partes da aplicação. Esse novo recurso é chamado **Bean Validation** e foi definido pela especificação JSR 303, que pode ser obtida através da url <http://jcp.org/en/jsr/detail?id=303>.

Para definir as validações com bean validation, basta adicionar anotações nas classes do modelo. No código abaixo, o atributo nome foi marcado com a anotação @NotNull. Com essa anotação, o JSF verificará se o valor do atributo nome não é nulo.

```

1 public class Funcionario {
2     @NotNull
3     private String nome;
4     ...
5 }
```

Código Java 10.9: Funcionario.java



Mais Sobre

Considere uma caixa de texto vinculada a uma propriedade do tipo `String`. Por padrão, quando essa caixa não é preenchida, o JSF considera o seu conteúdo como sendo a string vazia ("") e não como o valor `null`. Assim, a anotação `@NotNull` não pode ser utilizada para obrigar que o usuário preencha essa caixa de texto.

Podemos alterar o comportamento padrão do JSF fazendo com que ele considere o conteúdo de uma caixa de texto não preenchida como `null`. Para isso, devemos acrescentar o seguinte trecho no arquivo de configuração da aplicação web (`web.xml`).

```

1 ...
2 <context-param>
3     <param-name>
4         javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
5     </param-name>
6     <param-value>true</param-value>
7 </context-param>
8 ...
```

Código XML 10.5: web.xml

Veja abaixo as anotações disponíveis e suas respectivas funções.

- **@AssertFalse**
Verifica se uma propriedade booleana possui valor false.
- **@AssertTrue**
Verifica se uma propriedade booleana possui valor true.
- **@DecimalMax**
Define o valor real máximo que uma propriedade pode armazenar.
- **@DecimalMin**
Define o valor real mínimo que uma propriedade pode assumir.
- **@Digits**
Define a quantidade máxima de dígitos da parte inteira (através do atributo `integer`) ou da parte fracionária (através do atributo `fraction`) de um número.
- **@Future**
Verifica se uma data é posterior ao instante atual.
- **@Max**
Define o valor inteiro máximo que uma propriedade pode assumir.
- **@Min**
Define o valor inteiro mínimo que uma propriedade pode armazenar.
- **@NotNull**
Verifica se o valor de uma propriedade não é null.
- **@Null**
Verifica se o valor de uma propriedade é null.
- **@Past**
Verifica se uma data é anterior ao instante atual.
- **@Pattern**
Verifica se o valor de uma propriedade respeita uma expressão regular.
- **@Size**
Define os tamanhos mínimo (através do atributo `min`) e máximo (através do atributo `max`) para uma Collection, array ou String.

As validações definidas através das anotações da especificação bean validation são aplicadas automaticamente pelo JSF. As mensagens de erro referentes a essas validações são automaticamente acrescentadas no contexto do processamento da requisição correspondente.



Mais Sobre

Para desabilitar bean validations em um componente particular de um formulário, podemos usar a tag `<f:validateBean>`. Isso pode ser feito atribuindo o valor “true” ao atributo `disabled` dessa tag. Considere o seguinte managed bean.

```
1 public class FuncionarioBean {  
2     private Funcionario funcionario = new Funcionario();  
3     // GETTER E SETTER  
4 }
```

Código Java 10.10: FuncionarioBean.java

No exemplo abaixo, a validação introduzida pela anotação `@NotNull` do atributo `nome` da classe `Funcionario` é desabilitada.

```
1 <h:inputText value="#{funcionarioBean.funcionario.nome}">
2   <f:validateBean disabled="true" />
3 </h:inputText>
```

Código XHTML 10.23: Desabilitando validações introduzidas por anotações

Personalizando as mensagens de erro

Podemos personalizar as mensagens de erro através do atributo `message` das anotações `validation`. Veja o seguinte código.

```
1 public class Funcionario {
2   @NotNull(message="O nome não pode ser nulo")
3   private String nome;
4   ...
5 }
```

Código Java 10.11: Funcionario.java

No exemplo acima, a mensagem de erro de validação está definida de maneira fixa. Pode ser mais apropriado defini-la em um arquivo de mensagens. Nesse caso, devemos criar um arquivo chamado `ValidationMessages.properties` no classpath da aplicação. No exemplo abaixo, a mensagem “O nome do funcionário não pode ser nulo” está associada à chave `br.com.k19.Funcionario.nome`.

```
1 br.com.k19.Funcionario.nome=O nome do funcionário não pode ser nulo
```

Arquivo de Propriedades 10.1: ValidationMessages.properties

Finalmente, devemos definir o valor do atributo `message` com a chave associada à mensagem que desejamos exibir. O identificador da mensagem deve estar entre chaves, como mostrado a seguir.

```
1 public class Funcionario {
2   @NotNull(message="{br.com.k19.Funcionario.nome}")
3   private String nome;
4   ...
5 }
```

Código Java 10.12: Funcionario.java

A mensagem “O nome do funcionário não pode ser nulo” é específica para o caso acima. Agora, suponha que desejamos criar uma mensagem comum associada ao validador `@NotNull`. Como o valor padrão da propriedade `message` desse validador é `"{javax.validation.constraints.NotNull.message}"`, basta definir uma mensagem com essa chave no arquivo `ValidationMessages.properties`.

```
1 javax.validation.constraints.NotNull.message=Valor não pode ser nulo
```

Arquivo de Propriedades 10.2: ValidationMessages.properties

Podemos também alterar as mensagens dos demais validadores do bean validation. As chaves das mensagens associadas a esses validadores são listadas abaixo:

- @AssertFalse: "{javax.validation.constraints.AssertFalse.message}"
- @AssertTrue: "{javax.validation.constraints.AssertTrue.message}"
- @DecimalMax: "{javax.validation.constraints.DecimalMax.message}"
- @DecimalMin: "{javax.validation.constraints.DecimalMin.message}"
- @Digits: "{javax.validation.constraints.Digits.message}"
- @Future: "{javax.validation.constraints.Future.message}"
- @Max: "{javax.validation.constraints.Max.message}"
- @Min: "{javax.validation.constraints.Min.message}"
- @Null: "{javax.validation.constraints.Null.message}"
- @Past: "{javax.validation.constraints.Past.message}"
- @Pattern: "{javax.validation.constraints.Pattern.message}"
- @Size: "{javax.validation.constraints.Size.message}"



Exercícios de Fixação

- 12** Remova os validadores do arquivo `cadastro.xhtml` que você adicionou no exercício anterior.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7  <h:head>
8   <title>K19 Treinamentos</title>
9  </h:head>
10
11 <h:body>
12  <h1>Cadastro de Funcionário</h1>
13  <h:form>
14   <h:panelGrid columns="3">
15    <h:outputLabel value="Salário: R$ " for="campo-salario"/>
16    <h:inputText id="campo-salario" value="#{funcionarioBean.funcionario.salario}">
17     <f:convertNumber locale="pt_BR"/>
18    </h:inputText>
19    <h:message for="campo-salario"/>
20
21    <h:outputLabel value="Código: " for="campo-codigo"/>
22    <h:inputText id="campo-codigo" value="#{funcionarioBean.funcionario.codigo}">
23    <h:message for="campo-codigo"/>
24
25    <h:outputLabel value="Data: (dd/MM/yyyy)" for="campo-aniversario"/>
26    <h:inputText id="campo-aniversario"
27      value="#{funcionarioBean.funcionario.aniversario}">
28      <f:convertDateTime pattern="dd/MM/yyyy"/>
29    </h:inputText>
30    <h:message for="campo-aniversario"/>
```

```

31      <h:commandButton value="Cadastrar"/>
32  </h:panelGrid>
33  <h:messages/>
34  </h:form>
35 </h:body>
36 </html>
37

```

Código XHTML 10.24: cadastro.xhtml

- 13** Usando bean validation, modifique a classe Funcionario para considerar as restrições do exercício anterior, isto é:

- O salário de um funcionário deve ser não-negativo.
- O código de um funcionário deve ser um número inteiro maior ou igual a 5 e estritamente menor do que 20.
- Todos os campos são obrigatórios.

```

1 package model;
2
3 public class Funcionario {
4
5     @NotNull
6     @Min(value = 0)
7     private Double salario;
8
9     @NotNull
10    @Min(value = 5)
11    @Max(value = 19)
12    private Integer codigo;
13
14    @NotNull
15    private Date aniversario;
16
17    // GETTERS E SETTERS
18 }

```

Código Java 10.13: Funcionario.java

- 14** Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/cadastro.xhtml>

Teste o formulário com diversos valores e observe o que acontece.

Criando o seu Próprio Conversor

Se os conversores padrão não forem suficientes para atender as necessidades de uma aplicação, podemos criar os nossos próprios conversores. Considere uma aplicação que armazena números de telefones internacionais. Esses números serão modelados pela seguinte classe.

```

1 public class Telefone {
2     private String codigoDoPais;
3     private String codigoDeArea;
4     private String numeroLocal;
5
6     // GETTERS E SETTERS
7 }

```

Código Java 10.14: Telefone.java

Um número de telefone é dividido em três partes: o código do país, o código da área e o número local. A interface da nossa aplicação utilizará o seguinte formato para apresentar ou receber números de telefone:

```
CodigoDoPais CodigoDeArea NumeroLocal
```

As três partes são separadas por espaços. Tanto o código do país quanto o código de área são formados apenas por dígitos. O número local, por outro lado, é formado por dígitos e, opcionalmente, por hifens. No exemplo abaixo, o número de telefone da K19 é mostrado nesse formato. Veja abaixo alguns exemplos de como o número de telefone da K19 poderia ser apresentado segundo esse formato.

- 55 11 2387-3791
- 55 11 23873791
- 55 11 23-87-37-91

Para automatizar o processo de transformação das strings digitadas pelos usuários em objetos da classe Telefone e vice-e-versa, podemos implementar um conversor JSF.

O primeiro passo para implementar um conversor JSF é criar uma classe que implementa a interface `javax.faces.convert.Converter`. Nessa classe, devemos adicionar a anotação `@FacesConverter` para indicar a classe associada a esse conversor, como no código abaixo.

```
1 @FacesConverter(forClass=Telefone.class)
2 public class ConversorDeTelefone implements Converter {
3     ...
4 }
```

Código Java 10.15: ConversorDeTelefone.java

A interface `Converter` exige a implementação de dois métodos: `getAsObject()` e `getAsString()`. O primeiro método deve transformar um objeto do tipo `String` em um objeto Java (em nosso caso, um objeto do tipo `Telefone`). Já o método `getAsString()` deve converter um objeto Java em um objeto do tipo `String` (em nosso caso, criar uma `String` a partir de um `Telefone`).

Uma possível implementação para o conversor de números de telefone é apresentada a seguir.

```
1 package br.com.k19;
2
3 @FacesConverter(forClass=Telefone.class)
4 public class ConversorDeTelefone implements Converter {
5
6     @Override
7     public Object getAsObject(FacesContext context, UIComponent component,
8         String value) {
9
10        value = value.trim();
11
12        if (!Pattern.matches("[0-9]+\\s+[0-9]+\\s+[0-9-]+", value)) {
13            FacesMessage mensagem = new FacesMessage("Número de telefone inválido");
14        }
15    }
16
17    public String getAsString(FacesContext context, UIComponent component,
18        Object value) {
19        return ((Telefone) value).getCodigoDoPais() + " "
20            + ((Telefone) value).getCodigoDeArea() + " "
21            + ((Telefone) value).getNumeroLocal();
22    }
23}
```

```

14     mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
15     throw new ConverterException(mensagem);
16 }
17
18 String campos[] = value.split("\\s+");
19
20 String codigoDoPais = campos[0];
21 String codigoDeArea = campos[1];
22 String numeroLocal = campos[2].replaceAll("-", " ");
23
24 Telefone telefone = new Telefone(codigoDoPais, codigoDeArea, numeroLocal);
25 return telefone;
26 }
27
28 @Override
29 public String getAsString(FacesContext context, UIComponent component,
30 Object value) {
31     Telefone telefone = (Telefone) value;
32     return telefone.getCodigoDoPais()
33         + " " + telefone.getCodigoDeArea()
34         + " " + telefone.getNumeroLocal();
35 }
36 }
```

Código Java 10.16: ConversorDeTelefone.java

Se a string recebida pelo método `getAsObject()` não respeita o formato de números de telefone definido acima, uma mensagem de erro é criada e uma exceção do tipo `javax.faces.convert.ConverterException` guardando essa mensagem é lançada.



Exercícios de Fixação

- 15** Considere um formulário que possua uma caixa de texto para receber um número de CPF (Cadastro de Pessoas Físicas). Um número de CPF é dividido em duas partes. A primeira parte do número é formada por nove dígitos. Chamaremos essa parte de número de identificação. A segunda parte é formada por dois dígitos, que são chamados de dígitos verificadores. Esses dois últimos dígitos são usados para verificar o número de identificação e prevenir erros. Um número de CPF é apresentado com a primeira parte separada da segunda por um hífen.

Implemente um conversor de números de CPF. Primeiramente, crie uma classe chamada `CPF` dentro do pacote `model` para representar um número de CPF. Essa classe deve ter três atributos do tipo `int`: um para armazenar o número de identificação e outros dois para guardar os dígitos verificadores.

```

1 package model;
2
3 public class CPF {
4     private int numeroDeIdentificacao;
5     private int primeiroDigitoVerificador;
6     private int segundoDigitoVerificador;
7
8     // GETTERS E SETTERS
9 }
```

Código Java 10.17: CPEjava

- 16** No diretório src, crie um pacote chamado converters.
- 17** Agora, dentro do pacote converters, crie uma classe chamada ConversorDeCPF para implementar o conversor de números de CPF. Um número de CPF deve respeitar o seguinte formato: XXXXXXXXX-XX. Lembre que essa classe deve ser anotada com @FacesConverter e implementar a interface javax.faces.convert.Converter.

```

1 package converters;
2
3 @FacesConverter(forClass=CPF.class)
4 public class ConversorDeCPF implements Converter {
5
6     @Override
7     public Object getAsObject(FacesContext context, UIComponent component,
8         String value) {
9
10        value = value.trim();
11
12        if (!Pattern.matches("[0-9]{9}-[0-9]{2}", value)) {
13            FacesMessage mensagem = new FacesMessage("Número de CPF inválido");
14            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
15            throw new ConverterException(mensagem);
16        }
17
18        String partesDoCPF[] = value.split("-");
19        int numeroDeIdentificacao = Integer.parseInt(partesDoCPF[0]);
20        int primeiroDigitoVerificador = Integer.parseInt(partesDoCPF[1].substring(0, 1));
21        int segundoDigitoVerificador = Integer.parseInt(partesDoCPF[1].substring(1, 2));
22
23        CPF cpf = new CPF();
24        cpf.setNumeroDeIdentificacao(numeroDeIdentificacao);
25        cpf.setPrimeiroDigitoVerificador(primeiroDigitoVerificador);
26        cpf.setSegundoDigitoVerificador(segundoDigitoVerificador);
27
28        return cpf;
29    }
30
31    @Override
32    public String getAsString(FacesContext context, UIComponent component,
33        Object value) {
34        CPF cpf = (CPF) value;
35        return cpf.getNumeroDeIdentificacao() + "-" + cpf.getPrimeiroDigitoVerificador()
36            + cpf.getSegundoDigitoVerificador();
37    }
38}
```

Código Java 10.18: ConversorDeCPF.java

- 18** No pacote managedbeans, crie uma classe chamada CPFBean para implementar o managed bean que dará suporte ao formulário.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class CPFBean {
5     private CPF cpf;
6
7     // GETTER E SETTER
8 }
```

Código Java 10.19: CPFBean.java

- 19** Finalmente, crie um arquivo chamado formulario.xhtml e implemente um formulário para receber um número de CPF.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html">
6  <h:head>
7   <title>K19 Treinamentos</title>
8  </h:head>
9
10 <h:body>
11  <h1>Formulário</h1>
12  <h:form>
13   <h:outputLabel value="CPF: " for="cpf"/>
14   <h:inputText id="cpf" value="#{cPFBean.cpf}" />
15   <h:commandButton value="Enviar"/>
16  </h:form>
17
18  <h:message for="cpf" style="color: red" />
19
20 </h:body>
</html>

```

Código XHTML 10.25: formulario.xhtml

- 20 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/formulario.xhtml>

Teste o formulário com diversos valores para campo de CPF. Observe as mensagens de erro de conversão para valores inválidos.



Exercícios Complementares

- 4 Em 2010, foi lançado um novo documento de identificação no Brasil, o Registro de Identidade Civil (RIC). Esse documento tem um número que identifica unicamente cada cidadão. O número RIC tem duas partes. A primeira parte é formada por dez dígitos e a segunda, o dígito verificador, é formada por um único dígito. A primeira parte é opcionalmente separada da segunda por um hífen. Primeiro, você deve criar uma classe para modelar esse documento. Em seguida, você deve implementar um conversor para esse tipo de objeto. Por fim, você deve implementar um formulário para testar o seu conversor.

Criando o seu Próprio Validador

A criação de um validador é análoga à criação de um conversor. Precisamos criar uma classe que implemente a interface `javax.faces.validator.Validator`. Essa interface define um único método, o `validate()`. Além disso, a classe também precisa ser anotada com `@FacesValidator`.

Considere um formulário HTML que possui uma caixa de texto para o usuário digitar um número primo. Um número primo é um número natural maior do que 1 cujos únicos divisores positivos são 1 e ele mesmo. Essa caixa de texto está associada a uma propriedade do tipo `long` em um managed bean.

Após o usuário submeter esse formulário e uma vez que a etapa de conversão foi concluída, precisamos validar os dados do formulário. Em particular, precisamos verificar se o número digitado

pelo usuário é primo. Criaremos uma classe chamada `ValidadorDeNumerosPrimos` para implementar um validador que execute essa tarefa.

```

1 package br.com.k19;
2
3 @FacesValidator("br.com.k19.ValidadorDeNumerosPrimos")
4 public class ValidadorDeNumerosPrimos implements Validator {
5     ...
6 }
```

Código Java 10.23: ValidadorDeNumerosPrimos.java

Uma forma simples de verificar se determinado número é primo é testar se ele é divisível por algum número maior do que 1 e menor ou igual à raiz quadrada desse número. Se ele for divisível por algum desses números, então ele não é primo. Caso contrário, ele é um número primo. No código abaixo, apresentamos uma implementação para esse validador.

```

1 package br.com.k19;
2
3 @FacesValidator("br.com.k19.ValidadorDeNumerosPrimos")
4 public class ValidadorDeNumerosPrimos implements Validator {
5
6     @Override
7     public void validate(FacesContext context, UIComponent component, Object value) {
8
9         long numero = (Long) value;
10        boolean isPrimo = false;
11
12        if (numero > 1) {
13            double raizQuadrada = Math.sqrt((double) numero);
14            isPrimo = true;
15
16            for (long i = 2; i <= raizQuadrada; i++) {
17                if (numero % i == 0) {
18                    isPrimo = false;
19                    break;
20                }
21            }
22        }
23
24        if (!isPrimo) {
25            FacesMessage mensagem = new FacesMessage("O número " + numero + " não é primo");
26            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
27            throw new ValidatorException(mensagem);
28        }
29    }
30 }
```

Código Java 10.24: ValidadorDeNumerosPrimos.java

Se o número que estamos validando não for primo então precisamos lançar uma exceção do tipo `javax.faces.validator.ValidatorException`. Analogamente ao que fizemos na elaboração de um conversor, lançamos uma exceção contendo uma `FacesMessage` com uma mensagem de erro.

Para associar um validador a um determinado campo de um formulário, podemos usar a tag `<f:validator>`. Essa tag possui um atributo chamado `validatorId` cujo valor deve ser igual ao valor registrado na anotação `@FacesValidator`. Em nosso caso particular, esse valor é "br.com.k19.ValidadorDeNumerosPrimos".

O código XHTML abaixo define um formulário com uma caixa de texto e vincula essa caixa ao validador de números primos.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <h:form>
13     <h:outputLabel value="Digite um número primo: " for="numero-primo" />
14     <h:inputText value="#{testeBean.numeroPrimo}" id="numero-primo">
15       <f:validator validatorId="br.com.k19.ValidadorDeNumerosPrimos" />
16     </h:inputText>
17     <h:commandButton value="Enviar" />
18     <br />
19     <h:message for="numero-primo" style="color: red" />
20   </h:form>
21 </h:body>
22 </html>

```

Código XHTML 10.27: Associando um validador a uma propriedade

Passando Parâmetros para o Validador

Para realizar a validação de um determinado dado, informações adicionais podem ser necessárias. Por exemplo, considere o caso em que precisamos verificar se uma data escolhida por um usuário pertence a um determinado intervalo. Nesse caso, precisamos de duas informações adicionais: as datas de início e fim do intervalo.

Da mesma forma que criamos um validador de números primos, podemos criar um validador para realizar essa verificação. A diferença é que agora precisamos das datas que determinam o intervalo. Por exemplo, se tivermos uma caixa de texto para receber a data escolhida pelo usuário, então as datas de início e fim do intervalo podem ser definidas como atributos dessa caixa.

Para definir um atributo de um componente visual, podemos usar a tag `<f:attribute>`. Essa tag possui dois parâmetros (`name` e `value`), que são usados para definir o nome e o valor desse atributo.

Em nosso exemplo, criaremos dois atributos, que chamaremos de `"inicio"` e `"fim"`, para definir as datas que delimitam o intervalo a ser considerado. No código abaixo, criamos esses atributos com valores `"01/05/2014"` e `"30/08/2014"`, respectivamente.

```

1 <h:inputText value="#{testeBean.data}" id="campo-data">
2   <f:attribute name="inicio" value="01/05/2014" />
3   <f:attribute name="fim" value="30/08/2014" />
4 </h:inputText>

```

Código XHTML 10.28: Adicionando atributos a um componente visual

Esses atributos podem ser recuperados no método `validate()` do validador da seguinte forma:

```

1 @Override
2 public void validate(FacesContext context, UIComponent component, Object value) {
3
4   String inicio = (String) component.getAttributes().get("inicio");
5   String fim = (String) component.getAttributes().get("fim");

```

```

6     ...
7 }
```

Código Java 10.25: Recuperando atributos de um componente visual



Exercícios de Fixação

- 21** Agora, crie um validador de números de CPF. Como dissemos no exercício anterior, os dois últimos dígitos do CPF (dígitos verificadores) são usados para validar os demais dígitos (número de identificação). Os dígitos verificadores são gerados por um algoritmo chamado *Módulo 11* a partir do número de identificação.

O primeiro dígito verificador é obtido da seguinte forma. Suponha que o número de identificação seja $x_9x_8x_7x_6x_5x_4x_3x_2x_1$, onde x_i representa o i -ésimo dígito da direita para a esquerda do número de identificação. Seja y o valor dado por

$$y = 2x_1 + 3x_2 + 4x_3 + 5x_4 + 6x_5 + 7x_6 + 8x_7 + 9x_8 + 10x_9.$$

Se o valor do resto da divisão de y por 11 ($y \% 11$ em Java) for menor do que 2, então o primeiro dígito verificador é $d_1 = 0$. Caso contrário, o primeiro dígito verificador d_1 é igual a 11 menos o resto da divisão de y por 11.

O segundo dígito verificador é obtido de forma análoga, mas aplicando o algoritmo acima ao número $x_9x_8x_7x_6x_5x_4x_3x_2x_1d_1$. Nesse caso, o novo valor de y será

$$y = 2d_1 + 3x_1 + 4x_2 + 5x_3 + 6x_4 + 7x_5 + 8x_6 + 9x_7 + 10x_8 + 11x_9.$$

Crie um pacote chamado `br.com.k19.validators` no diretório `src`.

- 22** No pacote `validators`, crie uma classe chamada `ValidadorDeCPF`. Lembre que essa classe deve ser anotada com `@FacesValidator` e implementar a interface `javax.faces.validator.Validator`.

```

1 package br.com.k19.validators;
2
3 @FacesValidator(value = "br.com.k19.validators.ValidadorDeCPF")
4 public class ValidadorDeCPF implements Validator {
5
6     @Override
7     public void validate(FacesContext context, UIComponent component,
8             Object value) throws ValidatorException {
9
10        CPF cpf = (CPF) value;
11        int numeroDeIdentificacao = cpf.getNumeroDeIdentificacao();
12        int primeiroDigitoVerificador = cpf.getPrimeiroDigitoVerificador();
13        int segundoDigitoVerificador = cpf.getSegundoDigitoVerificador();
14
15        if (!this.validaCPF(numeroDeIdentificacao, primeiroDigitoVerificador,
16                segundoDigitoVerificador)) {
17
18            String numero = numeroDeIdentificacao + "-"
19                    + primeiroDigitoVerificador + segundoDigitoVerificador;
20
21            FacesMessage mensagem = new FacesMessage("O número " + numero
22                    + " não é um CPF válido");
23            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
24            throw new ValidatorException(mensagem);
25        }
26    }
27}
```

```

26 }
27
28     private boolean validaCPF(int numeroDeIdentificacao,
29         int primeiroDigitoVerificador, int segundoDigitoVerificador) {
30
31         long primeiroDigito = this.modulo11((long) numeroDeIdentificacao);
32         long segundoDigito = this.modulo11((long) numeroDeIdentificacao * 10
33             + primeiroDigito);
34         return primeiroDigitoVerificador == primeiroDigito
35             && segundoDigitoVerificador == segundoDigito;
36     }
37
38     private long modulo11(long numero) {
39         long soma = 0;
40         long multiplicador = 2;
41         while (numero > 0) {
42             long digito = numero % 10;
43             soma += multiplicador * digito;
44             numero /= 10;
45             multiplicador++;
46         }
47         long resto = soma % 11;
48         if (resto < 2)
49             return 0;
50         else
51             return 11 - resto;
52     }
53 }
```

Código Java 10.26: ValidadorDeCPF.java

- 23** No arquivo `formulario.xhtml`, associe a caixa de texto ao validador de números de CPF criado.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7
8  <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11
12 <h:body>
13   <h1>Formulário</h1>
14   <h:form>
15     <h:outputLabel value="CPF: " for="cpf"/>
16     <h:inputText id="cpf" value="#{cPFBean.cpf}">
17       <f:validator validatorId="br.com.k19.validators.ValidadorDeCPF" />
18     </h:inputText>
19     <h:commandButton value="Enviar"/>
20   </h:form>
21   <h:message for="cpf" style="color: red" />
22 </h:body>
23 </html>
```

Código XHTML 10.29: formulario.xhtml

- 24** Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/formulario.xhtml>

Submeta o formulário com valores inválidos para o número de CPF e observe as mensagens de erro de validação.



Exercícios Complementares

- 5 Considere um formulário que possua um campo de texto para o usuário digitar uma data. Essa data deve pertencer a um determinado intervalo. Implemente um validador que considere essa restrição. As datas que definem o intervalo devem ser passadas como parâmetro para o validador com o uso da tag <f:attribute>.

Criando o seu Próprio Bean Validator

Vimos que os bean validators podem ser registrados através de anotações. Agora, mostraremos como criar o seu próprio bean validator e uma anotação para aplicá-lo. Para exemplificar, considere a validação de números primos. No exemplo abaixo, definimos a anotação @Primo.

```

1 package br.com.k19;
2
3 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
4 import static java.lang.annotation.ElementType.FIELD;
5 import static java.lang.annotation.ElementType.METHOD;
6 import static java.lang.annotation.RetentionPolicy.RUNTIME;
7
8 import java.lang.annotation.Retry;
9 import java.lang.annotation.Target;
10
11 import javax.validation.Constraint;
12 import javax.validation.Payload;
13
14 @Target({ METHOD, FIELD, ANNOTATION_TYPE })
15 @Retention(RUNTIME)
16 @Constraint(validatedBy = ValidadorDePrimo.class)
17 public @interface Primo {
18     String message() default "{br.com.k19.Primo.message}";
19     Class<?>[] groups() default {};
20     Class<? extends Payload>[] payload() default {};
21 }
```

Código Java 10.29: Primo.java

Note que a anotação @Primo é vinculada ao validador definido pela classe ValidadorDePrimo. Esse vínculo é estabelecido através da propriedade validatedBy da anotação @Constraint.

Observe também que, de acordo com o código acima, a mensagem definida pela chave br.com.-k19.Primo.message será utilizada por padrão quando um erro for identificado na validação.

A classe que define o validador de números primos deve implementar a interface javax.validation.ConstraintValidator. Essa interface possui dois métodos: initialize() e isValid(). O primeiro é utilizado para extrair, se necessário, os valores dos atributos da anotação. O segundo deve implementar a validação propriamente dita.

```

1 public class ValidadorDePrimo implements ConstraintValidator<Primo, Long> {
2
3     @Override
4     public void initialize(Primo constraintAnnotation) {
5
6     }
7
8     @Override
```

```

9  public boolean isValid(Long value, ConstraintValidatorContext context) {
10    if (value <= 1) {
11      return false;
12    }
13    double raizQuadrada = Math.sqrt((double) value);
14
15    for (long i = 2; i <= raizQuadrada; i++) {
16      if (value % i == 0) {
17        return false;
18      }
19    }
20    return true;
21  }
22 }
```

Código Java 10.30: ValidatorDePrimo.java

Por fim, devemos definir o valor da mensagem associada à chave `br.com.k19.Primo.message` no arquivo `ValidationMessages.properties`.

```
1 br.com.k19.Primo.message=0 número não é primo
```

Arquivo de Propriedades 10.3: ValidationMessages.properties



Exercícios de Fixação

25 Seguindo o exercício anterior, crie um bean validator para validar números de CPF. Primeiro, crie um pacote chamado `beanvalidators` no diretório `src`.

26 No pacote `beanvalidators`, defina a anotação `@CPF` como no código abaixo.

```

1 package beanvalidators;
2
3 import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
4 import static java.lang.annotation.ElementType.FIELD;
5 import static java.lang.annotation.ElementType.METHOD;
6 import static java.lang.annotation.RetentionPolicy.RUNTIME;
7
8 import java.lang.annotation.Retention;
9 import java.lang.annotation.Target;
10
11 import javax.validation.Constraint;
12 import javax.validation.Payload;
13
14 @Target({ METHOD, FIELD, ANNOTATION_TYPE })
15 @Retention(RUNTIME)
16 @Constraint(validatedBy = beanvalidators.CPFBeanValidator.class)
17 public @interface CPF {
18   String message() default "O número de CPF não é válido";
19   Class<?>[] groups() default {};
20   Class<? extends Payload>[] payload() default {};
21 }
```

Código Java 10.31: CPF.java

27 No pacote `beanvalidators`, crie uma classe chamada `CPFBeanValidator` para implementar o validador associado à anotação `@CPF`. Essa classe deve implementar a interface `javax.validation.ConstraintValidator`.

```

1 package beanvalidators;
2
3 import javax.validation.ConstraintValidator;
4 import javax.validation.ConstraintValidatorContext;
5
6 public class CPFBeanValidator implements
7     ConstraintValidator<beanvalidators.CPF, model.CPF> {
8
9     @Override
10    public void initialize(beanvalidators.CPF constraintAnnotation) {
11    }
12
13    @Override
14    public boolean isValid(model.CPF cpf, ConstraintValidatorContext context) {
15
16        int numeroDeIdentificacao = cpf.getNumeroDeIdentificacao();
17        int primeiroDigitoVerificador = cpf.getPrimeiroDigitoVerificador();
18        int segundoDigitoVerificador = cpf.getSegundoDigitoVerificador();
19
20        return this.validaCPF(numeroDeIdentificacao, primeiroDigitoVerificador,
21            segundoDigitoVerificador);
22    }
23
24    private boolean validaCPF(int numeroDeIdentificacao,
25        int primeiroDigitoVerificador, int segundoDigitoVerificador) {
26
27        long primeiroDigito = this.modulo11((long) numeroDeIdentificacao);
28        long segundoDigito = this.modulo11((long) numeroDeIdentificacao * 10
29            + primeiroDigito);
30        return primeiroDigitoVerificador == primeiroDigito
31            && segundoDigitoVerificador == segundoDigito;
32    }
33
34    private long modulo11(long numero) {
35        long soma = 0;
36        long multiplicador = 2;
37        while (numero > 0) {
38            long digito = numero % 10;
39            soma += multiplicador * digito;
40            numero /= 10;
41            multiplicador++;
42        }
43        long resto = soma % 11;
44        if (resto < 2)
45            return 0;
46        else
47            return 11 - resto;
48    }
49}

```

Código Java 10.32: CPFBeanValidator.java

- 28** Na classe CPFBean, adicione a anotação @CPF ao atributo cpf para usar o validador que você acabou de criar.

```

1 package managedbeans;
2
3 import javax.faces.bean.ManagedBean;
4 import model.CPF;
5
6 @ManagedBean
7 public class CPFBean {
8
9     @beanvalidators.CPF
10    private CPF cpf;
11
12    // GETTER E SETTER
13}

```

Código Java 10.33: CPFBean.java

- 29 Modifique no arquivo formulario.xhtml a linha em destaque.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10
11 <h:body>
12   <h1>Formulário</h1>
13   <h:form>
14     <h:outputLabel value="CPF: " for="cpf"/>
15     <h:inputText id="cpf" value="#{cPFBean.cpf}" />
16     <h:commandButton value="Enviar"/>
17   </h:form>
18   <h:message for="cpf" style="color: red" />
19 </h:body>
20 </html>
```

Código XHTML 10.32: formulario.xhtml

- 30 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/formulario.xhtml>

EVENTOS

Aplicações JSF são fortemente baseadas em eventos. Esses eventos podem ser gerados pelos usuários da aplicação ou pelo próprio JSF. Veja abaixo alguns exemplos de eventos gerados pelos usuários.

- Um clique em um botão ou link.
- A troca do valor preenchido em uma caixa de texto.
- A troca da opção escolhida em uma caixa de seleção.

Veja abaixo alguns exemplos de eventos gerados pelo JSF.

- Inicialização da aplicação.
- Finalização da aplicação.
- Erro no processamento de uma requisição.

Um evento pode ser associado a procedimentos que serão executados quando esse evento ocorrer. No JSF, os eventos são divididos em três categorias: FacesEvent, PhaseEvent e SystemEvent.

FacesEvent

Há dois tipos de FacesEvents: ActionEvent e ValueChangeEvent. Apresentaremos a seguir o funcionamento desses tipos de eventos.

ActionEvent

Um ActionEvent é gerado quando um botão ou link é pressionado pelo usuário. Métodos de um managed bean podem ser associados a esses eventos. Toda vez que um evento ocorrer os métodos associados a ele serão executados. Para associar métodos de um managed bean a um ActionEvent, podemos utilizar os atributos action ou actionListener dos componentes `<h:commandButton>` e `<h:commandLink>`.

O atributo action deve ser associado a um método público do tipo void ou String de um managed bean. Se esse método devolver uma String, ela será utilizada para determinar a navegação das páginas (veja Capítulo 8). Conceitualmente, o atributo action deve ser associado a métodos que representam alguma regra de negócio da aplicação.

No exemplo abaixo, o botão “Salva” está associado ao método `salva()` do managed bean `produtoBean` através do atributo action da tag `<h:commandButton>`. Esse método implementa a lógica de negócio para cadastrar produtos. A String “lista-produtos” devolvida pelo método `salva()` será utilizada para determinar a navegação das páginas.

```
1 <h:commandButton value="Salva" action="#{produtoBean.salva}" />
```

Código XHTML 11.1: Botão associado ao método salva() do managed bean produtoBean

```
1 @ManagedBean
2 public class ProdutoBean {
3     private Produto produto = new Produto();
4     private List<Produto> produtos = new ArrayList<Produto>();
5
6     public String salva() {
7         this.produtos.add(this.produto);
8         this.produto = new Produto();
9         return "lista-produtos";
10    }
11
12    // GETTERS E SETTERS
13 }
```

Código Java 11.1: ProdutoBean.java

O atributo `actionListener`, por sua vez, deve ser associado a um método público do tipo `void` que receba um `ActionEvent` como parâmetro. Conceitualmente, o atributo `actionListener` deve ser associado a métodos que implementam alguma lógica associada à interface do usuário.

No exemplo abaixo, o botão “Clique Aqui” está associado ao método `mudaTexto()` do managed bean `cliqueBean` através do atributo `actionListener` da tag `<h:commandButton>`. Esse método altera o texto do botão para “Clicado”.

```
1 <h:commandButton value="Clique Aqui" actionListener="#{cliqueBean.mudaTexto}" />
```

Código XHTML 11.2: Botão associado ao método mudaTexto() do managed bean cliqueBean

```
1 @ManagedBean
2 public class CliqueBean {
3     public void mudaTexto(ActionEvent e) {
4         UICommand c = (UICommand) e.getComponent();
5         c.setValue("Clicado");
6     }
7 }
```

Código Java 11.2: CliqueBean.java

Os atributos `action` e `actionListener` permitem que no máximo dois métodos sejam associados a um evento de ação. Para associar mais do que dois métodos a um evento de ação, podemos usar a tag `<f:actionListener>`. O valor do atributo `type` dessa tag deve ser o nome de uma classe que implemente a interface `javax.faces.event.ActionListener`. Essa interface exige a criação de um método chamado `processAction()`. Veja o exemplo abaixo.

```
1 <h:commandLink value="Enviar" action="..." actionListener="...">
2     <f:actionListener type="br.com.k19.MudaCorDaFonte"/>
3 </h:commandLink>
```

Código XHTML 11.3: Link associado ao método processAction() da classe MudaCorDaFonte

```
1 package br.com.k19;
2
3 public class MudaCorDaFonte implements ActionListener {
4     @Override
5     public void processAction(ActionEvent e) {
```

```

6     UICommand c = (UICommand) e.getComponent();
7     c.getAttributes().put("style", "color: red;");
8 }
9 }
```

Código Java 11.3: MudaCorDaFonte.java

Nesse exemplo, usamos a tag <f:actionListener> para associar o link ao método processAction() da classe br.com.k19.MudaCorDaFonte.



Importante

Os métodos associados a um evento de ação são executados na seguinte ordem:

1. O método associado com o atributo actionListener.
2. Métodos associados com tag <f:actionListener> de acordo com a ordem em que elas aparecem no arquivo XHTML.
3. O método indicado pelo atributo action.



Importante

Por padrão, os métodos associados a eventos de ação são executados na fase *Invoke Application* do processamento de uma requisição (veja o Capítulo 5 para mais informações).

ValueChangeEvent

Um ValueChangeEvent é produzido quando o valor de uma caixa de texto ou a opção de uma caixa de seleção são alterados. Podemos associar métodos de um managed bean a esses eventos. Tais métodos devem receber um parâmetro do tipo ValueChangeEvent e serem públicos. Para estabelecer essa associação, podemos utilizar o atributo valueChangeListener das caixas de texto ou de seleção ou a tag <f:valueChangeListener>.

Considere um formulário usado para editar as informações dos produtos de uma loja. No exemplo abaixo, associamos uma caixa de texto ao método mudaPreco() do managed bean produtoBean através do atributo valueChangeListener da tag <h:inputText>. Esse método registra a mudança do preço de um produto.

```

1 <h:outputLabel value="Preço: " for="preco" />
2 <h:inputText valueChangeListener="#{produtoBean.mudaPreco}" id="preco" />
```

Código XHTML 11.4: Caixa de texto associada ao método mudaPreco() do managed bean produtoBean

```

1 @ManagedBean
2 public class ProdutoBean {
3     public void mudaPreco(ValueChangeEvent e) {
4         System.out.println("Preço antigo: " + e.getOldValue());
5         System.out.println("Preço novo: " + e.getNewValue());
6     }
}
```

```
7 }
```

Código Java 11.4: ProdutoBean.java

O mesmo pode ser feito com o uso da tag <f:valueChangeListener>. No exemplo abaixo, criamos uma classe chamada RegistraAlteracao, que implementa a interface ValueChangeListener, e associamos a caixa de texto ao método processValueChange() dessa classe.

```
1 <h:outputLabel value="Preço: " for="preco" />
2 <h:inputText id="preco">
3   <f:valueChangeListener type="br.com.k19.RegistraAlteracao" />
4 </h:inputText>
```

Código XHTML 11.5: Caixa de texto associada ao método processValueChange() da classe RegistraAlteracao

```
1 package br.com.k19;
2
3 public class RegistraAlteracao implements ValueChangeListener {
4   @Override
5   public void processValueChange(ValueChangeEvent e) {
6     System.out.println("Preço antigo: " + e.getOldValue());
7     System.out.println("Preço novo: " + e.getNewValue());
8   }
9 }
```

Código Java 11.5: RegistraAlteracao.java



Importante

Os métodos associados a um evento de mudança de valor são executados na seguinte ordem:

1. O método associado com o atributo valueChangeListener.
2. Métodos associados com tag <f:valueChangeListener> na ordem em que elas aparecem no arquivo XHTML.



Importante

Os métodos associados a eventos de mudança de valor são executados na fase *Process Validations* do processamento de uma requisição. Veja o Capítulo 5 para mais informações sobre o processamento de uma requisição.



Exercícios de Fixação

- 1 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Eventos** seguindo os passos vistos no exercício do Capítulo 5.
- 2 Criaremos uma página com três botões. Um dos botões estará habilitado e os outros dois desabilitados. Quando o usuário pressionar o botão habilitado, a aplicação deve escolher aleatoriamente qual dos três botões estará habilitado da próxima vez e desabilitar os outros dois.

Adicione na pasta WebContent o seguinte arquivo XHTML:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10 <title>K19 - Eventos</title>
11 </h:head>
12
13 <h:body>
14   <h:form>
15     <h:commandButton
16       id="botao-jonas"
17       value="Jonas"
18       disabled="false"
19       actionListener="#{botaoBean.sorteiaBotao}"/>
20
21     <h:commandButton
22       id="botao-marcelo"
23       value="Marcelo"
24       disabled="true"
25       actionListener="#{botaoBean.sorteiaBotao}"/>
26
27     <h:commandButton
28       id="botao-rafael"
29       value="Rafael"
30       disabled="true"
31       actionListener="#{botaoBean.sorteiaBotao}"/>
32   </h:form>
33 </h:body>
34 </html>
```

Código XHTML 11.6: *botoes.xhtml*

- 3 Crie um managed bean para implementar o tratamento dos botões. Adicione em um pacote chamado `managedbeans` a seguinte classe:

```

1 package managedbeans;
2
3 @ManagedBean
4 public class BotaoBean {
5   public void sorteiaBotao(ActionEvent event){
6     UIComponent formulario = event.getComponent().getParent();
7
8     UIComponent botaoJonas = formulario.findComponent("botao-jonas");
9     UIComponent botaoMarcelo = formulario.findComponent("botao-marcelo");
10    UIComponent botaoRafael = formulario.findComponent("botao-rafael");
11
12    botaoJonas.getAttributes().put("disabled", true);
13    botaoMarcelo.getAttributes().put("disabled", true);
14    botaoRafael.getAttributes().put("disabled", true);
15
16    double numero = Math.random();
17
18    if(numero < 1.0/3.0){
19      botaoJonas.getAttributes().put("disabled", false);
20    } else if(numero < 2.0/3.0){
21      botaoMarcelo.getAttributes().put("disabled", false);
22    } else {
23      botaoRafael.getAttributes().put("disabled", false);
24    }
25  }
26}
```

Código Java 11.6: BotaoBean.java

- 4 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Eventos/botoes.xhtml>

PhaseEvent

Os PhaseEvents são eventos disparados automaticamente pelo JSF antes e depois de cada uma das fases do processamento de uma requisição. Esses eventos são úteis basicamente para monitorar a execução dessas fases.

Para tratar um PhaseEvent, podemos criar uma classe que implemente a interface `javax.faces.event.PhaseListener`. Além disso, precisamos registrar essa classe no arquivo `faces-config.xml`. A interface `PhaseListener` obriga a implementação de três métodos: `afterPhase()`, `beforePhase()` e `getPhaseId()`.

O método `getPhaseId()` deve devolver a referência de um objeto do tipo `javax.faces.event.PhaseId`, que indica em quais fases do processamento de uma requisição estamos interessados. Se estivermos interessados nos eventos da fase *Apply Request Values*, por exemplo, podemos fazer esse método devolver `PhaseId.APPLY_REQUEST_VALUES`. Os valores que podem ser utilizados são:

- `PhaseId.RESTORE_VIEW`
- `PhaseId.APPLY_REQUEST_VALUES`
- `PhaseId.INVOKE_APPLICATION`
- `PhaseId.PROCESS_VALIDATIONS`
- `PhaseId.UPDATE_MODEL_VALUES`
- `PhaseId.RENDER_RESPONSE`
- `PhaseId.ANY_PHASE`

O método `beforePhase()` recebe a referência de um `PhaseEvent` como parâmetro e é executado antes das fases especificadas pelo método `getPhaseId()`. O método `afterPhase()` é análogo ao `beforePhase()` e é executado após as fases especificadas pelo método `getPhaseId()`.

No exemplo abaixo, criamos a classe `MonitorPhaseListener` que implementa a interface `PhaseListener`. No método `getPhaseId()`, devolvemos `PhaseId.ANY_PHASE`, que indica que estamos interessados nos eventos relacionados a todas as fases.

```
1 package br.com.k19;
2
3 public class MonitorPhaseListener implements PhaseListener {
4
5     @Override
6     public void afterPhase(PhaseEvent event) {
7         System.out.println("MonitorPhaseListener.afterPhase()");
8         System.out.println(event.getPhaseId());
9     }
10
11     @Override
```

```

12     public void beforePhase(PhaseEvent event) {
13         System.out.println("MonitorPhaseListener.beforePhase()");
14         System.out.println(event.getPhaseId());
15     }
16
17     @Override
18     public PhaseId getPhaseId() {
19         return PhaseId.ANY_PHASE;
20     }
21 }
```

Código Java 11.7: MonitorPhaseListener.java

No arquivo faces-config.xml, registramos essa classe da seguinte forma:

```

1 ...
2 <lifecycle>
3   <phase-listener>br.com.k19.MonitorPhaseListener</phase-listener>
4 </lifecycle>
5 ...
```

Código XML 11.1: faces-config.xml

Exercícios de Fixação

- 5** Crie uma classe em um pacote chamado phaselisteners dentro do diretório src com o seguinte conteúdo:

```

1 package phaselisteners;
2
3 public class MonitorPhaseListener implements PhaseListener {
4
5     @Override
6     public void afterPhase(PhaseEvent event) {
7         System.out.println("MonitorPhaseListener.afterPhase() - " + event.getPhaseId());
8     }
9
10    @Override
11    public void beforePhase(PhaseEvent event) {
12        System.out.println("MonitorPhaseListener.beforePhase() - " + event.getPhaseId());
13    }
14
15    @Override
16    public PhaseId getPhaseId() {
17        return PhaseId.ANY_PHASE;
18    }
19 }
```

Código Java 11.8: MonitorPhaseListener.java

- 6** Registre a classe MonitorPhaseListener no arquivo faces-config.xml.

```

1 ...
2 <lifecycle>
3   <phase-listener>phaselisteners.MonitorPhaseListener</phase-listener>
4 </lifecycle>
5 ...
```

Código XML 11.2: faces-config.xml

- 7 Acesse a página `botoes.xhtml`, clique nos botões e verifique as mensagens impressas no console.

SystemEvent

SystemEvents são similares aos PhaseEvents no sentido de que ambos estão relacionados a pontos específicos de execução de uma aplicação JSF. Contudo, os eventos de sistema são gerados em diversos pontos não cobertos pelos eventos de fase. Todas as classes que definem eventos de sistema são subclasses de `javax.faces.event.SystemEvent`. Apresentamos abaixo algumas dessas classes.

- `PostConstructApplicationEvent`
É gerado imediatamente após o início da aplicação, depois que todas as configurações são processadas.
- `PreDestroyApplicationEvent`
Esse tipo de evento é gerado imediatamente antes da aplicação ser finalizada.
- `ExceptionQueuedEvent`
Esse evento é gerado assim que uma exceção não esperada é lançada durante o processamento de uma requisição. Alguns exemplos de exceções esperadas durante o processamento de uma requisição são aquelas relacionadas à conversão e validação dos dados.
- `PreValidateEvent` e `PostValidateEvent`
Esses eventos são gerados imediatamente antes e logo após um componente ser validado, respectivamente.

As três primeiras classes listadas acima são subclasses diretas de `SystemEvent`, enquanto que as duas últimas são subclasses de `javax.faces.event.ComponentSystemEvent`.

Uma das formas para se registrar um interessado em eventos de sistema é usando a tag `<f:event>`. Essa tag possui dois atributos: `type` e `listener`. O atributo `type` é usado para indicar o tipo de evento. Alguns dos possíveis valores para esse atributo são `preValidate` e `postValidate`. O atributo `listener` deve indicar um método que será chamado quando o evento for processado. Esse método deve ser público, do tipo `void`, e receber como parâmetro uma referência para um objeto do tipo `javax.faces.event.ComponentSystemEvent`.

O evento `PostValidateEvent` pode ser útil na validação de múltiplos campos, por exemplo. Considere uma aplicação bancária que permite gerar extratos da conta corrente. Os dados de entrada são duas datas e o resultado é o extrato referente ao período determinado por essas datas. Uma restrição que devemos impor é que a data final não pode vir antes da data inicial.

No formulário do código abaixo, usamos a tag `<f:event>` para indicar que o método `validaDatas()` do managed bean `extratoBean` deve ser chamado assim que o evento `postValidate` for processado.

```
1 <h:form>
2   <f:event type="postValidate" listener="#{extratoBean.validaDatas}" />
3   <h:messages/>
4
5   <h:outputLabel value="Data inicial: " for="data-inicial"/>
6   <h:inputText value="#{extratoBean.dataInicial}" id="data-inicial" required="true">
7     <f:convertDateTime pattern="dd/MM/yyyy" />
8   </h:inputText>
9 
```

```

10 <h:outputLabel value="Data final: " for="data-final"/>
11 <h:inputText value="#{extratoBean.dataFinal}" id="data-final" required="true">
12   <f:convertDateTime pattern="dd/MM/yyyy" />
13 </h:inputText>
14
15 <h:commandButton value="Ver extrato" action="#{extratoBean.geraExtrato}" />
16 </h:form>

```

Código XHTML 11.7: O método validaDatas() do managed bean extratoBean é registrado para receber eventos do tipo PostValidateEvent

Na classe ExtratoBean, definimos o método validaDatas() que recebe uma referência para um ComponentSystemEvent como parâmetro. Após verificar se as datas inicial e final são validas (tarefa executada pelo validador de datas padrão de cada um dos componentes), prosseguimos com a verificação da nossa restrição adicional, isto é, a data inicial deve vir antes da data final.

Se essa restrição não for respeitada, não devemos exibir extrato algum. Dessa forma, adicionamos uma mensagem no contexto do processamento da requisição e redirecionamos o fluxo para a última fase do processamento, isto é, para a fase Render Response.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class ExtratoBean {
5
6   private Date dataInicial;
7   private Date dataFinal;
8
9   public void validaDatas(ComponentSystemEvent event) {
10
11     UIComponent source = event.getComponent();
12     UIInput dataInicialInput = (UIInput) source.findComponent("data-inicial");
13     UIInput dataFinalInput = (UIInput) source.findComponent("data-final");
14
15     if (dataInicialInput.isValid() && dataFinalInput.isValid()) {
16
17       Date dataInicialEscolhida = (Date) dataInicialInput.getLocalValue();
18       Date dataFinalEscolhida = (Date) dataFinalInput.getLocalValue();
19
20       if (dataFinalEscolhida.before(dataInicialEscolhida)) {
21         FacesMessage message =
22           new FacesMessage("A data final não pode vir antes da data inicial");
23         message.setSeverity(FacesMessage.SEVERITY_ERROR);
24         FacesContext context = FacesContext.getCurrentInstance();
25         context.addMessage(source.getClientId(), message);
26         context.renderResponse();
27       }
28     }
29   }
30
31   public void geraExtrato() {
32     FacesMessage message = new FacesMessage("extrato gerado com sucesso");
33     message.setSeverity(FacesMessage.SEVERITY_INFO);
34     FacesContext context = FacesContext.getCurrentInstance();
35     context.addMessage(null, message);
36   }
37
38   // GETTERS E SETTERS
39 }

```

Código Java 11.9: ExtratoBean.java



Exercícios de Fixação

- 8 Na pasta WebContent adicione um arquivo XHTML com o seguinte conteúdo:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10  <title>K19 - Eventos</title>
11 </h:head>
12
13 <h:body>
14  <h:form>
15    <f:event type="postValidate" listener="#{extratoBean.validaData}" />
16    <h:messages/>
17    <h:outputLabel value="Data inicial: " for="data-inicial" />
18    <h:inputText
19      value="#{extratoBean.dataInicial}"
20      id="data-inicial"
21      required="true">
22      <f:convertDateTime pattern="dd/MM/yyyy" />
23    </h:inputText>
24
25    <h:outputLabel value="Data final: " for="data-final" />
26    <h:inputText
27      value="#{extratoBean.dataFinal}"
28      id="data-final"
29      required="true">
30      <f:convertDateTime pattern="dd/MM/yyyy" />
31    </h:inputText>
32
33    <h:commandButton value="Ver extrato" action="#{extratoBean.geraExtrato}" />
34  </h:form>
35 </h:body>
36 </html>
```

Código XHTML 11.8: extrato.xhtml

- 9 Adicione a classe ExtratoBean em um pacote chamado managedbeans com o seguinte conteúdo:

```

1 package managedbeans;
2
3 @ManagedBean
4 public class ExtratoBean {
5
6     private Date dataInicial;
7     private Date dataFinal;
8
9     public void validaDatas(ComponentSystemEvent event) {
10
11         UIComponent source = event.getComponent();
12         UIInput dataInicialInput = (UIInput) source.findComponent("data-inicial");
13         UIInput dataFinalInput = (UIInput) source.findComponent("data-final");
14
15         if (dataInicialInput.isValid() && dataFinalInput.isValid()) {
16
17             Date dataInicialEscolhida = (Date) dataInicialInput.getLocalValue();
18             Date dataFinalEscolhida = (Date) dataFinalInput.getLocalValue();
19
20             if (dataFinalEscolhida.before(dataInicialEscolhida)) {
21                 FacesMessage message =
22                     new FacesMessage("A data final não pode vir antes da data inicial");
23                 message.setSeverity(FacesMessage.SEVERITY_ERROR);
24             }
25         }
26     }
27 }
```

```

24     FacesContext context = FacesContext.getCurrentInstance();
25     context.addMessage(source.getClientId(), message);
26     context.renderResponse();
27   }
28 }
29
30
31 public void geraExtrato() {
32   FacesMessage message = new FacesMessage("extrato gerado com sucesso");
33   message.setSeverity(FacesMessage.SEVERITY_INFO);
34   FacesContext context = FacesContext.getCurrentInstance();
35   context.addMessage(null, message);
36 }
37
38 // GETTERS E SETTERS
39 }
```

Código Java 11.10: ExtratoBean.java

- 10 Acesse a página extrato.xhtml e teste a validação das datas.

Immediate

Por padrão, a conversão e a validação dos dados de um componente de entrada (`<h:inputText>`, por exemplo) são realizadas na fase *Process Validation*. Além disso, os eventos de mudança de valor (`ValueChangeEvent`) também ocorrem nessa fase.

Também por padrão, os eventos de ação (`ActionEvent`) associados aos componentes `<h:commandButton>` ou `<h:commandLink>` são disparados no final da fase *Invoke Application* do processamento de uma requisição.

Porém, esse comportamento pode ser alterado através do atributo `immediate` desses componentes. Se o valor do atributo `immediate` de um componente de entrada for `true`, então a conversão e validação dos dados desse componente irão ocorrer durante a fase *Apply Request Values*, onde também irão ser disparados possíveis eventos de mudança de valor. No caso dos links e botões, os eventos de ação serão disparados no final da fase *Apply Request Values*.



Exercícios de Fixação

Crie um formulário para o cadastro de pessoas. Esse formulário deve possuir três campos obrigatórios. O primeiro deve ser uma caixa de texto para receber o nome de uma pessoa. O segundo deve ser uma caixa de seleção para o usuário escolher um estado. O terceiro campo também deve ser uma caixa de seleção, que permitirá ao usuário escolher uma cidade. A caixa de seleção de cidade deve exibir apenas as cidades do estado escolhido. Ou seja, quando o usuário selecionar determinado estado, a lista de cidades deve ser atualizada de acordo.

- 11 Na pasta `src` do projeto K19-Eventos, crie um pacote chamado `model`.
- 12 No pacote `model`, crie uma classe chamada `Estado` para modelar um estado. Essa classe deve ter como atributos o nome e a sigla do estado, bem como uma lista de cidades.

```

1 package model;
2
```

```

3  public class Estado {
4      private String nome;
5      private String sigla;
6      private List<String> cidades = new ArrayList<String>();
7
8      // GETTERS E SETTERS
9  }

```

Código Java 11.11: Estado.java

- 13 No pacote managedbeans, crie uma classe chamada CadastroBean, que dará suporte ao formulário de cadastro. Essa classe também deve armazenar uma lista de estados que poderão ser escolhidos no formulário. Anote essa classe com @SessionScoped para que os dados do formulário sejam mantidos entre uma requisição e outra.

```

1 package managedbeans;
2
3 @ManagedBean
4 @SessionScoped
5 public class CadastroBean {
6
7     private String nome;
8     private String cidade;
9     private String siglaDoEstadoEscolhido;
10
11    private Estado estadoSelecionado = new Estado();
12    private List<Estado> estados = new ArrayList<Estado>();
13
14    public CadastroBean() {
15        Estado sp = new Estado();
16        sp.setSigla("SP");
17        sp.setNome("São Paulo");
18        sp.getCidades().add("São Paulo");
19        sp.getCidades().add("Campinas");
20
21        Estado rj = new Estado();
22        rj.setSigla("RJ");
23        rj.setNome("Rio de Janeiro");
24        rj.getCidades().add("Rio de Janeiro");
25        rj.getCidades().add("Niterói");
26
27        Estado rn = new Estado();
28        rn.setSigla("RN");
29        rn.setNome("Rio Grande do Norte");
30        rn.getCidades().add("Natal");
31        rn.getCidades().add("Mossoró");
32
33        this.estados.add(sp);
34        this.estados.add(rj);
35        this.estados.add(rn);
36    }
37
38    public void mudaEstado(ValueChangeEvent event) {
39        this.siglaDoEstadoEscolhido = event.getNewValue().toString();
40        for (Estado e : this.estados) {
41            if (e.getSigla().equals(this.siglaDoEstadoEscolhido)) {
42                this.estadoSelecionado = e;
43                break;
44            }
45        }
46    }
47
48    // GETTERS E SETTERS
49 }

```

Código Java 11.12: CadastroBean.java

- 14** Na pasta WebContent, crie um arquivo chamado `cadastro.xhtml` e implemente o formulário desejado.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7  <h:head>
8   <title>K19 Treinamentos</title>
9  </h:head>
10
11 <h:body>
12  <h1>Cadastro</h1>
13  <h:form>
14   <h:panelGrid columns="3">
15
16    <h:outputLabel value="Nome: " for="nome" />
17    <h:inputText value="#{cadastroBean.nome}" required="true" id="nome" />
18    <h:message for="nome" />
19
20    <h:outputLabel value="Estado: " for="estado" />
21    <h:selectOneMenu id="estado"
22      value="#{cadastroBean.siglaDoEstadoEscolhido}"
23      valueChangeListener="#{cadastroBean.mudaEstado}"
24      onchange="this.form.submit();"
25      required="true">
26
27     <f:selectItem itemLabel="Escolha um estado" noSelectionOption="true" />
28     <f:selectItems
29       value="#{cadastroBean.estados}"
30       var="estado"
31       itemLabel="#{estado.nome}"
32       itemValue="#{estado.sigla}" />
33
34    </h:selectOneMenu>
35    <h:message for="estado" />
36
37    <h:outputLabel value="Cidade: " for="cidade" />
38    <h:selectOneMenu id="cidade" value="#{cadastroBean.cidade}" required="true">
39      <f:selectItem itemLabel="Escolha uma cidade" noSelectionOption="true" />
40      <f:selectItems value="#{cadastroBean.estadoSelecionado.cidades}" />
41    </h:selectOneMenu>
42    <h:message for="cidade" />
43
44   </h:panelGrid>
45   <h:commandButton value="Cadastrar" />
46  </h:form>
47 </h:body>
48 </html>
```

Código XHTML 11.9: `cadastro.xhtml`

Para que a lista de cidades seja atualizada assim que o usuário selecionar um novo estado, usamos os atributos `onchange` e `valueChangeListener` do componente `<h:selectOneMenu>`. Definimos o valor do atributo `onchange` como sendo `"this.form.submit();"`, o que significa que o formulário será submetido toda vez que um novo estado for selecionado. Além disso, o atributo `valueChangeListener` está associado ao método `mudaEstado()`, o que significa que esse método será chamado quando o evento de mudança de valor for gerado no processamento da requisição. Nesse método, alteramos o valor da propriedade `estadoSelecionado` para refletir a escolha atual do usuário, o que implica que a caixa de seleção de cidade exibirá as cidades desse estado. Veja as linhas em

destaque no código acima.

- 15 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Eventos/cadastro.xhtml>

Note que, quando selecionamos um estado, o formulário é submetido automaticamente e os dados de todos os campos são convertidos e validados.

- 16 Para atualizar a lista de cidades, precisamos que o dado presente na caixa de seleção de estado passe pelas etapas de conversão e validação. Por outro lado, queremos evitar que os demais campos sejam convertidos e validados, bem como evitar que o cadastro seja efetivado sem que o usuário tenha submetido o formulário.

Assim, precisamos (i) antecipar a conversão e validação dessa caixa de seleção e, logo em seguida, (ii) desviar o fluxo do processamento da requisição para a fase *Render Response*. A primeira tarefa pode ser feita definindo como true o valor do atributo *immediate* do componente `<h:selectOneMenu>`. O fluxo do processamento da requisição pode ser desviado para a fase *Render Response* se chamartermos o método `renderResponse()` da instância da classe `FacesContext` logo após atualizarmos o valor do atributo `estadoSelecionado` do managed bean `cadastroBean`.

```

1 ...
2 <h:selectOneMenu id="estado"
3   value="#{cadastroBean.siglaDoEstadoEscolhido}"
4   required="true"
5   valueChangeListener="#{cadastroBean.mudaEstado}"
6   onchange="this.form.submit();"
7   immediate="true">
8
9   <f:selectItem itemLabel="Escolha um estado" noSelectionOption="true" />
10  <f:selectItems value="#{cadastroBean.estados}" var="estado"
11    itemLabel="#{estado.nome}" itemValue="#{estado.sigla}" />
12 </h:selectOneMenu>
13 ...

```

Código XHTML 11.10: *cadastro.xhtml*

```

1 ...
2 public void mudaEstado(ValueChangeEvent event) {
3     this.siglaDoEstadoEscolhido = event.getNewValue().toString();
4     for (Estado e : this.estados) {
5         if (e.getSigla().equals(this.siglaDoEstadoEscolhido)) {
6             this.estadoSelecionado = e;
7             break;
8         }
9     }
10    FacesContext.getCurrentInstance().renderResponse();
11 }
12 ...

```

Código Java 11.13: *CadastroBean.java*

AJAX

Quando as aplicações possuem telas complexas, com grande quantidade de conteúdo, não é interessante recarregar uma página inteira só para modificar uma pequena parte da tela.

Com o intuito de melhorar a interatividade entre as aplicações e os usuários, podemos aplicar o conceito do AJAX (Asynchronous Javascript And XML). Aplicando esse conceito, obtemos duas capacidades muito úteis. A primeira é a possibilidade de atualizar trechos de uma página ao invés da página inteira. A segunda é poder realizar requisições sem interromper a navegação dos usuários.

Por exemplo, considere uma aplicação web para a visualização de fotos. As fotos são exibidas individualmente em uma tela que também apresenta outros conteúdos. Os usuários podem, através do link “Próxima”, avançar para a próxima foto, ou voltar para a foto anterior através do link “Anterior”. Quando o usuário decide visualizar a próxima foto ou a anterior, não é necessário recarregar a página inteira, já que o único conteúdo alterado da página é a foto.

Outro exemplo, considere uma aplicação de mensagens instantâneas (GTalk, MSN, ...). A lista-gem de contatos de um usuário pode ser atualizada frequentemente sem que ele tenha que pressionar qualquer botão ou link, sem que a página tenha de ser recarregada e sem interromper a navegação do usuário.

A versão 2 do JSF, diferentemente das anteriores, oferece suporte nativo a AJAX. Neste capítulo, veremos como utilizar esse suporte.

Fazendo requisições AJAX

As requisições AJAX são realizadas quando determinados eventos ocorrem. Esses eventos estão fortemente relacionados aos componentes visuais adicionados às páginas. Devemos indicar para o JSF quais componentes e eventos devem realizar requisições AJAX. Para fazer isso, utilizamos a tag `<f:ajax>`.

```
1 <h:inputText>
2   <f:ajax/>
3 </h:inputText>
```

Código XHTML 12.1: Aplicando a tag `<f:ajax>`

No exemplo acima, uma requisição AJAX será realizada toda vez que o valor da caixa de texto for modificado, já que o evento padrão associado ao componente `<h:inputText>` é o **onchange**.

Por outro lado, podemos explicitar o evento que deve disparar as requisições AJAX através do atributo **event** da tag `<f:ajax>`. Devemos tomar cuidado pois nem todos os eventos são aceitos por todos os componentes.

```
1 <h:inputText>
```

```
1 <f:ajax event="keyup" />
2 </h:inputText>
```

Código XHTML 12.2: Definindo o tipo de evento que deve disparar requisições AJAX

Quando temos vários componentes para os quais desejamos oferecer o suporte do AJAX, podemos agrupá-los através da tag `<f:ajax>`.

```
1 <f:ajax>
2   <h:inputText/>
3   <h:inputSecret/>
4   <h:commandButton value="OK" />
5 </f:ajax>
```

Código XHTML 12.3: Aplicando a tag `<f:ajax>` a vários componentes

Novamente, se não escolhermos explicitamente o evento que vai disparar as requisições AJAX, o JSF assumirá o evento padrão de cada componente. O padrão dos componentes `<h:inputText>` e `<h:inputSecret>` é `onchange`. Já o padrão do componente `<h:commandButton>` é `onclick`.

Podemos explicitar o evento que deve disparar as requisições AJAX para um determinado grupo de componentes da mesma forma como fizemos anteriormente.

```
1 <f:ajax event="mouseout" />
2   <h:inputText/>
3   <h:inputSecret/>
4   <h:commandButton value="OK" />
5 </f:ajax>
```

Código XHTML 12.4: Definindo o tipo de evento que deve disparar requisições AJAX

Processando uma parte específica da tela

Quando uma requisição AJAX é feita, podemos determinar quais componentes da tela devem ser avaliados pelo JSF. Por exemplo, quando enviamos um formulário, talvez seja necessário avaliar apenas os componentes que estão no próprio formulário.

Podemos definir quais componentes devem ser avaliados no servidor através do atributo **execute** da tag `<f:ajax>`. O valor desse atributo deve ser uma lista contendo os identificadores dos componentes que precisam ser avaliados no servidor. Quando um componente é avaliado, todos os componentes definidos em seu corpo também serão avaliados.

```
1 <h:form id="formulario-login">
2   <h:inputText/>
3
4   <h:inputSecret/>
5
6   <h:commandButton value="Enviar">
7     <f:ajax event="click" execute="formulario-login" />
8   </h:commandButton>
9 </h:form>
```

Código XHTML 12.5: Definindo quais componentes serão avaliados

No exemplo acima, quando o botão “Enviar” for pressionado, uma requisição AJAX será realizada. No servidor, o formulário e todos os componentes definidos dentro dele serão avaliados.

Recarregando parte da tela

Podemos definir quais componentes devem ser atualizados quando a resposta de uma requisição AJAX chega no navegador. Para isso, devemos utilizar o atributo `render` da tag `<f:ajax>`. O valor desse atributo deve ser uma lista contendo os identificadores dos componentes que precisam ser atualizados quando a resposta de uma requisição AJAX chegar no navegador.

```

1 <h:commandButton value="Gera Número">
2   <f:ajax event="click" render="numero"/>
3 </h:commandButton>
4 <h:outputText id="numero" value="#{geradorDeNumeroBean.numero}" />

```

Código XHTML 12.6: Definindo quais componentes serão atualizados

No exemplo acima, uma requisição AJAX é realizada quando o botão “Gera Número” é pressionado. Quando a resposta dessa requisição chega no navegador, apenas o componente `<h:outputText>` com o identificador `numero` será atualizado. Já no código abaixo, dois componentes `<h:outputText>` são atualizados. Mais especificamente, os componentes com os identificadores `numero1` e `numero2` são atualizados.

```

1 <h:commandButton value="Gera Números">
2   <f:ajax event="click" render="numero1 numero2"/>
3 </h:commandButton>
4 <h:outputText id="numero1" value="#{geradorDeNumeroBean.numero}" />
5 <h:outputText id="numero2" value="#{geradorDeNumeroBean.numero}" />

```

Código XHTML 12.7: Definindo dois componentes para serem atualizados

Associando um procedimento a uma requisição AJAX

Podemos associar um método a uma requisição AJAX. Esse método será executado durante o processamento dessa requisição no servidor. Mais especificamente, ele será executado na fase *Invoke Application*. Essa associação é realizada através do atributo `listener` da tag `<f:ajax>`.

```

1 <h:commandButton value="Salva">
2   <f:ajax
3     event="click"
4     execute="formulario"
5     render="formulario"
6     listener="#{produtoBean.salva}" />
7 </h:commandButton>

```

Código XHTML 12.8: Definindo quais componentes serão atualizados

No exemplo acima, as requisições AJAX realizadas através do botão “Salva” foram associadas ao método `salva()` do managed bean `produtoBean`.

Palavras especiais

Como podemos passar uma lista de componentes para os atributos `render` e `execute`, o JSF criou palavras chaves associadas a grupos especiais de componentes.

@all refere-se a todos os componentes da tela.

@none refere-se a nenhum componente.

@this refere-se ao componente que disparou a requisição AJAX.

@form refere-se aos componentes do formulário que contém o componente que disparou a requisição AJAX.

Podemos alterar o código do formulário anterior para utilizar a palavra especial **@form** no lugar do identificador do formulário.

```
1 <h:form id="formulario-login">
2   <h:inputText/>
3
4   <h:inputSecret/>
5
6   <h:commandButton value="Enviar">
7     <f:ajax event="click" execute="@form"/>
8   </h:commandButton>
9 </h:form>
```

Código XHTML 12.9: Utilizando a palavra especial @form



Exercícios de Fixação

- 1 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Ajax** seguindo os passos vistos no exercício do Capítulo 5.
- 2 Criaremos um formulário com a validação dos campos realizadas com AJAX. Na pasta *WebContent*, adicione um arquivo XHTML com o seguinte conteúdo:

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10  <title>K19 - Ajax</title>
11 </h:head>
12
13 <h:body>
14   <h:form>
15     <h:panelGrid columns="2">
16       <h:inputText id="caixa">
17         <f:ajax event="keyup" execute="caixa" render="mensagem" />
18         <f:validateLength minimum="5" maximum="10" />
19       </h:inputText>
20
21       <h:message id="mensagem" for="caixa" style="color: red;">
22
23         <h:commandButton value="Enviar" />
24       </h:panelGrid>
25     </h:form>
26   </h:body>
27 </html>
```

Código XHTML 12.10: formulario.xhtml

- 3 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Ajax/formulario.xhtml>



INTEGRAÇÃO JSF E JPA

Como vimos nos primeiros capítulos, os SGDBs são utilizados para armazenar os dados manipulados pelas aplicações. Até agora, os dados manipulados pelas aplicações JSF desenvolvidas nos exercícios não foram armazenados em um SGDB. Adicionaremos essa capacidade às aplicações JSF neste capítulo. Para isso, mostraremos uma maneira de integrar os recursos do JSF e do JPA.

Bibliotecas

Para utilizar os recursos do JPA em uma aplicação JSF, os jars do provedor JPA e do driver JDBC que serão utilizados devem estar no classpath da aplicação. Nos capítulos anteriores, as aplicações JSF desenvolvidas nos exercícios foram implantadas no Glassfish 3.0.1 que é um servidor de aplicação Java EE 6.

Por padrão, a versão 3.0.1 do Glassfish possui os jars do provedor JPA EclipseLink. Dessa forma, as aplicações JSF implantadas nessa versão do Glassfish utilizarão o EclipseLink como implementação do JPA. Contudo, queremos utilizar o provedor JPA Hibernate. Podemos facilmente substituir os jars do EclipseLink pelos jars do Hibernate através da interface de administração do Glassfish.



Mais Sobre

Consulte o artigo da K19 sobre a substituição dos jars do provedor JPA EclipseLink pelos jars do provedor JPA Hibernate.

<http://www.k19.com.br/artigos/configurando-hibernate-no-glassfish-3-1/>

Utilizaremos o MySQL Server como SGDB. Dessa forma, devemos adicionar o driver JDBC do MySQL Server no classpath das aplicações JSF. O Glassfish 3.0.1 não possui os jars desse driver JDBC. Contudo, podemos adicioná-los manualmente. Para isso, basta acrescentar os jars do driver JDBC do MySQL em uma pasta apropriada do Glassfish. Nos exercícios deste capítulo, mostraremos como realizar tal tarefa.

Configuração

Como vimos no Capítulo 3, devemos configurar as unidades de persistência utilizadas através do arquivo `persistence.xml` da pasta META-INF do classpath da aplicação.

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

```

5   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
6   ns/persistence/persistence_2_0.xsd">
7   <persistence-unit name="K19-PU" transaction-type="RESOURCE_LOCAL">
8     <provider>org.hibernate.ejb.HibernatePersistence</provider>
9     <properties>
10       <property name="hibernate.dialect" value="org.hibernate.dialect.←
11         MySQL5InnoDBDialect"/>
12       <property name="hibernate.hbm2ddl.auto" value="create"/>
13       <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
14       <property name="javax.persistence.jdbc.user" value="usuario"/>
15       <property name="javax.persistence.jdbc.password" value="senha"/>
16       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/←
17         K19-DB"/>
18     </properties>
19   </persistence-unit>
20 </persistence>
21
22
23

```

Código XML 13.1: persistence.xml

Mapeamento

Também devemos definir o mapeamento das entidades. No Capítulo 3, vimos como utilizar as anotações do JPA para estabelecer esse mapeamento.

```

1 @Entity
2 public class Produto {
3   @Id @GeneratedValue
4   private Long id;
5
6   private String nome;
7
8   private Double preco;
9
10  // GETTERS e SETTERS
11 }

```

Código Java 13.1: Produto.java

Inicialização e Finalização

As unidades de persistência devem ser inicializadas antes de serem utilizadas, e finalizadas quando não forem mais necessárias. A inicialização e a finalização de uma unidade de persistência devem ser realizadas apenas uma vez durante a execução da aplicação.

Para implementar essa característica em aplicações web Java, podemos utilizar um filtro. Os filtros de uma aplicação web Java são inicializados automaticamente depois que a aplicação é implantada no Web Container e antes da primeira requisição HTTP. Além disso, eles são finalizados ao término da execução da aplicação.

Para adicionar um filtro em uma aplicação web Java, é necessário criar uma classe que implemente a interface `javax.servlet.Filter`.

```

1  @WebFilter(servletNames={"Faces Servlet"})
2  public class JPAFilter implements Filter {
3
4      private EntityManagerFactory factory;
5
6      @Override
7      public void init(FilterConfig filterConfig) throws ServletException {
8          this.factory = Persistence.createEntityManagerFactory("K19-PU");
9      }
10
11     @Override
12     public void destroy() {
13         this.factory.close();
14     }
15
16     @Override
17     public void doFilter(ServletRequest request, ServletResponse response,
18             FilterChain chain) throws IOException, ServletException {
19
20         // por enquanto vazio
21     }
22 }
```

Código Java 13.2: JPAFilter.java

Um filtro pode ser registrado no Web Container através da anotação **@WebFilter**. Com essa anotação, podemos definir qual servlet será associada ao filtro. No exemplo acima, o filtro definido pela classe JPAFilter foi associado a servlet *Faces Servlet*.

O método **init()** é chamado automaticamente na inicialização do filtro. No exemplo acima, esse método inicializa a unidade de persistência *K19-PU*. O método **destroy()** é chamado automaticamente para desativar o filtro no encerramento da aplicação. No exemplo acima, finalizamos a unidade de persistência *K19-PU*.

Transações

Como vimos no Capítulo 3, para atualizar as informações armazenadas no SGDB de acordo com os dados da memória da aplicação, devemos abrir uma transação e confirmá-la através do método **commit()**.

O filtro criado anteriormente para controlar a inicialização e finalização das unidades de persistência pode também gerenciar a abertura e a confirmação das transações da aplicação. Para isso, utilizaremos o método **doFilter()** desse filtro.

```

1  @WebFilter(servletNames={"Faces Servlet"})
2  public class JPAFilter implements Filter {
3
4      private EntityManagerFactory factory;
5
6      @Override
7      public void init(FilterConfig filterConfig) throws ServletException {
8          this.factory = Persistence.createEntityManagerFactory("K19-PU");
9      }
10
11     @Override
12     public void destroy() {
13         this.factory.close();
14     }
15 }
```

```

16  @Override
17  public void doFilter(ServletRequest request, ServletResponse response,
18      FilterChain chain) throws IOException, ServletException {
19
20      // CHEGADA
21      EntityManager manager = this.factory.createEntityManager();
22      request.setAttribute("EntityManager", manager);
23      entityManager.getTransaction().begin();
24      // CHEGADA
25
26      // FACES SERVLET
27      chain.doFilter(request, response);
28      // FACES SERVLET
29
30      // SAÍDA
31      try {
32          entityManager.getTransaction().commit();
33      } catch (Exception e) {
34          entityManager.getTransaction().rollback();
35      } finally {
36          entityManager.close();
37      }
38      // SAÍDA
39  }
40

```

Código Java 13.3: JPAFilter.java

No exemplo acima, o método `doFilter()` é chamado toda vez que uma requisição é realizada para a servlet *Faces Servlet*. Antes de repassar a requisição para a *Faces Servlet*, o método `doFilter()` cria um `EntityManager`, armazena-o na requisição e abre uma transação. Depois que a *Faces Servlet* processou a requisição, o método `doFilter()` tenta confirmar a transação através do método `commit()`. Se um erro ocorrer nessa tentativa, o método `rollback()` é chamado para cancelar a transação.

Recuperando o EntityManager da Requisição

O `EntityManager` armazenado dentro da requisição pelo filtro pode ser recuperado a qualquer momento durante o processamento da requisição. Veja o código abaixo.

```

1 FacesContext fc = FacesContext.getCurrentInstance();
2 ExternalContext ec = fc.getExternalContext();
3 HttpServletRequest request = (HttpServletRequest) ec.getRequest();
4 EntityManager manager = (EntityManager) request.getAttribute("EntityManager");

```

Código Java 13.4: Recuperando o EntityManager da requisição

O `EntityManager` será utilizado pela aplicação para realizar as operações de persistência.



Exercícios de Fixação

1 Entre na pasta K19-Arquivos/MySQL-Connector-JDBC da Área de Trabalho e copie o arquivo **mysql-connector-java-5.1.13-bin.jar** para pasta glassfishv3/glassfish/lib também da sua Área de Trabalho. OBS: O Glassfish deve ser reiniciado para reconhecer o driver JDBC do MySQL.

2 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Integracao-JSF-JPA** seguindo os

passos vistos no exercício do Capítulo 5.

- 3 Adicione uma pasta chamada META-INF na pasta src do projeto K19-Integracao-JSF-JPA.
- 4 Configure o JPA adicionando o arquivo persistence.xml na pasta META-INF.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <persistence version="2.0"
3   xmlns="http://java.sun.com/xml/ns/persistence"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
6     ns/persistence/persistence_2_0.xsd">
7
8   <persistence-unit name="K19-PU" transaction-type="RESOURCE_LOCAL">
9     <provider>org.hibernate.ejb.HibernatePersistence</provider>
10    <properties>
11      <property name="hibernate.dialect" value="org.hibernate.dialect.←
12        MySQL5InnoDBialect"/>
13
14      <property name="hibernate.hbm2ddl.auto" value="update"/>
15
16      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
17
18      <property name="javax.persistence.jdbc.user" value="root"/>
19
20      <property name="javax.persistence.jdbc.password" value="root"/>
21
22      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/←
23        K19-DB"/>
24    </properties>
25  </persistence-unit>
26 </persistence>

```

Código XML 13.2: persistence.xml

- 5 Abra um terminal; entre no cliente do MySQL Server; apague se existir a base de dados K19-DB; e crie uma base de dados nova chamada K19-DB.

```

k19@k19-11:~/rafael$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 36
Server version: 5.1.58-1ubuntu1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> DROP DATABASE IF EXISTS 'K19-DB';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> CREATE DATABASE 'K19-DB';
Query OK, 1 row affected (0.02 sec)

```

- 6 Crie um pacote chamado filters na pasta src do projeto K19-Integracao-JSF-JPA.
- 7 No pacote filters, crie uma classe chamada JPAFilter com o seguinte conteúdo:

```

1 package filters;
2
3 import java.io.IOException;
4 import javax.persistence.*;
5 import javax.servlet.*;
6 import javax.servlet.annotation.WebFilter;
7

```

```

8  @WebFilter(servletNames={"Faces Servlet"})
9  public class JPAFilter implements Filter {
10
11    private EntityManagerFactory factory;
12
13    @Override
14    public void init(FilterConfig filterConfig) throws ServletException {
15      this.factory = Persistence.createEntityManagerFactory("K19-PU");
16    }
17
18    @Override
19    public void destroy() {
20      this.factory.close();
21    }
22
23    @Override
24    public void doFilter(ServletRequest request, ServletResponse response,
25      FilterChain chain) throws IOException, ServletException {
26
27      // CHEGADA
28      EntityManager manager = this.factory.createEntityManager();
29      request.setAttribute("EntityManager", manager);
30      manager.getTransaction().begin();
31      // CHEGADA
32
33      // FACES SERVLET
34      chain.doFilter(request, response);
35      // FACES SERVLET
36
37      // SAÍDA
38      try {
39        manager.getTransaction().commit();
40      } catch (Exception e) {
41        manager.getTransaction().rollback();
42      } finally {
43        manager.close();
44      }
45      // SAÍDA
46    }
47  }

```

Código Java 13.5: JPAFilter.java

- 8 Crie um pacote chamado model na pasta src do projeto K19-Integracao-JSF-JPA.
- 9 No pacote model, crie uma classe chamada Carro com o seguinte conteúdo:

```

1 package model;
2
3 @Entity
4 public class Carro {
5
6   @Id @GeneratedValue
7   private Long id;
8   private String marca;
9   private String modelo;
10
11   // GETTERS E SETTERS
12 }

```

Código Java 13.6: Carro.java

- 10 No pacote model, crie uma classe chamada CarroRepository com o seguinte conteúdo:

```

1 package model;
2
3 public class CarroRepository {

```

```

4   private EntityManager manager;
5
6   public CarroRepository(EntityManager manager) {
7     this.manager = manager;
8   }
9
10  public void adiciona(Carro carro) {
11    this.manager.persist(carro);
12  }
13
14  public List<Carro> buscaTodos() {
15    Query query = this.manager.createQuery("select x from Carro x");
16    return query.getResultList();
17  }
18}
19

```

Código Java 13.7: CarroRepository.java

- 11** Crie um pacote chamado `managedbeans` na pasta `src` do projeto K19-Integracao-JSF-JPA.
- 12** No pacote `managedbeans`, crie uma classe chamada `CarroBean` com o seguinte código:

```

1 package managedbeans;
2
3 @ManagedBean
4 public class CarroBean {
5
6   private Carro carro = new Carro();
7
8   public void adicionaCarro() {
9     EntityManager manager = this.getEntityManager();
10    CarroRepository repository = new CarroRepository(manager);
11
12    repository.adiciona(this.carro);
13    this.carro = new Carro();
14  }
15
16  public List<Carro> getCarros() {
17    EntityManager manager = this.getEntityManager();
18    CarroRepository repository = new CarroRepository(manager);
19    return repository.buscaTodos();
20  }
21
22  private EntityManager getEntityManager() {
23    FacesContext fc = FacesContext.getCurrentInstance();
24    ExternalContext ec = fc.getExternalContext();
25    HttpServletRequest request = (HttpServletRequest) ec.getRequest();
26    EntityManager manager = (EntityManager)request.getAttribute("EntityManager");
27
28    return manager;
29  }
30  // GETTERS E SETTERS
31}

```

Código Java 13.8: CarroBean.java

- 13** No diretório `WebContent`, crie um arquivo XHTML chamado `carros.xhtml` com o seguinte conteúdo:

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"

```

```

7   xmlns:ui="http://java.sun.com/jsf/facelets">
8
9 <h:head>
10    <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13   <h:form>
14     <h:panelGrid columns="2">
15       <h:outputLabel value="Marca: " for="campo-marca" />
16       <h:inputText value="#{carroBean.carro.marca}" id="campo-marca" />
17
18       <h:outputLabel value="Modelo: " for="campo-modelo" />
19       <h:inputText value="#{carroBean.carro.modelo}" id="campo-modelo" />
20
21       <h:commandButton value="Adicionar" action="#{carroBean.adicionaCarro}" />
22     </h:panelGrid>
23   </h:form>
24 </h:body>
25 </html>

```

Código XHTML 13.1: carros.xhtml

- 14** No arquivo carros.xhtml, adicione também um trecho de código para exibir os carros adicionados.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8
9 <h:head>
10   <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13   <h:form>
14     <h:panelGrid columns="2">
15       <h:outputLabel value="Marca: " for="campo-marca" />
16       <h:inputText value="#{carroBean.carro.marca}" id="campo-marca" />
17
18       <h:outputLabel value="Modelo: " for="campo-modelo" />
19       <h:inputText value="#{carroBean.carro.modelo}" id="campo-modelo" />
20
21       <h:commandButton value="Adicionar" action="#{carroBean.adicionaCarro}" />
22     </h:panelGrid>
23   </h:form>
24
25   <h:panelGroup rendered="#{not empty carroBean.carros}">
26     <h1>Lista de carros:</h1>
27     <ul>
28       <ui:repeat value="#{carroBean.carros}" var="carro">
29         <li><h:outputText value="#{carro.marca} #{carro.modelo}" /></li>
30       </ui:repeat>
31     </ul>
32   </h:panelGroup>
33 </h:body>
34 </html>

```

Código XHTML 13.2: carros.xhtml

- 15** Acesse a aplicação no endereço:

<http://localhost:8080/K19-Integracao-JSF-JPA/carros.xhtml>

Adicione alguns carros e verifique se eles foram adicionados no SGDB.

Otimizando o número de consultas ao SGDB

Durante o processamento de uma requisição, os métodos *getters* são chamados pelo JSF. A quantidade de chamadas não pode ser controlada pela aplicação. A lógica implementada nos *getters* pode comprometer a performance da aplicação. Por exemplo, nos exercícios anteriores, toda vez que o método `getCarros()` do managed bean `carroBean` é chamado, uma consulta é realizada no SGDB através do repositório de carros.

Podemos diminuir a quantidade de consultas armazenando o resultado da primeira chamada em um atributo da classe `CarroBean`. Esse resultado pode ser mantido até que alguma alteração na listagem de carros ocorra. Por exemplo, se um carro for adicionado ou removido, a listagem armazenada deve ser descartada e recuperada novamente do SGDB. Veja o código abaixo.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class CarroBean {
5
6     private Carro carro = new Carro();
7
8     private List<Carro> carros;
9
10    public void adicionaCarro() {
11        EntityManager manager = this.getEntityManager();
12        CarroRepository repository = new CarroRepository(manager);
13
14        repository.adiciona(this.carro);
15        this.carro = new Carro();
16        this.carros = null;
17    }
18
19    public List<Carro> getCarros() {
20        if(this.carros == null) {
21            EntityManager manager = this.getEntityManager();
22            CarroRepository repository = new CarroRepository(manager);
23            this.carros = repository.buscaTodos();
24        }
25
26        return this.carros;
27    }
28
29    private EntityManager getEntityManager() {
30        FacesContext fc = FacesContext.getCurrentInstance();
31        ExternalContext ec = fc.getExternalContext();
32        HttpServletRequest request = (HttpServletRequest) ec.getRequest();
33        EntityManager manager = (EntityManager)request.getAttribute("EntityManager");
34
35        return manager;
36    }
37    // GETTERS E SETTERS
38 }
```

Código Java 13.9: `CarroBean.java`

No código acima, o atributo `carros` armazena a última lista carregada do SGDB. No método `getCarros()`, verificamos se já existe uma lista carregada no atributo `carros`. Se não existir uma lista carregada então utilizamos o repositório de carros para recuperar os dados do SGDB e carregar a lista. Caso contrário, simplesmente utilizamos a lista que já estava carregada. No método

adicionaCarro(), depois que o repositório de carros é acionado para persistir um carro, descartamos o conteúdo do atributo carros. Dessa forma, na próxima chamada do método getCarros(), a lista será carregada novamente e conterá o carro que acabou de ser adicionado.



Exercícios de Fixação

- 16** Vamos monitorar a quantidade de chamadas ao método getCarros() do managed bean carroBean. Altere a classe desse managed bean da seguinte forma:

```

1 package managedbeans;
2
3 @ManagedBean
4 public class CarroBean {
5
6     private Carro carro = new Carro();
7
8     public void adicionaCarro() {
9         EntityManager manager = this.getEntityManager();
10        CarroRepository repository = new CarroRepository(manager);
11
12        repository.adiciona(this.carro);
13        this.carro = new Carro();
14    }
15
16    public List<Carro> getCarros() {
17        EntityManager manager = this.getEntityManager();
18        CarroRepository repository = new CarroRepository(manager);
19
20        System.out.println("CHAMANDO O REPOSITORIO");
21        return repository.buscaTodos();
22    }
23
24    private EntityManager getEntityManager() {
25        FacesContext fc = FacesContext.getCurrentInstance();
26        ExternalContext ec = fc.getExternalContext();
27        HttpServletRequest request = (HttpServletRequest) ec.getRequest();
28        EntityManager manager = (EntityManager)request.getAttribute("EntityManager");
29
30        return manager;
31    }
32    // GETTERS E SETTERS
33 }
```

Código Java 13.10: CarroBean.java

- 17** Acesse a página <http://localhost:8080/K19-Integracao-JSF-JPA/carros.xhtml> e depois verifique no console do Eclipse quantas vezes o repositório de carros foi acionado.
- 18** Altere o comportamento do managed bean carroBean para diminuir o número de chamadas ao repositório de carros.

```

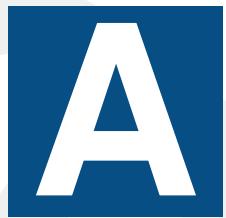
1 package managedbeans;
2
3 @ManagedBean
4 public class CarroBean {
5
6     private Carro carro = new Carro();
7
8     private List<Carro> carros;
9
10    public void adicionaCarro() {
```

```
11 EntityManager manager = this.getEntityManager();
12 CarroRepository repository = new CarroRepository(manager);
13
14     repository.adiciona(this.carro);
15     this.carro = new Carro();
16     this.carros = null;
17 }
18
19 public List<Carro> getCarros() {
20     if(this.carros == null) {
21         EntityManager manager = this.getEntityManager();
22         CarroRepository repository = new CarroRepository(manager);
23         System.out.println("CHAMANDO O REPOSITORIO");
24         this.carros = repository.buscaTodos();
25     }
26
27     return this.carros;
28 }
29
30 private EntityManager getEntityManager() {
31     FacesContext fc = FacesContext.getCurrentInstance();
32     ExternalContext ec = fc.getExternalContext();
33     HttpServletRequest request = (HttpServletRequest) ec.getRequest();
34     EntityManager manager = (EntityManager)request.getAttribute("EntityManager");
35
36     return manager;
37 }
38 // GETTERS E SETTERS
39 }
```

Código Java 13.11: CarroBean.java

- 19 Acesse novamente a página <http://localhost:8080/K19-Integracao-JSF-JPA/carros.xhtml> e depois verifique no console do Eclipse quantas vezes o repositório de carros foi acionado.





AUTENTICAÇÃO

Neste capítulo, apresentaremos uma maneira de implementar o processo de autenticação dos usuários de uma aplicação JSF.



Exercícios de Fixação

- 1 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Autenticacao** seguindo os passos vistos no exercício do Capítulo 5.
- 2 Por simplicidade, utilizaremos um atributo de um managed bean para armazenar os usuários da aplicação e suas respectivas senhas. A implementação que será apresentada a seguir pode ser alterada para que esses dados sejam armazenadas em um arquivo ou em um banco de dados.



Importante

Por motivos de segurança, as senhas dos usuários não devem ser armazenadas literalmente. Ao invés disso, as senhas dos usuários devem passar por um processo de transformação (criptografia) antes de serem armazenadas.

Quando um usuário tenta logar no sistema, ele digita o seu nome de usuário e sua senha. Para garantir que o usuário tenha acesso ao sistema, precisamos verificar se o nome de usuário digitado está cadastrado no sistema e se sua senha está correta. Como nós não armazenamos a senha do usuário, o que fazemos é aplicar a mesma transformação feita anteriormente e comparar o valor obtido com aquele armazenado no servidor. Se esses valores forem iguais, então permitimos que o usuário acesse o sistema. Caso contrário, o acesso ao sistema é negado.

Crie uma classe chamada AutenticadorBean em um pacote chamado `managedbeans` no projeto K19-Autenticacao com o seguinte conteúdo:

```

1 package managedbeans;
2
3 @ManagedBean
4 public class AutenticadorBean {
5
6     private static Map<String, String> mapa = new HashMap<String, String>();
7 }

```

Código Java A.1: AutenticadorBean.java

- 3 Acrescente alguns usuários e suas respectivas senhas no atributo mapa.

```

1 @ManagedBean
2 public class AutenticadorBean {

```

```

3     private static Map<String, String> mapa = new HashMap<String, String>();
4
5     static {
6         AutenticadorBean.mapa.put("k19", "k19");
7         AutenticadorBean.mapa.put("jonas.hirata", "jonas.hirata");
8         AutenticadorBean.mapa.put("marcelo.martins", "marcelo.martins");
9         AutenticadorBean.mapa.put("rafael.cosentino", "rafael.cosentino");
10    }
11 }
12 }
```

Código Java A.2: AutenticadorBean.java

- 4 Crie propriedades para armazenar os dados enviados através do formulário de identificação, um método para implementar o processo de autenticação e outro método para registrar a saída do usuário.

```

1  @ManagedBean
2  public class AutenticadorBean {
3
4      private static Map<String, String> mapa = new HashMap<String, String>();
5
6      private String usuario;
7
8      private String senha;
9
10     static {
11         AutenticadorBean.mapa.put("k19", "k19");
12         AutenticadorBean.mapa.put("jonas.hirata", "jonas.hirata");
13         AutenticadorBean.mapa.put("marcelo.martins", "marcelo.martins");
14         AutenticadorBean.mapa.put("rafael.cosentino", "rafael.cosentino");
15     }
16
17     public String autentica() {
18         FacesContext fc = FacesContext.getCurrentInstance();
19
20         if (AutenticadorBean.mapa.containsKey(this.usuario)
21             && AutenticadorBean.mapa.get(this.usuario).equals(this.senha)) {
22
23             ExternalContext ec = fc.getExternalContext();
24             HttpSession session = (HttpSession)ec.getSession(false);
25             session.setAttribute("usuario", this.usuario);
26
27             return "/home";
28         } else {
29             FacesMessage fm = new FacesMessage("usuário e/ou senha inválidos");
30             fm.setSeverity(FacesMessage.SEVERITY_ERROR);
31             fc.addMessage(null, fm);
32             return "/login";
33         }
34     }
35
36     public String registraSaida() {
37         FacesContext fc = FacesContext.getCurrentInstance();
38         ExternalContext ec = fc.getExternalContext();
39         HttpSession session = (HttpSession)ec.getSession(false);
40         session.removeAttribute("usuario");
41
42         return "/login";
43     }
44
45     // GETTERS E SETTERS
46 }
```

Código Java A.3: AutenticadorBean.java

O método autentica() verifica se os dados enviados pelo formulário de autenticação estão ca-

dastrados no mapa. Se esses dados estiverem cadastrados, registramos o nome do usuário na sessão HTTP e navegamos para a página principal da aplicação. Caso contrário, adicionamos no contexto do processamento da requisição uma mensagem de erro e navegamos para a página do formulário de autenticação. O método `registraSaida()` simplesmente retira da sessão HTTP o nome do usuário anteriormente autenticado.

- 5 Para testar, crie a tela do formulário de autenticação e a tela principal da aplicação.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14   <h:messages/>
15   <h:form>
16     <h:outputLabel value="Usuário: " for="campo-usuario" />
17     <h:inputText value="#{autenticadorBean.usuario}" id="campo-usuario" />
18
19     <h:outputLabel value="Senha: " for="campo-senha" />
20     <h:inputSecret value="#{autenticadorBean.senha}" id="campo-senha" />
21
22     <h:commandButton value="Entrar" action="#{autenticadorBean.autentica}"/>
23   </h:form>
24 </h:body>
25 </html>
```

Código XHTML A.1: *login.xhtml*

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14   Olá #{sessionScope.usuario}!
15   Você está logado.
16
17   <h:form>
18     <h:commandLink value="Sair" action="#{autenticadorBean.registraSaida}"/>
19   </h:form>
20 </h:body>
21 </html>
```

Código XHTML A.2: *home.xhtml*

Na tela principal da aplicação, utilizamos a variável `sessionScope` para recuperar o nome do

usuário autenticado.

- 6 Teste a aplicação acessando <http://localhost:8080/K19-Autenticacao/login.xhtml>
- 7 Somente usuários autenticados podem acessar a página principal da aplicação. Para controlar o acesso às páginas da aplicação, implemente um filtro para interceptar todas as requisições HTTP direcionadas à “Faces Servlet”.

Crie uma classe chamada ControleDeAcesso em um pacote chamado `filtros` no projeto K19-Autenticacao com o seguinte conteúdo:

```
1 @WebFilter(servletNames={"Faces Servlet"})
2 public class ControleDeAcesso implements Filter {
3
4     @Override
5     public void doFilter(ServletRequest request, ServletResponse response,
6             FilterChain chain) throws IOException, ServletException {
7
8         HttpServletRequest req = (HttpServletRequest) request;
9         HttpSession session = req.getSession();
10
11        if (session.getAttribute("usuario") != null
12            || req.getRequestURI().endsWith("login.xhtml")) {
13            chain.doFilter(request, response);
14        } else {
15            HttpServletResponse res = (HttpServletResponse) response;
16            res.sendRedirect("login.xhtml");
17        }
18    }
19
20    @Override
21    public void init(FilterConfig filterConfig) throws ServletException {
22    }
23
24    @Override
25    public void destroy() {
26    }
27
28 }
```

Código Java A.4: ControleDeAcesso.java

A anotação `@WebServlet` registra o filtro no Web Container. O atributo `servletNames` é utilizado para definir quais servlets serão interceptadas pelo filtro. O método `doFilter()` é chamado toda vez que uma requisição HTTP à servlet “Faces Servlet” é realizada. Esse método verifica se existe um usuário registrado na sessão HTTP ou se a página requisitada é a do formulário de autenticação. Se uma dessas condições for satisfeita, o filtro permite que o fluxo prossiga para a “Faces Servlet”. Caso contrário, ele redireciona o usuário para a página do formulário de autenticação.

- 8 Reinicie a aplicação e tente acessar diretamente a página principal da aplicação acessando <http://localhost:8080/K19-Autenticacao/home.xhtml>. Observe que a aplicação redireciona o navegador para a página do formulário de autenticação.

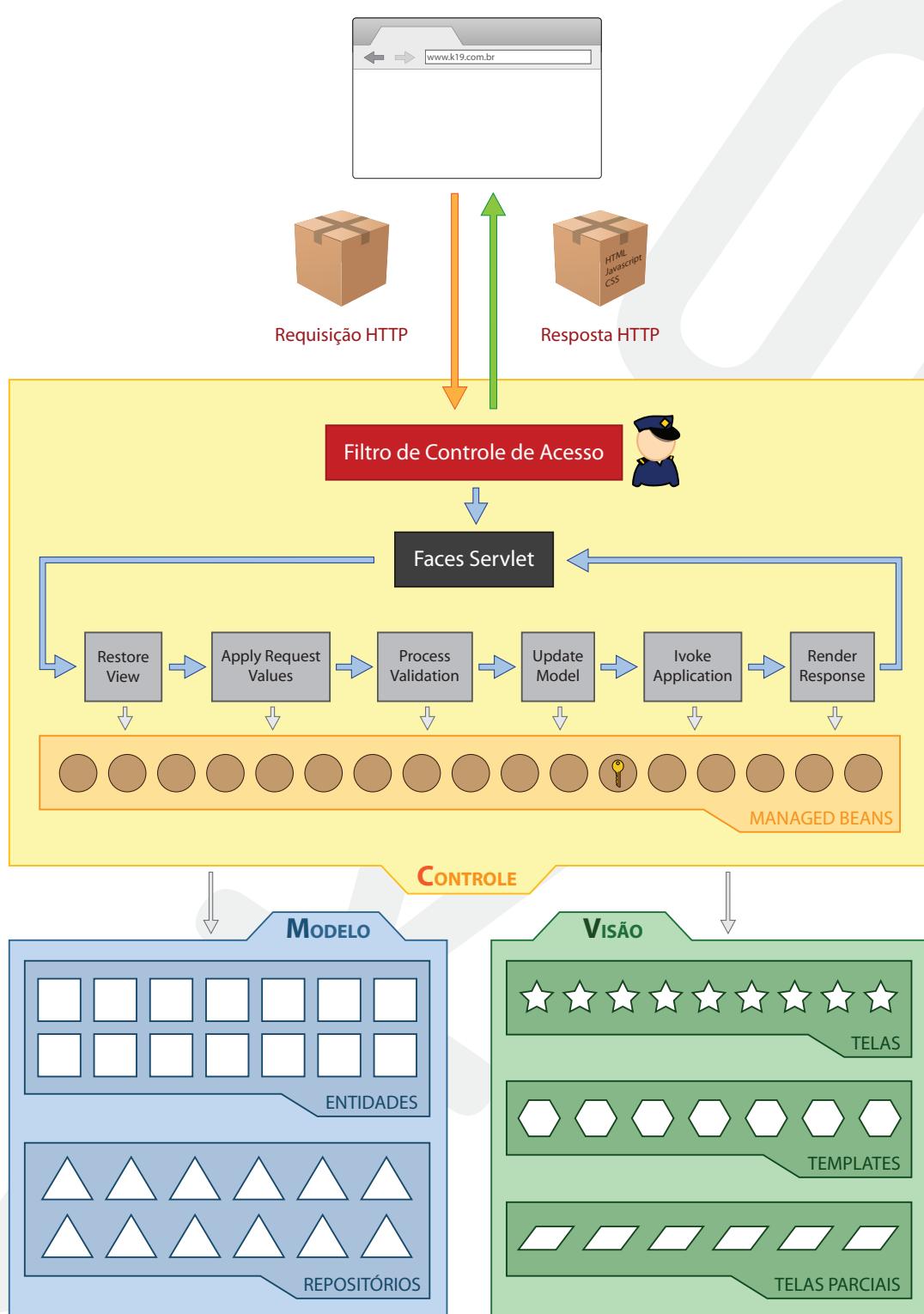


Figura A.1: Filtro e Managed Bean para implementar o controle de autenticação



PÁGINAS DE ERRO

Por padrão, quando determinados erros ocorrem no processamento de uma requisição, páginas com informações técnicas sobre o problema que ocorreu são geradas e enviadas para os usuários. Na fase de desenvolvimento, essas páginas são úteis para os desenvolvedores. Por outro lado, na fase de produção, essas páginas podem confundir os usuários da aplicação e revelar a estrutura do sistema, expondo possíveis falhas de segurança.



```

Caused by: java.lang.IllegalArgumentException: id to load is required for loading
    at org.hibernate.event.LoadEvent.<init>(LoadEvent.java:89)
    at org.hibernate.event.LoadEvent.<init>(LoadEvent.java:61)
    at org.hibernate.impl.SessionImpl.get(SessionImpl.java:994)
    at org.hibernate.impl.SessionImpl.get(SessionImpl.java:990)
    at org.hibernate.ejb.AbstractEntityManagerImpl.find(AbstractEntityManagerImpl.java:554)
    at org.hibernate.ejb.AbstractEntityManagerImpl.find(AbstractEntityManagerImpl.java:529)
    at org.hibernate.Query.procura(TimeRepository.java:38)
    at model.repositories.TimeRepository.procura(TimeRepository.java:30)
    at managedbeans.JogadorBean.adiciona(JogadorBean.java:30)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at com.sun.el.parser.AstValue.invoke(AstValue.java:234)
    at com.sun.el.MethodExpressionImpl.invoke(MethodExpressionImpl.java:297)
    at com.sun.faces.facelets.el.TagMethodExpression.invoke(TagMethodExpression.java:105)
    ... 41 more
  
```

Figura B.1: Expondo possíveis falhas de segurança

Neste capítulo, apresentaremos uma maneira de personalizar as páginas de erro da aplicação.



Exercícios de Fixação

- 1** Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Paginas-de-Erro** seguindo os passos vistos no exercício do Capítulo 5.
- 2** Criaremos uma página de erro padrão. Adicione na pasta *WebContent* um arquivo *XHTML* com o seguinte conteúdo.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14   <h1>Ocorreu um erro interno no sistema.</h1>
15   <h3>Tente novamente em alguns instantes.</h3>
  
```

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

Código XHTML B.1: pagina-de-erro.xhtml

- 3 Criaremos uma página com um botão que sempre produzirá um erro ao ser clicado. Adicione na pasta WebContent um arquivo XHTML com o seguinte conteúdo.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12
13 <h:body>
14 <h:form>
15 <h:commandButton value="Gera Erro" action="#{erroBean.geraErro}" />
16 </h:form>
17 </h:body>
18 </html>
```

Código XHTML B.2: gerador-de-erro.xhtml

- 4 Crie um managed bean que provoque propositalmente um erro. Adicione em um pacote chamado managedbeans uma classe com o seguinte código:

```
1 package managedbens;
2
3 @ManagedBean
4 public class ErroBean {
5     public void geraErro(){
6         System.out.println(10/0);
7     }
8 }
```

Código Java B.1: AutenticadorBean.java

- 5 Teste a aplicação acessando o seguinte endereço:

<http://localhost:8080/K19-Paginas-de-Erro/gerador-de-erro.xhtml>

- 6 Configure o Web Container para direcionar todas as exceptions para a página de erro padrão. Adicione o seguinte código no arquivo web.xml.

```
1 ...
2 <error-page>
3   <exception-type>java.lang.Exception</exception-type>
4   <location>/pagina-de-erro.xhtml</location>
5 </error-page>
6 ...
```

Código XML B.1: web.xml

- 7 Teste a aplicação novamente acessando o seguinte endereço:

<http://localhost:8080/K19-Paginas-de-Erro/gerador-de-erro.xhtml>



PROJETO FUTEBOL K19

Para consolidar o conhecimento obtido durante o treinamento, desenvolveremos uma aplicação completa para auxiliar no gerenciamento de um campeonato de futebol. Essa aplicação deverá permitir que os usuários possam cadastrar novos times e jogadores, bem como listá-los, alterá-los e removê-los. Essa aplicação também deverá possibilitar que jogadores sejam associados a times de futebol, respeitando-se a restrição de que cada jogador deve pertencer a no máximo um time de futebol.

Integração JSF e JPA

Criaremos uma aplicação web Java com JSF e adicionaremos os recursos do JPA nessa aplicação como visto no Capítulo 13.



Exercícios de Fixação

- 1 Entre na pasta K19-Arquivos/MySQL-Connector-JDBC da Área de Trabalho e copie o arquivo **mysql-connector-java-5.1.13-bin.jar** para a pasta glassfishv3/glassfish/lib também da sua Área de Trabalho. OBS: O Glassfish deve ser reiniciado para reconhecer o driver JDBC do MySQL.
- 2 Crie um projeto do tipo *Dynamic Web Project* chamado **K19-Futebol** seguindo os passos vistos no exercício do Capítulo 5.
- 3 Adicione uma pasta chamada META-INF na pasta src do projeto K19-Futebol.
- 4 Configure o JPA adicionando o arquivo **persistence.xml** na pasta META-INF.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.0"
3      xmlns="http://java.sun.com/xml/ns/persistence"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/←
6          ns/persistence/persistence_2_0.xsd">
7
8      <persistence-unit name="K19-Futebol-PU" transaction-type="RESOURCE_LOCAL">
9          <provider>org.hibernate.ejb.HibernatePersistence</provider>
10         <properties>
11             <property name="hibernate.dialect" value="org.hibernate.dialect.←
12                 MySQL5InnoDBialect"/>
13
14             <property name="hibernate.hbm2ddl.auto" value="update"/>
15
16             <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
17
18             <property name="javax.persistence.jdbc.user" value="root"/>
19             <property name="javax.persistence.jdbc.password" value="root"/>

```

```

19      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/←
20          K19-Futebol-DB"/>
21    </properties>
22  </persistence-unit>
23</persistence>
```

Código XML C.1: persistence.xml

- 5 Abra um terminal; entre no cliente do MySQL Server; apague se existir a base de dados K19-Futebol-DB; e crie uma base de dados nova chamada K19-Futebol-DB.

```

k19@k19-11:~/rafael$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 36
Server version: 5.1.58-1ubuntu1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> DROP DATABASE IF EXISTS 'K19-Futebol-DB';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> CREATE DATABASE 'K19-Futebol-DB';
Query OK, 1 row affected (0.02 sec)
```

- 6 Crie um pacote chamado filters na pasta src do projeto K19-Futebol.
 7 No pacote filters, crie uma classe chamada JPAFilter com o seguinte conteúdo:

```

1 package filters;
2
3 @WebFilter(servletNames={"Faces Servlet"})
4 public class JPAFilter implements Filter {
5
6     private EntityManagerFactory factory;
7
8     @Override
9     public void init(FilterConfig filterConfig) throws ServletException {
10         this.factory = Persistence.createEntityManagerFactory("K19-Futebol-PU");
11     }
12
13     @Override
14     public void destroy() {
15         this.factory.close();
16     }
17
18     @Override
19     public void doFilter(ServletRequest request, ServletResponse response,
20             FilterChain chain) throws IOException, ServletException {
21
22         // CHEGADA
23         EntityManager manager = this.factory.createEntityManager();
24         request.setAttribute("EntityManager", manager);
25         manager.getTransaction().begin();
26         // CHEGADA
27
28         // FACES SERVLET
29         chain.doFilter(request, response);
30         // FACES SERVLET
31
32         // SAÍDA
33         try {
34             manager.getTransaction().commit();
35         } catch (Exception e) {
```

```

36     manager.getTransaction().rollback();
37 } finally {
38     manager.close();
39 }
40 // SAÍDA
41 }
42 }
```

Código Java C.1: JPAFilter.java

Modelo

Na camada de modelo da nossa aplicação, implementaremos duas entidades: Jogador e Time. Além disso, criaremos os respectivos repositórios para implementar as diversas operações relacionadas à persistência dessas duas entidades.



Exercícios de Fixação

- 8 Na pasta `src` do projeto K19-Futebol, crie um pacote chamado `model`, depois crie um subpacote chamado `entities`.
- 9 No pacote `entities`, crie duas classes para modelar os times e jogadores.

```

1 package model.entities;
2
3 @Entity
4 public class Time {
5
6     @Id @GeneratedValue
7     private Long id;
8
9     private String nome;
10
11    private String tecnico;
12
13    // GETTERS E SETTERS
14 }
```

Código Java C.2: Time.java

```

1 package model.entities;
2
3 @Entity
4 public class Jogador {
5
6     @Id @GeneratedValue
7     private Long id;
8
9     private String nome;
10
11    private String posicao;
12
13    private Calendar dataDeNascimento = new GregorianCalendar();
14
15    @ManyToOne
16    private Time time;
17
18    // GETTERS E SETTERS
19 }
```

Código Java C.3: Jogador.java

- 10 Dentro do pacote model crie um pacote chamado repositories.
- 11 No pacote repositories, implemente as classes JogadorRepository e TimeRepository.

```
1 package model.repositories;
2
3 public class JogadorRepository {
4
5     private EntityManager manager;
6
7     public JogadorRepository(EntityManager manager) {
8         this.manager = manager;
9     }
10
11    public void adiciona(Jogador jogador) {
12        this.manager.persist(jogador);
13    }
14
15    public void remove(Long id) {
16        Jogador jogador = this.procura(id);
17        this.manager.remove(jogador);
18    }
19
20    public Jogador atualiza(Jogador jogador) {
21        return this.manager.merge(jogador);
22    }
23
24    public Jogador procura(Long id) {
25        return this.manager.find(Jogador.class, id);
26    }
27
28    public List<Jogador> getLista() {
29        Query query = this.manager.createQuery("select x from Jogador x");
30        return query.getResultList();
31    }
32 }
```

Código Java C.4: JogadorRepository.java

```
1 package model.repositories;
2
3 public class TimeRepository {
4
5     private EntityManager manager;
6
7     public TimeRepository(EntityManager manager) {
8         this.manager = manager;
9     }
10
11    public void adiciona(Time time) {
12        this.manager.persist(time);
13    }
14
15    public void remove(Long id) {
16        Time time = this.procura(id);
17        Query query = this.manager.createQuery("select x from Jogador x");
18        List<Jogador> jogadores = query.getResultList();
19        for (Jogador jogador : jogadores) {
20            jogador.setTime(null);
21        }
22        this.manager.remove(time);
23    }
24 }
```

```

25     public Time atualiza(Time time) {
26         return this.manager.merge(time);
27     }
28
29     public Time procura(Long id) {
30         return this.manager.find(Time.class, id);
31     }
32
33     public List<Time> getLista() {
34         Query query = this.manager.createQuery("select x from Time x");
35         return query.getResultList();
36     }
37 }
```

Código Java C.5: TimeRepository.java

Managed Beans

Na camada de controle, implementaremos managed beans para controlar as operações relacionadas às entidades Jogador e Time.



Exercícios de Fixação

- 12 Na pasta src, crie um pacote chamado managedbeans.
- 13 No pacote managedbeans, implemente a classe TimeBean.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class TimeBean {
5
6     private Time time = new Time();
7     private List<Time> times;
8
9     public void adiciona() {
10        EntityManager manager = this.getManager();
11        TimeRepository repository = new TimeRepository(manager);
12        if (this.time.getId() == null) {
13            repository.adiciona(this.time);
14        } else {
15            repository.atualiza(this.time);
16        }
17        this.time = new Time();
18        this.times = null;
19    }
20
21    public void preparaAlteracao() {
22        Map<String, String> params =
23            FacesContext.getCurrentInstance().getExternalContext().←
24            getRequestParameterMap();
25        Long id = Long.parseLong(params.get("id"));
26        EntityManager manager = this.getManager();
27        TimeRepository repository = new TimeRepository(manager);
28        this.time = repository.procura(id);
29    }
30
31    public void remove() {
32        Map<String, String> params =
33            FacesContext.getCurrentInstance().getExternalContext().←
```

```
33         getRequestParameterMap();
34     Long id = Long.parseLong(params.get("id"));
35     EntityManager manager = this.getManager();
36     TimeRepository repository = new TimeRepository(manager);
37     repository.remove(id);
38     this.times = null;
39 }
40
41 public List<Time> getTimes() {
42     if (this.times == null) {
43         EntityManager manager = this.getManager();
44         TimeRepository repository = new TimeRepository(manager);
45         this.times = repository.getList();
46     }
47     return this.times;
48 }
49
50 private EntityManager getManager() {
51     FacesContext fc = FacesContext.getCurrentInstance();
52     ExternalContext ec = fc.getExternalContext();
53     HttpServletRequest request = (HttpServletRequest) ec.getRequest();
54     return (EntityManager) request.getAttribute("EntityManager");
55 }
56
57 // GETTERS E SETTERS
58 }
```

Código Java C.6: TimeBean.java

- 14 No pacote managedbeans, implemente a classe JogadorBean.

```
1 package managedbeans;
2
3 @ManagedBean
4 public class JogadorBean {
5
6     private Jogador jogador = new Jogador();
7
8     private Long timeID;
9
10    private List<Jogador> jogadores;
11
12    public void adiciona() {
13        EntityManager manager = this.getManager();
14        TimeRepository timeRepository = new TimeRepository(manager);
15        JogadorRepository jogadorRepository = new JogadorRepository(manager);
16
17        if(this.timeID != null) {
18            Time time = timeRepository.procura(this.timeID);
19            this.jogador.setTime(time);
20        }
21
22        if (this.jogador.getId() == null) {
23            jogadorRepository.adiciona(this.jogador);
24
25        } else {
26            jogadorRepository.atualiza(this.jogador);
27        }
28
29        this.jogador = new Jogador();
30        this.jogadores = null;
31    }
32
33    public void preparaAlteracao() {
34        Map<String, String> params =
35            FacesContext.getCurrentInstance().getExternalContext().  
←
36            getRequestParameterMap();
37        Long id = Long.parseLong(params.get("id"));
38        EntityManager manager = this.getManager();
```

```

38     JogadorRepository repository = new JogadorRepository(manager);
39     this.jogador = repository.procura(id);
40 }
41
42 public void remove() {
43     Map<String, String> params =
44         FacesContext.getCurrentInstance().getExternalContext(). $\leftarrow$ 
45         getRequestParamMap();
46     Long id = Long.parseLong(params.get("id"));
47     EntityManager manager = this.getManager();
48     JogadorRepository repository = new JogadorRepository(manager);
49     repository.remove(id);
50     this.jogadores = null;
51 }
52
53 public List<Jogador> getJogadores() {
54     if (this.jogadores == null) {
55         EntityManager manager = this.getManager();
56         JogadorRepository repository = new JogadorRepository(manager);
57         this.jogadores = repository.getLista();
58     }
59     return this.jogadores;
60 }
61
62 private EntityManager getManager() {
63     FacesContext fc = FacesContext.getCurrentInstance();
64     ExternalContext ec = fc.getExternalContext();
65     HttpServletRequest request = (HttpServletRequest) ec.getRequest();
66     return (EntityManager) request.getAttribute("EntityManager");
67 }
68
69 // GETTERS E SETTERS
}

```

Código Java C.7: JogadorBean.java

Telas

Vamos implementar a camada de apresentação da nossa aplicação. Criaremos algumas telas usando templates e telas parciais.



Exercícios de Fixação

- 15** Na pasta WebContent, crie um diretório chamado resources. Nesse diretório, crie diretórios chamados css e imagens.
- 16** No diretório resources/css, crie um arquivo chamado style.css com o seguinte conteúdo:

```

1 body {
2     font: 18px Arial;
3     margin: 0 0 0 0;
4     background-color:#EEEEEE;
5 }
6
7 h1 {
8     margin: 0;
9     padding: 16px 0;
10    color: #000000;
11    font: bold 40px Arial;
12 }

```

```
13 h2 {  
14     font: bold 24px Arial;  
15     color: #595959;  
16 }  
17  
18 a {  
19     color: #3a34c3;  
20     text-decoration: none;  
21 }  
22  
23 a:hover {  
24     text-decoration: underline;  
25 }  
26  
27 .dados {  
28     border-collapse: collapse;  
29 }  
30  
31 .dados td,.dados th {  
32     border: 1px solid #0000c6;  
33     padding: 3px 7px 2px 7px;  
34 }  
35  
36 .dados th {  
37     padding-top: 5px;  
38     padding-bottom: 4px;  
39     background-color: #156bbd;  
40     color: #ffffff;  
41 }  
42  
43 .dados tr.impar td {  
44     color: #000000;  
45     background-color: #d4d4d9;  
46 }  
47  
48 .dados tr.par td {  
49     color: #000000;  
50     background-color: #f6f6fa;  
51 }  
52  
53 input {  
54     font-size: 16px;  
55     border: 1px solid #0000c6;  
56     background-color: #f6f6fa;  
57 }  
58  
59 #header {  
60     color: #FFFFFF;  
61     background-color: #252525;  
62     padding: 10px 0px 28px 10px;  
63 }  
64  
65 #header a {  
66     color: #FFFFFF;  
67     font: bold 24px Arial;  
68 }  
69  
70 #conteudo {  
71     margin: 0px 0px 50px 10px;  
72 }  
73 }
```

Código CSS C.1: style.css

- 17 Copie o arquivo k19-logo.png da pasta K19-Arquivos da Área de Trabalho para a pasta resources/imagens.

- 18) Na pasta WEB-INF, crie um diretório chamado templates.
- 19) No diretório WEB-INF/templates, crie um arquivo chamado template.xhtml com o seguinte conteúdo:

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9  <h:head>
10   <title>K19 Futebol</title>
11 </h:head>
12
13 <h:body>
14   <h:outputStylesheet library="css" name="style.css" />
15
16   <div id="header">
17     <h:form style="float:left;">
18       <h:link value="Times" outcome="times" />
19       &nbsp;
20       <h:link value="Jogadores" outcome="jogadores" />
21     </h:form>
22     <h:graphicImage library="imagens" name="k19-logo.png" style="float:right;" />
23     <div style="clear:both"></div>
24   </div>
25
26   <div id="conteudo">
27     <ui:insert name="conteudo"> Espaço para o conteúdo da tela </ui:insert>
28   </div>
29
30   <div id="footer" style="text-align: center">
31     <hr />
32     &copy; 2012 K19. Todos os direitos reservados.
33   </div>
34 </h:body>
35 </html>
```

Código XHTML C.1: template.xhtml

- 20) Na pasta WebContent, crie um arquivo chamado times.xhtml com o conteúdo abaixo.

```

1  <ui:composition template="/WEB-INF/templates/template.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core">
6
7  <ui:define name="conteudo">
8
9    <h1>Times</h1>
10
11   <h:form>
12     <ui:include src="/formulario-novo-time.xhtml" />
13     <ui:include src="/lista-de-times.xhtml" />
14   </h:form>
15 </ui:define>
16 </ui:composition>
```

Código XHTML C.2: times.xhtml

- 21 Também na pasta WebContent, crie as telas parciais definidas pelos arquivos `formulario-novo-time.xhtml` e `lista-de-times.xhtml`.

```

1 <ui:composition
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core">
6
7   <h:inputHidden value="#{timeBean.time.id}" />
8   <h:panelGrid columns="3">
9     <h:outputLabel value="Nome: " for="nome" />
10    <h:inputText
11      id="nome"
12      required="true"
13      requiredMessage="O nome do time é obrigatório"
14      value="#{timeBean.time.nome}">
15    </h:inputText>
16    <h:message for="nome" />
17
18    <h:outputLabel value="Técnico: " for="technico" />
19    <h:inputText
20      id="technico"
21      required="true"
22      requiredMessage="O nome do técnico é obrigatório"
23      value="#{timeBean.time.technico}">
24    </h:inputText>
25    <h:message for="technico" />
26
27    <h:commandButton value="Cadastrar" >
28      <f:ajax
29        event="click"
30        execute="@form"
31        listener="#{timeBean.adiciona}"
32        render="@all" />
33    </h:commandButton>
34  </h:panelGrid>
35 </ui:composition>
```

Código XHTML C.3: `formulario-novo-time.xhtml`

```

1 <ui:composition
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core">
6
7   <h:panelGroup rendered="#{not empty timeBean.times}">
8     <h2>Lista de Times</h2>
9
10    <h:dataTable
11      id="tabela"
12      value="#{timeBean.times}"
13      var="time"
14      styleClass="dados"
15      rowClasses="par,impar">
16
17      <h:column>
18        <f:facet name="header">
19          <h:outputText value="Id" />
20        </f:facet>
21        #{time.id}
22      </h:column>
23
24      <h:column>
25        <f:facet name="header">
26          <h:outputText value="Nome" />
27        </f:facet>
```

```

28      #{time.nome}
29  </h:column>
30
31  <h:column>
32    <f:facet name="header">
33      <h:outputText value="Técnico" />
34    </f:facet>
35    #{time.tecnico}
36  </h:column>
37
38  <h:column>
39    <f:facet name="header">
40      <h:outputText value="Alterar" />
41    </f:facet>
42
43    <f:ajax
44      event="click"
45      render="@form"
46      listener="#{timeBean.preparaAlteracao}">
47      <h:commandLink>
48        <f:param name="id" value="#{time.id}" />
49        Alterar
50      </h:commandLink>
51    </f:ajax>
52  </h:column>
53
54  <h:column>
55    <f:facet name="header">
56      <h:outputText value="Remover" />
57    </f:facet>
58
59    <f:ajax
60      event="click"
61      render="@form"
62      listener="#{timeBean.remove}">
63      <h:commandLink>
64        <f:param name="id" value="#{time.id}" />
65        Remover
66      </h:commandLink>
67    </f:ajax>
68  </h:column>
69  </h:dataTable>
70</h:panelGroup>
71</ui:composition>
```

Código XHTML C.4: lista-de-times.xhtml

- 22 Na pasta WebContent, crie um arquivo chamado jogadores.xhtml com o conteúdo abaixo.

```

1  <ui:composition template="/WEB-INF/templates/template.xhtml"
2    xmlns="http://www.w3.org/1999/xhtml"
3    xmlns:ui="http://java.sun.com/jsf/facelets"
4    xmlns:h="http://java.sun.com/jsf/html"
5    xmlns:f="http://java.sun.com/jsf/core">
6
7  <ui:define name="conteudo">
8
9    <h1>Jogadores</h1>
10
11   <h:form>
12     <ui:include src="/formulario-novo-jogador.xhtml" />
13     <ui:include src="/lista-de-jogadores.xhtml" />
14   </h:form>
15 </ui:define>
16 </ui:composition>
```

Código XHTML C.5: jogadores.xhtml

- 23** Ainda na pasta WebContent, crie as telas parciais definidas pelos arquivos `formulario-novo-jogador.xhtml` e `lista-de-jogadores.xhtml`.

```

1 <ui:composition
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core">
6
7   <h:inputHidden value="#{jogadorBean.jogador.id}" />
8   <h:panelGrid columns="3">
9     <h:outputLabel value="Nome: " for="jogador-nome" />
10    <h:inputText
11      id="jogador-nome"
12      required="true"
13      requiredMessage="O nome do jogador é obrigatório"
14      value="#{jogadorBean.jogador.nome}">
15    </h:inputText>
16    <h:message for="jogador-nome" />
17
18    <h:outputLabel value="Posição: " for="jogador-posicao" />
19    <h:inputText
20      id="jogador-posicao"
21      required="true"
22      requiredMessage="A posição do jogador deve ser especificada"
23      value="#{jogadorBean.jogador.posicao}">
24    </h:inputText>
25    <h:message for="jogador-posicao" />
26
27    <h:outputLabel value="Data de Nascimento: " for="jogador-nascimento" />
28    <h:inputText
29      id="jogador-nascimento"
30      required="true"
31      requiredMessage="Informe a data de nascimento do jogador"
32      value="#{jogadorBean.jogador.dataDeNascimento.time}">
33      <f:convertDateTime pattern="dd/MM/yyyy" />
34    </h:inputText>
35    <h:message for="jogador-nascimento" />
36
37    <h:outputLabel value="Time: " for="jogador-time" />
38    <h:selectOneMenu id="jogador-time" value="#{jogadorBean.timeID}">
39      <f:selectItems
40        value="#{timeBean.times}"
41        var="time"
42        itemLabel="#{time.nome}"
43        itemValue="#{time.id}" />
44    </h:selectOneMenu>
45    <h:message for="jogador-time" />
46
47    <h:commandButton value="Cadastrar" styleClass="botao-formulario">
48      <f:ajax
49        event="click"
50        execute="@form"
51        listener="#{jogadorBean.adiciona}"
52        render="@form" />
53    </h:commandButton>
54  </h:panelGrid>
55 </ui:composition>
```

Código XHTML C.6: `formulario-novo-jogador.xhtml`

```

1 <ui:composition
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:ui="http://java.sun.com/jsf/facelets"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core">
6
7   <h:panelGroup rendered="#{not empty jogadorBean.jogadores}">
```

```

8 <h2>Lista de Jogadores</h2>
9
10 <h: dataTable
11   id="tabela"
12   value="#{jogadorBean.jogadores}"
13   var="jogador"
14   styleClass="dados" rowClasses="par,impar">
15
16   <h: column>
17     <f: facet name="header">
18       <h: outputText value="Id" />
19     </f: facet>
20     #{jogador.id}
21   </h: column>
22
23   <h: column>
24     <f: facet name="header">
25       <h: outputText value="Nome" />
26     </f: facet>
27     #{jogador.nome}
28   </h: column>
29
30   <h: column>
31     <f: facet name="header">
32       <h: outputText value="Posição" />
33     </f: facet>
34     #{jogador.posicao}
35   </h: column>
36
37   <h: column>
38     <f: facet name="header">
39       <h: outputText value="Data de Nascimento" />
40     </f: facet>
41
42     <h: outputText value="#{jogador.dataDeNascimento.time}">
43       <f: convertDateTime pattern="dd/MM/yyyy"/>
44     </h: outputText>
45   </h: column>
46
47   <h: column>
48     <f: facet name="header">
49       <h: outputText value="Time" />
50     </f: facet>
51     #{jogador.time.nome}
52   </h: column>
53
54   <h: column>
55     <f: facet name="header">
56       <h: outputText value="Alterar" />
57     </f: facet>
58
59     <f: ajax
60       event="click"
61       render="@form"
62       listener="#{jogadorBean.preparaAlteracao}">
63       <h: commandLink>
64         <f: param name="id" value="#{jogador.id}" />
65         Alterar
66       </h: commandLink>
67     </f: ajax>
68   </h: column>
69
70   <h: column>
71     <f: facet name="header">
72       <h: outputText value="Remover" />
73     </f: facet>
74
75     <f: ajax
76       event="click"
77       render="@form"

```

```

78     listener="#{jogadorBean.remove}">
79     <h:commandLink>
80       <f:param name="id" value="#{jogador.id}" />
81       Remover
82     </h:commandLink>
83     </f:ajax>
84   </h:column>
85 </h:dataTable>
86 </h:panelGroup>
87 </ui:composition>

```

Código XHTML C.7: lista-de-jogadores.xhtml

- 24 Acesse a aplicação no endereço:

<http://localhost:8080/K19-Futebol/times.xhtml>

Autenticação

Vamos acrescentar um controle de segurança à nossa aplicação como vimos no Apêndice A.

- 25 Na pasta src/managedbeans, crie um managed bean de autenticação.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class AutenticadorBean {
5
6   private static Map<String, String> mapa = new HashMap<String, String>();
7
8   private String usuario;
9
10  private String senha;
11
12  static {
13    AutenticadorBean.mapa.put("k19", "k19");
14    AutenticadorBean.mapa.put("jonas.hirata", "jonas.hirata");
15    AutenticadorBean.mapa.put("marcelo.martins", "marcelo.martins");
16    AutenticadorBean.mapa.put("rafael.cosentino", "rafael.cosentino");
17  }
18
19  public String autentica() {
20    FacesContext fc = FacesContext.getCurrentInstance();
21
22    if (AutenticadorBean.mapa.containsKey(this.usuario)
23        && AutenticadorBean.mapa.get(this.usuario).equals(this.senha)) {
24
25      ExternalContext ec = fc.getExternalContext();
26      HttpSession session = (HttpSession)ec.getSession(false);
27      session.setAttribute("usuario", this.usuario);
28
29      return "/times";
30    } else {
31      FacesMessage fm = new FacesMessage("usuário e/ou senha inválidos");
32      fm.setSeverity(FacesMessage.SEVERITY_ERROR);
33      fc.addMessage(null, fm);
34      return "/login";
35    }
36  }
37
38  public String registraSaida() {
39    FacesContext fc = FacesContext.getCurrentInstance();
40    ExternalContext ec = fc.getExternalContext();
41    HttpSession session = (HttpSession)ec.getSession(false);

```

```

42     session.removeAttribute("usuario");
43
44     return "/login";
45 }
46
47 // GETTERS E SETTERS
48 }
```

Código Java C.8: AutenticadorBean.java

- 26 Na pasta filters, implemente um filtro para o controle de acesso.

```

1 package filters;
2
3 @WebFilter(servletNames={"Faces Servlet"})
4 public class ControleDeAcesso implements Filter {
5
6     @Override
7     public void doFilter(ServletRequest request, ServletResponse response,
8             FilterChain chain) throws IOException, ServletException {
9
10    HttpServletRequest req = (HttpServletRequest) request;
11    HttpSession session = req.getSession();
12
13    if (session.getAttribute("usuario") != null
14        || req.getRequestURI().endsWith("login.xhtml")) {
15        chain.doFilter(request, response);
16    } else {
17        HttpServletResponse res = (HttpServletResponse) response;
18        res.sendRedirect("login.xhtml");
19    }
20 }
21
22     @Override
23     public void init(FilterConfig filterConfig) throws ServletException {
24 }
25
26     @Override
27     public void destroy() {
28 }
29 }
30 }
```

Código Java C.9: ControleDeAcesso.java

- 27 Na pasta WebContent, crie um arquivo chamado login.xhtml e implemente o formulário de autenticação.

```

1 <ui:composition template="/WEB-INF/templates/template.xhtml"
2     xmlns="http://www.w3.org/1999/xhtml"
3     xmlns:ui="http://java.sun.com/jsf/facelets"
4     xmlns:h="http://java.sun.com/jsf/html"
5     xmlns:f="http://java.sun.com/jsf/core">
6
7 <ui:define name="conteudo">
8     <p>Preencha o formulário abaixo para entrar no sistema.</p>
9     <h:form>
10    <h:panelGrid columns="2">
11        <h:outputLabel value="Usuário: " for="campo-usuario" />
12        <h:inputText value="#{autenticadorBean.usuario}" id="campo-usuario" />
13
14        <h:outputLabel value="Senha: " for="campo-senha" />
15        <h:inputSecret value="#{autenticadorBean.senha}" id="campo-senha" />
16    </h:panelGrid>
17
18    <h:commandButton value="Entrar" action="#{autenticadorBean.autentica}"/>
```

```

19   </h:form>
20
21   <h:messages/>
22 </ui:define>
23 </ui:composition>
```

Código XHTML C.8: login.xhtml

- 28 No arquivo `template.xhtml`, acrescente um link para o usuário fazer logout. O menu de navegação deve ser exibido apenas se o usuário estiver logado.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html"
7   xmlns:f="http://java.sun.com/jsf/core">
8
9 <h:head>
10  <title>K19 Futebol</title>
11 </h:head>
12
13 <h:body>
14  <h:outputStylesheet library="css" name="style.css" />
15
16 <div id="header">
17  <h:form style="float:left;" rendered="#{not empty sessionScope.usuario}">
18    <h:link value="Times" outcome="times" />
19    &nbsp;
20    <h:link value="Jogadores" outcome="jogadores" />
21    &nbsp;
22    <h:commandLink value="Logout" action="#{autenticadorBean.registraSaida}" />
23  </h:form>
24  <h:graphicImage library="imagens" name="k19-logo.png" style="float:right;"/>
25  <div style="clear:both"></div>
26 </div>
27
28 <div id="conteudo">
29  <ui:insert name="conteudo"> Espaço para o conteúdo da tela </ui:insert>
30 </div>
31
32 <div id="footer" style="text-align: center">
33  <hr />
34  &copy; 2012 K19. Todos os direitos reservados.
35 </div>
36 </h:body>
37 </html>
```

Código XHTML C.9: template.xhtml

- 29 Acessa a aplicação no endereço:

`http://localhost:8080/K19-Futebol/login.xhtml`



RESPOSTAS

Resposta do Complementar 2.1

```

1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.PreparedStatement;
4 import java.util.Scanner;
5
6 public class InsereLivros {
7     public static void main(String[] args) {
8         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
9         String usuario = "root";
10        String senha = "root";
11
12        Scanner entrada = new Scanner(System.in);
13
14        try {
15            System.out.println("Abrindo conexão...");
16            Connection conexao =
17                DriverManager.getConnection(stringDeConexao, usuario, senha);
18
19            System.out.println("Digite o título do livro: ");
20            String titulo = entrada.nextLine();
21
22            System.out.println("Digite o preço do livro: ");
23            String preco = entrada.nextLine();
24
25            System.out.println("Digite o id da editora do livro: ");
26            String id = entrada.nextLine();
27
28            String sql = "INSERT INTO Livro (titulo, preco, editora_id) " +
29                "VALUES ('" + titulo + "', " + preco + ", " + id + ")";
30
31            PreparedStatement comando = conexao.prepareStatement(sql);
32
33            System.out.println("Executando comando...");
34            comando.execute();
35
36            System.out.println("Fechando conexão...");
37            conexao.close();
38        }
39        catch (Exception e) {
40            e.printStackTrace();
41        }
42    }
43}

```

Código Java 2.7: *InsereLivros.java*

Resposta do Complementar 2.3

```

1 import java.sql.Connection;
2 import java.sql.DriverManager;

```

```
3 import java.sql.PreparedStatement;
4 import java.util.Scanner;
5
6 public class InsereLivros {
7     public static void main(String[] args) {
8         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
9         String usuario = "root";
10        String senha = "root";
11
12        Scanner entrada = new Scanner(System.in);
13
14        try {
15            System.out.println("Abrindo conexão...");
16            Connection conexao =
17                DriverManager.getConnection(stringDeConexao, usuario, senha);
18
19            System.out.println("Digite o título do livro: ");
20            String titulo = entrada.nextLine();
21
22            System.out.println("Digite o preço do livro: ");
23            String preco = entrada.nextLine();
24
25            System.out.println("Digite o id da editora do livro: ");
26            String id = entrada.nextLine();
27
28            String sql = "INSERT INTO Livro (titulo, preco, editora_id) " +
29                "VALUES (?, ?, ?)";
30
31            PreparedStatement comando = conexao.prepareStatement(sql);
32            comando.setString(1, titulo);
33            comando.setString(2, preco);
34            comando.setString(3, id);
35
36            System.out.println("Executando comando...");
37            comando.execute();
38
39            System.out.println("Fechando conexão...");
40            conexao.close();
41        }
42        catch (Exception e) {
43            e.printStackTrace();
44        }
45    }
46}
```

Código Java 2.12: InsereLivros.java

Resposta do Complementar 2.5

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.PreparedStatement;
4 import java.sql.ResultSet;
5
6 public class ListaLivros {
7     public static void main(String[] args) {
8         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
9         String usuario = "root";
10        String senha = "root";
11
12        try {
13            System.out.println("Abrindo conexão...");
14            Connection conexao =
15                DriverManager.getConnection(stringDeConexao, usuario, senha);
```

```

17     String sql = "SELECT * FROM Livro;";
18
19     PreparedStatement comando = conexao.prepareStatement(sql);
20
21     System.out.println("Executando comando..."); 
22     ResultSet resultado = comando.executeQuery();
23
24     System.out.println("Resultados encontrados: \n");
25     while (resultado.next()) {
26         System.out.printf("ID: %d - TÍTULO: %s - PREÇO: %f - EDITORA: %d\n",
27             resultado.getInt("id"),
28             resultado.getString("titulo"),
29             resultado.getDouble("preco"),
30             resultado.getInt("editora_id"));
31     }
32
33     System.out.println("\nFechando conexão..."); 
34     conexao.close();
35 } catch (Exception e) {
36     e.printStackTrace();
37 }
38 }
39 }
```

Código Java 2.18: ListaLivros.java

Resposta do Complementar 2.6

```

1 import java.sql.Connection;
2 import java.sql.PreparedStatement;
3 import java.util.Scanner;
4
5 public class InsereLivros {
6     public static void main(String[] args) {
7         Scanner entrada = new Scanner(System.in);
8
9         try {
10             System.out.println("Abrindo conexão..."); 
11             Connection conexao = ConnectionFactory.createConnection();
12
13             System.out.println("Digite o título do livro: ");
14             String titulo = entrada.nextLine();
15
16             System.out.println("Digite o preço do livro: ");
17             String preco = entrada.nextLine();
18
19             System.out.println("Digite o id da editora do livro: ");
20             String id = entrada.nextLine();
21
22             String sql = "INSERT INTO Livro (titulo, preco, editora_id) " +
23                 "VALUES (?, ?, ?)";
24
25             PreparedStatement comando = conexao.prepareStatement(sql);
26             comando.setString(1, titulo);
27             comando.setString(2, preco);
28             comando.setString(3, id);
29
30             System.out.println("Executando comando..."); 
31             comando.execute();
32
33             System.out.println("Fechando conexão..."); 
34             conexao.close();
35 }
36 } catch (Exception e) {
37     e.printStackTrace();
38 }
```

```
38     }
39 }
40 }
```

Código Java 2.23: InsereLivros.java

```
1 import java.sql.Connection;
2 import java.sql.PreparedStatement;
3 import java.sql.ResultSet;
4
5 public class ListaLivros {
6     public static void main(String[] args) {
7         try {
8             System.out.println("Abrindo conexão...");
9             Connection conexao = ConnectionFactory.createConnection();
10
11             String sql = "SELECT * FROM Livro;";
12
13             PreparedStatement comando = conexao.prepareStatement(sql);
14
15             System.out.println("Executando comando...");
16             ResultSet resultado = comando.executeQuery();
17
18             System.out.println("Resultados encontrados: \n");
19             while (resultado.next()) {
20                 System.out.printf("ID: %d - TÍTULO: %s - PREÇO: %f - EDITORA: %d\n",
21                     resultado.getInt("id"),
22                     resultado.getString("titulo"),
23                     resultado.getDouble("preco"),
24                     resultado.getInt("editora_id"));
25             }
26
27             System.out.println("\nFechando conexão...");
28             conexao.close();
29         } catch (Exception e) {
30             e.printStackTrace();
31         }
32     }
33 }
```

Código Java 2.24: ListaLivros.java

Resposta do Complementar 6.1

No diretório src do projeto K19-Componentes-Visuais, crie uma classe chamada ContatoSACBean para implementar o managed bean que dará suporte ao formulário.

```
1 @ManagedBean
2 public class ContatoSACBean {
3     private String nome;
4     private String email;
5     private String codigoDeArea;
6     private String numeroDoTelefone;
7
8     private Integer numeroDoPedido;
9
10    private String assunto;
11    private String comentario;
12
13    private Boolean sexoFeminino;
14
15    private List<String> assuntos = new ArrayList<String>();
16
17    public ContatoSACBean() {
18        assuntos.add("Entrega");
19        assuntos.add("Pagamento");
```

```

20     assuntos.add("Trocas ou devoluções");
21     assuntos.add("Dúvidas gerais");
22     assuntos.add("Comentários");
23 }
24
25 // GETTERS E SETTERS
26 }
```

Código Java 6.8: ContatoSACBean.java

No diretório WebContent, crie um arquivo chamado contato-sac.xhtml e implemente o formulário de contato.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <h1>Serviço de Atendimento ao Consumidor</h1>
13
14   <h:form>
15     <h:panelGrid columns="2">
16       <h:outputLabel value="Nome: " for="campo-nome" />
17       <h:inputText value="#{contatoSACBean.nome}" id="campo-nome" />
18
19       <h:outputLabel value="E-mail: " for="campo-email" />
20       <h:inputText value="#{contatoSACBean.email}" id="campo-email" />
21
22       <h:outputLabel value="Telefone: " for="campo-telefone" />
23       <h:panelGroup>
24         (<h:inputText value="#{contatoSACBean.codigoDeArea}" id="campo-telefone"
25           size="2" />
26           <h:inputText value="#{contatoSACBean.numeroDoTelefone}" /> (DDD) XXXX-XXXX
27       </h:panelGroup>
28
29       <h:outputLabel value="Sexo: " for="campo-sexo" />
30       <h:selectOneRadio value="#{contatoSACBean.sexoFeminino}" id="campo-sexo">
31         <f:selectItem itemLabel="Feminino" itemValue="true" />
32         <f:selectItem itemLabel="Masculino" itemValue="false" />
33       </h:selectOneRadio>
34
35       <h:outputLabel value="Número do pedido: " for="campo-pedido" />
36       <h:inputText value="#{contatoSACBean.numeroDoPedido}" id="campo-pedido" />
37
38       <h:outputLabel value="Assunto" for="campo-assunto" />
39       <h:selectOneMenu value="#{contatoSACBean.assunto}">
40         <f:selectItem noSelectionOption="true" itemLabel="Escolha um assunto" />
41         <f:selectItems
42           value="#{contatoSACBean.assuntos}"
43           var="assunto"
44           itemLabel="#{assunto}"
45           itemValue="#{assunto}" />
46       </h:selectOneMenu>
47
48       <h:outputLabel value="Comentário: " for="campo-comentario" />
49       <h:inputTextarea value="#{contatoSACBean.comentario}" id="campo-comentario" />
50
51       <h:commandButton value="Enviar" />
52     </h:panelGrid>
53   </h:form>
54 </h:body>
55 </html>
```

Código XHTML 6.36: contato-sac.xhtml

Acesse a aplicação em

<http://localhost:8080/K19-Componentes-Visuais/contato-sac.xhtml>

Resposta do Complementar 6.2

Primeiramente, crie a classe Produto no diretório src do projeto K19-Componentes-Visuais.

```

1 public class Produto {
2     private String nome;
3     private Double preco;
4
5     public Produto(String nome, Double preco) {
6         this.nome = nome;
7         this.preco = preco;
8     }
9
10    // GETTERS E SETTERS
11 }
```

Código Java 6.12: Produto.java

Agora, crie um managed bean que guardará a lista de produtos disponíveis.

```

1 @ManagedBean
2 public class ProdutosBean {
3     private List<Produto> produtos = new ArrayList<Produto>();
4
5     public ProdutosBean() {
6         this.produtos.add(new Produto("Camiseta branca", 29.90));
7         this.produtos.add(new Produto("Camiseta preta", 39.99));
8         this.produtos.add(new Produto("Calça jeans", 95.99));
9         this.produtos.add(new Produto("Gravata", 19.90));
10        this.produtos.add(new Produto("Terno", 289.95));
11    }
12
13    // GETTER E SETTER
14 }
```

Código Java 6.13: ProdutosBean.java

Finalmente, crie o arquivo lista-de-produtos.xhtml na pasta WebContent e implemente a listagem dos produtos.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://java.sun.com/jsf/html"
6       xmlns:f="http://java.sun.com/jsf/core">
7
8 <h:head>
9     <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12     <h:dataTable value="#{produtosBean.produtos}" var="produto">
13         <f:facet name="header">Lista de produtos</f:facet>
14
15         <h:column>
```

```

16      <f:facet name="header">Nome</f:facet>
17      #{produto.nome}
18    </h:column>
19
20    <h:column>
21      <f:facet name="header">Preço</f:facet>
22      <h:outputFormat value="#{0, number, R$ .00}" >
23        <f:param value="#{produto.preco}" />
24      </h:outputFormat>
25    </h:column>
26  </h:dataTable>
27 </h:body>
28 </html>

```

Código XHTML 6.51: lista-de-produtos.xhtml

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Componentes-Visuais/lista-de-produtos.xhtml>

Resposta do Complementar 6.3

Na classe Produto, adicione um atributo booleano chamado mostraPreco.

```

1  public class Produto {
2    private String nome;
3    private Double preco;
4    private Boolean mostraPreco;
5
6    public Produto(String nome, Double preco) {
7      this.nome = nome;
8      this.preco = preco;
9      this.mostraPreco = true;
10   }
11
12   // GETTERS E SETTERS
13 }

```

Código Java 6.14: Produto.java

Modifique a tabela de forma a exibir somente os preços dos produtos para os quais a propriedade mostraPreco seja verdadeira. Use o atributo rendered para determinar quando um componente deve ser exibido.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7
8  <h:head>
9    <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <h:dataTable value="#{produtosBean.produtos}" var="produto">
13     <f:facet name="header">Lista de produtos</f:facet>
14
15     <h:column>
16       <f:facet name="header">Nome</f:facet>
17       #{produto.nome}
18     </h:column>
19
20     <h:column>

```

```

21      <f:facet name="header">Preço</f:facet>
22      <h:outputFormat value="{0, number, R$ .00}" rendered="#{produto.mostraPreco}" >
23          <f:param value="#{produto.preco}" />
24      </h:outputFormat>
25      <h:outputText value="Adicione o produto ao carrinho para ver o preço"
26          rendered="#{!produto.mostraPreco}" />
27      </h:column>
28  </h:dataTable>
29 </h:body>
30 </html>

```

Código XHTML 6.52: lista-de-produtos.xhtml

Defina como false o atributo mostraPreco de alguns produtos.

```

1 @ManagedBean
2 public class ProdutosBean {
3     private List<Produto> produtos = new ArrayList<Produto>();
4
5     public ProdutosBean() {
6         this.produtos.add(new Produto("Camiseta branca", 29.90));
7         this.produtos.add(new Produto("Camiseta preta", 39.99));
8         this.produtos.add(new Produto("Calça jeans", 95.99));
9         this.produtos.add(new Produto("Gravata", 19.90));
10        Produto produto = new Produto("Terno", 289.95);
11        produto.setMostraPreco(false);
12        this.produtos.add(produto);
13    }
14
15    // GETTER E SETTER
16 }

```

Código Java 6.15: ProdutosBean.java

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Componentes-Visuais/lista-de-produtos.xhtml>

Resposta do Complementar 6.4

Essa mensagem pode ser criada no método adicionaCurso() do managed bean cursosBean. Nesse método, adicione uma mensagem usando o método addMessage() da classe FacesContext.

```

1 @ManagedBean
2 @SessionScoped
3 public class CursosBean {
4     private List<Curso> cursos = new ArrayList<Curso>();
5     private Curso curso = new Curso();
6
7     public void adicionaCurso() {
8         this.cursos.add(this.curso);
9         this.curso = new Curso();
10        this.adicionaMensagemDeConfirmacao(curso);
11    }
12
13    private void adicionaMensagemDeConfirmacao(Curso curso) {
14        FacesMessage mensagem = new FacesMessage("O curso '" + curso.getSigla()
15            + " - " + curso.getNome() + "' foi adicionado");
16        FacesContext.getCurrentInstance().addMessage(
17            "form-adiciona:botao-adiciona", mensagem);
18    }
19
20    // GETTERS E SETTERS
21 }

```

Código Java 6.17: CursosBean.java

Modifique o arquivo `cursos.xhtml` para exibir a mensagem. Faça isso usando a tag `<h:message>`, como mostrado abaixo.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7
8 <h:head>
9   <title>K19 Treinamentos</title>
10 </h:head>
11 <h:body>
12   <h:form id="form-adiciona">
13     <h:panelGrid columns="2">
14       <h:outputLabel value="Sigla: " for="campo-sigla" />
15       <h:inputText value="#{cursosBean.curso.sigla}" id="campo-sigla" />
16
17       <h:outputLabel value="Nome: " for="campo-nome" />
18       <h:inputText value="#{cursosBean.curso.nome}" id="campo-nome" />
19
20       <h:commandButton value="Adicionar" action="#{cursosBean.adicionaCurso}"
21         id="botao-adiciona"/>
22     </h:panelGrid>
23   </h:form>
24
25   <p><h:message for="botao-adiciona" style="color: green;" /></p>
26
27   <h:dataTable value="#{cursosBean.cursos}" var="curso"
28     rendered="#{not empty cursosBean.cursos}">
29
30     <f:facet name="header">Lista de Cursos</f:facet>
31
32     <h:column>
33       <f:facet name="header">Sigla</f:facet>
34       #{curso.sigla}
35     </h:column>
36
37     <h:column>
38       <f:facet name="header">Nome</f:facet>
39       #{curso.nome}
40     </h:column>
41   </h:dataTable>
42 </h:body>
43 </html>

```

Código XHTML 6.59: cursos.xhtml

Acesse a aplicação em:

<http://localhost:8080/K19-Componentes-Visuais/cursos.xhtml>

Resposta do Complementar 7.1

No pacote `model`, crie uma classe chamada `Produto` para representar os produtos da loja.

```

1 package model;
2
3 public class Produto {

```

```

4     private String nome;
5     private String descricao;
6     private String caminhoDaFigura;
7     private double preco;
8     private int id;
9
10    public Produto(String nome, String descricao, String caminhoDaFigura,
11                  double preco, int id) {
12        this.nome = nome;
13        this.descricao = descricao;
14        this.caminhoDaFigura = caminhoDaFigura;
15        this.preco = preco;
16        this.id = id;
17    }
18
19
20    // GETTERS E SETTERS
21

```

Código Java 7.3: Produto.java

Na pasta WebContent do projeto, crie um diretório chamado `imagens`. Copie os arquivos da pasta K19-Arquivos/`imagens/loja-virtual` para o diretório WebContent/`imagens`.

No pacote `managedbeans`, crie uma classe chamada `ProdutosBean`. Essa classe deve armazenar uma lista de produtos e deve ter um atributo para guardar o produto cujos detalhes devem ser exibidos.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class ProdutosBean {
5
6     private Produto produtoSelecionado;
7     private int idDoProdutoSelecionado;
8     private List<Produto> produtos = new ArrayList<Produto>();
9
10    public ProdutosBean() {
11        this.produtos.add(new Produto(
12            "Camiseta branca",
13            "Esta é uma camiseta básica na cor branca. Ela é feita de algodão e " +
14            "está disponível nos tamanhos P, M e G.",
15            "/imagens/camiseta-branca.png", 29.90, 1));
16
17        this.produtos.add(new Produto(
18            "Camiseta preta",
19            "Esta é uma camiseta na cor preta, que está disponível " +
20            "nos tamanhos P, M e G. Seu material é 100% algodão.",
21            "/imagens/camiseta-preta.png", 39.90, 2));
22
23        this.produtos.add(new Produto(
24            "Calça jeans",
25            "Calça jeans disponível nos tamanhos 38 a 50.",
26            "/imagens/calca-jeans.png", 95.99, 3));
27
28        this.produtos.add(new Produto(
29            "Terno",
30            "Terno de seda na cor cinza.",
31            "/imagens/terno.png", 589.95, 4));
32
33        this.produtos.add(new Produto(
34            "Gravata",
35            "Gravata nas cores vermelha, azul e verde.",
36            "/imagens/gravatas.png", 19.90, 5));
37
38        this.produtoSelecionado = this.produtos.get(0);
39    }

```

```

40
41     public void carregaProduto() {
42         for (Produto produto : produtos) {
43             if (this.idDoProdutoSelecionado == produto.getId()) {
44                 this.produtoSelecionado = produto;
45                 break;
46             }
47         }
48     }
49
50     // GETTERS E SETTERS
51 }
```

Código Java 7.4: ProdutosBean.java

No diretório WEB-INF/templates, crie um arquivo chamado template-loja-virtual.xhtml para definir um template para as páginas da loja.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:h="http://java.sun.com/jsf/html"
6      xmlns:f="http://java.sun.com/jsf/core"
7      xmlns:ui="http://java.sun.com/jsf/facelets">
8
9  <h:head>
10     <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13     <div id="header">
14         <h1>Loja virtual</h1>
15         <hr />
16     </div>
17
18     <ui:insert name="conteudo" />
19
20     <div id="footer" style="text-align: center">
21         <hr />
22         &copy; 2012 K19. Todos os direitos reservados.
23     </div>
24 </h:body>
25 </html>
```

Código XHTML 7.13: template-loja-virtual.xhtml

No diretório WEB-INF, crie um diretório com o nome includes. No diretório includes, crie um arquivo chamado produto-info.xhtml e implemente o pedaço da tela responsável por exibir os detalhes de um produto. O produto particular a ser exibido deve ser passado como parâmetro para esse arquivo.

```

1  <ui:composition xmlns="http://www.w3.org/1999/xhtml"
2      xmlns:h="http://java.sun.com/jsf/html"
3      xmlns:f="http://java.sun.com/jsf/core"
4      xmlns:ui="http://java.sun.com/jsf/facelets">
5
6      <h:panelGrid columns="2" cellspacing="5">
7          <h:graphicImage value="#{produto.caminhoDaFigura}" width="100" />
8          <h:panelGroup>
9              <h2>#{produto.nome}</h2>
10
11              Preço:
12              <h:outputFormat value="{0, number, R$ .00}" style="color: red">
13                  <f:param value="#{produto.preco}" />
14              </h:outputFormat>
```

```

15    </h:panelGroup>
16  </h:panelGrid>
17
18  <h3>Descrição</h3>
19  #{produto.descricao}
20
21</ui:composition>
```

Código XHTML 7.14: *produto-info.xhtml*

Note que o parâmetro para o arquivo *produto-info.xhtml* deve se chamar *produto*. No diretório WebContent, crie um arquivo chamado *loja.xhtml*. Esse arquivo usará o template que você acabou de criar. O conteúdo da página será formado pela caixa de seleção de produto e pelo pedaço de tela de informações do produto.

```

1 <ui:composition
2   template="/WEB-INF/templates/template-loja-virtual.xhtml"
3   xmlns="http://www.w3.org/1999/xhtml"
4   xmlns:h="http://java.sun.com/jsf/html"
5   xmlns:f="http://java.sun.com/jsf/core"
6   xmlns:ui="http://java.sun.com/jsf/facelets">
7
8   <ui:define name="conteudo">
9
10  <h:form>
11    <h:selectOneMenu value="#{produtosBean.idDoProdutoSelecionado}">
12      <f:selectItems
13        value="#{produtosBean.produtos}"
14        var="produto"
15        itemValue="#{produto.id}"
16        itemLabel="#{produto.nome}">
17      </f:selectItems>
18    </h:selectOneMenu>
19    <h:commandButton value="Detalhes" action="#{produtosBean.carregaProduto}" />
20  </h:form>
21
22  <ui:include src="/WEB-INF/includes/produto-info.xhtml">
23    <ui:param name="produto" value="#{produtosBean.produtoSelecionado}" />
24  </ui:include>
25
26  </ui:define>
27
28</ui:composition>
```

Código XHTML 7.15: *loja.xhtml*

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Templates-e-Modularizacao/loja.xhtml>

Resposta do Complementar 8.1

No pacote *managedbeans*, do projeto K19-Navegacao crie uma classe chamada *UsuarioBean* e implemente um managed bean para armazenar a preferência do usuário.

```

1 package managedbeans;
2
3 @SessionScoped
4 @ManagedBean
5 public class UsuarioBean {
6   private String versaoPreferida = "Simples";
7
8   public String principal() {
```

```

9     if (this.versaoPreferida.equals("Avançada"))
10    return "principal-avancada.xhtml";
11   else
12    return "principal-simples.xhtml";
13  }
14
15 // GETTER E SETTER
16 }
```

Código Java 8.3: UsuarioBean.java

O método `principal()` da classe `UsuarioBean` será usado para determinar a página de resposta quando o link para a página principal for clicado.

Na pasta `/WEB-INF/templates`, crie um arquivo chamado `template-aplicacao.xhtml` para implementar um template para as páginas da aplicação.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5  xmlns:h="http://java.sun.com/jsf/html"
6  xmlns:f="http://java.sun.com/jsf/core"
7  xmlns:ui="http://java.sun.com/jsf/facelets">
8
9 <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13 <div id="header" style="text-align: center">
14   <h:outputText value="Loja Virtual" style="font-size: 36px; font-weight: bold" />
15   <h:form>
16     <h:commandLink value="Página principal" action="#{usuarioBean.principal}" />
17     -
18     <h:link value="Configurações" outcome="configuracoes"/>
19   </h:form>
20 </div>
21
22 <ui:insert name="conteudo" />
23
24 <div id="footer" style="text-align: center">
25   <hr />
26   &copy; 2012 K19. Todos os direitos reservados.
27 </div>
28
29 </h:body>
30 </html>
```

Código XHTML 8.18: template-aplicacao.xhtml

Note que o menu para navegar entre as páginas da aplicação foi definido nesse arquivo. Observe que a navegação para a página de configurações foi definida de forma estática, enquanto que a navegação para a página principal foi definida de forma dinâmica.

No diretório `WebContent`, crie dois arquivos chamados `principal-simples.xhtml` e `principal-avancada.xhtml` para implementar as versões simples e avançada da página principal, respectivamente.

```

1 <ui:composition template="/WEB-INF/templates/template-aplicacao.xhtml"
2  xmlns="http://www.w3.org/1999/xhtml"
3  xmlns:h="http://java.sun.com/jsf/html"
4  xmlns:ui="http://java.sun.com/jsf/facelets">
```

```

6   <ui:define name="conteudo">
7     <h1>Página Principal Simples</h1>
8     <ul>
9       <li><h:link>Visualizar produtos cadastrados</h:link></li>
10    </ul>
11   </ui:define>
12 </ui:composition>

```

Código XHTML 8.19: principal-simples.xhtml

```

1 <ui:composition template="/WEB-INF/templates/template-aplicacao.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:ui="http://java.sun.com/jsf/facelets">
5
6   <ui:define name="conteudo">
7     <h1>Página Principal Avançada</h1>
8     <ul>
9       <li><h:link>Visualizar produtos cadastrados</h:link></li>
10      <li><h:link>Editar produtos cadastrados</h:link></li>
11      <li><h:link>Adicionar ou remover produtos</h:link></li>
12      <li><h:link>Adicionar ou remover usuários</h:link></li>
13    </ul>
14   </ui:define>
15 </ui:composition>

```

Código XHTML 8.20: principal-avancada.xhtml

Finalmente, na pasta WebContent, crie um arquivo chamado `configuracoes.xhtml` para implementar a página da aplicação em que o usuário pode alterar suas preferências.

```

1 <ui:composition template="/WEB-INF/templates/template-aplicacao.xhtml"
2   xmlns="http://www.w3.org/1999/xhtml"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core"
5   xmlns:ui="http://java.sun.com/jsf/facelets">
6
7   <ui:define name="conteudo">
8     <h1>Configurações</h1>
9
10    <p>Você está usando a versão <b>#{usuarioBean.versaoPreferida}</b>
11      da aplicação. Se desejar, escolha outra versão abaixo e clique no
12      botão "Salvar".</p>
13
14    <h:form>
15      <h:panelGrid columns="3">
16        <h:outputLabel value="Versão da aplicação: " for="versao" />
17
18        <h:selectOneMenu value="#{usuarioBean.versaoPreferida}" id="versao">
19          <f:selectItem itemLabel="Simples" itemValue="Simples" />
20          <f:selectItem itemLabel="Avançada" itemValue="Avançada" />
21        </h:selectOneMenu>
22
23        <h:commandButton value="Salvar" />
24      </h:panelGrid>
25    </h:form>
26  </ui:define>
27 </ui:composition>

```

Código XHTML 8.21: configuracoes.xhtml

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Navegacao/principal-simples.xhtml>

Resposta do Complementar 8.2

No arquivo template-aplicacao.xhtml, altere o link para a página principal para que ele emita o sinal "principal".

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core"
7   xmlns:ui="http://java.sun.com/jsf/facelets">
8
9 <h:head>
10 <title>K19 Treinamentos</title>
11 </h:head>
12 <h:body>
13   <div id="header" style="text-align: center">
14     <h:outputText value="Loja Virtual" style="font-size: 36px; font-weight: bold" />
15     <h:form>
16       <h:link value="Página principal" outcome="principal"/>
17       -
18       <h:link value="Configurações" outcome="configuracoes"/>
19     </h:form>
20   </div>
21
22   <ui:insert name="conteudo" />
23
24   <div id="footer" style="text-align: center">
25     <hr />
26     &copy; 2012 K19. Todos os direitos reservados.
27   </div>
28
29 </h:body>
30 </html>
```

Código XHTML 8.22: template-aplicacao.xhtml

No arquivo faces-config.xhtml, defina as regras de navegação para a página principal. Se o usuário prefere a versão simples da aplicação, então o sinal "principal" deve direcionar o usuário para a página definida pelo arquivo principal-simples.xhtml. Por outro lado, se o usuário prefere a versão avançada da aplicação, então o sinal "principal" deve direcioná-lo para a página definida pelo arquivo principal-avancada.xhtml.

```

1 ...
2 <navigation-rule>
3   <from-view-id>*</from-view-id>
4
5   <navigation-case>
6     <from-outcome>principal</from-outcome>
7     <if>#{usuarioBean.versaoPreferida == 'Simples'}</if>
8     <to-view-id>principal-simples.xhtml</to-view-id>
9   </navigation-case>
10
11  <navigation-case>
12    <from-outcome>principal</from-outcome>
13    <if>#{usuarioBean.versaoPreferida == 'Avançada'}</if>
14    <to-view-id>principal-avancada.xhtml</to-view-id>
15  </navigation-case>
16
17 </navigation-rule>
18 ...
```

Código XML 8.6: faces-config.xml

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Navegacao/principal-simples.xhtml>

Resposta do Complementar 10.1

No diretório src do projeto K19-Conversao-e-Validacao, crie um pacote chamado model. Nesse pacote, crie uma classe chamada Cotacao para modelar a cotação de uma moeda.

```
1 package model;
2
3 public class Cotacao {
4     private Double valor;
5     private Double variacao;
6     private Date horario;
7
8     // GETTERS E SETTERS
9 }
```

Código Java 10.4: Cotacao.java

Crie um pacote chamado managedbeans no diretório src. Nesse pacote, crie uma classe chamada CotacaoBean para armazenar os valores submetidos pelo formulário.

```
1 package managedbeans;
2
3 @ManagedBean
4 public class CotacaoBean {
5     private Cotacao cotacao = new Cotacao();
6
7     // GETTER E SETTER
8 }
```

Código Java 10.5: CotacaoBean.java

No diretório WebContent, crie um arquivo chamado registrar-cotacao.xhtml e implemente o formulário para registrar uma cotação.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://java.sun.com/jsf/html"
6       xmlns:f="http://java.sun.com/jsf/core">
7 <h:head>
8     <title>K19 Treinamentos</title>
9 </h:head>
10
11 <h:body>
12     <h1>Registrar cotação do dólar</h1>
13     <h:form>
14         <h:panelGrid columns="2">
15             <h:outputLabel value="Valor: " for="valor" />
16             <h:panelGroup>
17                 <h:inputText value="#{cotacaoBean.cotacao.valor}" id="valor" required="true">
18                     <f:convertNumber locale="pt_BR" type="currency" />
19                 </h:inputText>
20                 <h:message for="valor" style="color: red" />
21             </h:panelGroup>
22
23             <h:outputLabel value="Variação: " for="variacao" />
24             <h:panelGroup>
```

```

25   <h:inputText value="#{cotacaoBean.cotacao.variacao}" id="variacao" required="←
26     true">
27     <f:convertNumber maxFractionDigits="5" type="percent" />
28   <h:message for="variacao" style="color: red" />
29 </h:panelGroup>
30
31   <h:outputLabel value="Horário: " for="horario" />
32 <h:panelGroup>
33   <h:inputText value="#{cotacaoBean.cotacao.horario}" id="horario" required="←
34     true">
35     <f:convertDateTime pattern="HH:mm dd-MM-yyyy" />
36   <h:message for="horario" style="color: red" />
37 </h:panelGroup>
38
39   <h:commandButton value="Registrar" />
40 </h:panelGrid>
41 </h:form>
42 </h:body>
43 </html>

```

Código XHTML 10.9: registrar-cotacao.xhtml

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/registrar-cotacao.xhtml>

Resposta do Complementar 10.2

No arquivo `registrar-cotacao.xhtml`, adicione as linhas em destaque do código abaixo.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10
11 <h:body>
12   <h1>Registrar cotação do dólar</h1>
13   <h:form>
14     <h:panelGrid columns="2">
15       <h:outputLabel value="Valor: " for="valor" />
16       <h:panelGroup>
17         <h:inputText value="#{cotacaoBean.cotacao.valor}" id="valor" required="true">
18           <f:convertNumber locale="pt_BR" type="currency" />
19         </h:inputText>
20         <h:message for="valor" style="color: red" />
21       </h:panelGroup>
22
23       <h:outputLabel value="Variação: " for="variacao" />
24       <h:panelGroup>
25         <h:inputText value="#{cotacaoBean.cotacao.variacao}" id="variacao" required="←
26           true">
27           <f:convertNumber maxFractionDigits="5" type="percent" />
28         <h:message for="variacao" style="color: red" />
29       </h:panelGroup>
30
31       <h:outputLabel value="Horário: " for="horario" />
32       <h:panelGroup>

```

```

33      <h:inputText value="#{cotacaoBean.cotacao.horario}" id="horario" required="true">
34          <f:convertDateTime pattern="HH:mm dd-MM-yyyy" />
35      </h:inputText>
36      <h:message for="horario" style="color: red" />
37  </h:panelGroup>
38
39      <h:commandButton value="Registrar" />
40  </h:panelGrid>
41 </h:form>
42
43 <h:panelGroup rendered="#{cotacaoBean.cotacao.valor != null}">
44     <h2>Cotação registrada:</h2>
45     Valor: #{cotacaoBean.cotacao.valor}
46     <br />
47     Variação: #{cotacaoBean.cotacao.variacao * 100}%
48     <br />
49     Horário: #{cotacaoBean.cotacao.horario}
50 </h:panelGroup>
51 </h:body>
52 </html>

```

Código XHTML 10.10: registrar-cotacao.xhtml

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/registrar-cotacao.xhtml>

Altere o valor do atributo `timeZone` do componente `<f:convertDateTime>`. Use, por exemplo, os valores “GMT-2” e “GMT-3”. Teste o formulário novamente.

Resposta do Complementar 10.3

No diretório `/src/model`, crie uma classe chamada `Produto`.

```

1 package model;
2
3 public class Produto {
4     private String nome;
5     private String codigo;
6     private Integer tamanho;
7     private Double volume;
8
9     // GETTERS E SETTERS
10 }

```

Código Java 10.7: Produto.java

No diretório `/src/managedbeans`, crie uma classe chamada `ProdutoBean`.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class ProdutoBean {
5     private Produto produto = new Produto();
6
7     // GETTER E SETTER
8 }

```

Código Java 10.8: ProdutoBean.java

No diretório WebContent, crie um arquivo chamado `cadastro-de-produtos.xhtml` e implemente o formulário.

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7  <h:head>
8   <title>K19 Treinamentos</title>
9  </h:head>
10
11 <h:body>
12   <h1>Cadastro de Produtos</h1>
13   <h:form>
14     <h:panelGrid columns="2">
15       <h:outputLabel value="Nome: " for="nome" />
16       <h:panelGroup>
17         <h:inputText value="#{produtoBean.produto.nome}" id="nome" required="true">
18           <f:validateRegex pattern="[a-zA-Z\s]+"/>
19         </h:inputText>
20         <h:message for="nome" style="color: red" />
21       </h:panelGroup>
22
23       <h:outputLabel value="Código: " for="codigo" />
24       <h:panelGroup>
25         <h:inputText value="#{produtoBean.produto.codigo}" id="codigo" required="true">
26           <f:validateRegex pattern="[A-Z][0-9]+"/>
27           <f:validateLength minimum="4" maximum="10" />
28         </h:inputText>
29         <h:message for="codigo" style="color: red" />
30       </h:panelGroup>
31
32       <h:outputLabel value="Tamanho: " for="tamanho" />
33       <h:panelGroup>
34         <h:inputText value="#{produtoBean.produto.tamanho}" id="tamanho">
35           <f:validateRequired />
36           <f:validateLongRange minimum="1" maximum="32" />
37         </h:inputText>
38         <h:message for="tamanho" style="color: red" />
39       </h:panelGroup>
40
41       <h:outputLabel value="Volume: " for="volume" />
42       <h:panelGroup>
43         <h:inputText value="#{produtoBean.produto.volume}" id="volume">
44           <f:validateRequired />
45           <f:validateDoubleRange minimum="0.1" />
46         </h:inputText>
47         <h:message for="volume" style="color: red" />
48       </h:panelGroup>
49
50       <h:commandButton value="Enviar" />
51     </h:panelGrid>
52   </h:form>
53 </h:body>
54 </html>
```

Código XHTML 10.22: `cadastro-de-produtos.xhtml`

Teste a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/cadastro-de-produtos.xhtml>

Resposta do Complementar 10.4

Na pasta model, crie uma classe chamada RIC.

```

1 package model;
2
3 public class RIC {
4     private String numeroDeIdentificacao;
5     private String digitoVerificador;
6
7     // GETTERS E SETTERS
8 }
```

Código Java 10.20: RIC.java

Na pasta converters, crie uma classe chamada ConversorDeRIC e implemente o conversor.

```

1 package converters;
2
3 @FacesConverter(forClass = RIC.class)
4 public class ConversorDeRIC implements Converter {
5
6     @Override
7     public Object getAsObject(FacesContext context, UIComponent component,
8         String value) {
9
10        String ricString = value.trim();
11
12        if (!Pattern.matches("[0-9]{10}[-]?[0-9]", ricString)) {
13            FacesMessage mensagem = new FacesMessage("Número RIC inválido");
14            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
15            throw new ConverterException(mensagem);
16        }
17
18        ricString = ricString.replaceAll("-", "");
19
20        RIC ric = new RIC();
21        ric.setNumeroDeIdentificacao(ricString.substring(0, 10));
22        ric.setDigitoVerificador(ricString.substring(10, 11));
23
24        return ric;
25    }
26
27    @Override
28    public String getAsString(FacesContext context, UIComponent component,
29        Object value) {
30        RIC ric = (RIC) value;
31        return ric.getNumeroDeIdentificacao() + " - " + ric.getDigitoVerificador();
32    }
33 }
```

Código Java 10.21: ConversorDeRIC.java

Na pasta managedbeans, crie uma classe chamada RICBean e implemente o managed bean que dará suporte ao seu formulário.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class RICBean {
5     private RIC ric;
6
7     // GETTER E SETTER
8 }
```

Código Java 10.22: RICBean.java

Finalmente, no diretório WebContent, crie um arquivo chamado ric.xhtml e implemente um formulário que possua uma caixa de texto associada à propriedade ric do managed bean RICBean.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10
11 <h:body>
12   <h:form>
13     <h:outputLabel value="RIC: " for="campo-ric" />
14     <h:inputText id="campo-ric" value="#{RICBean.ric}" />
15     <h:commandButton value="Enviar" />
16   </h:form>
17
18   <h:message for="campo-ric" style="color: red" />
19 </h:body>
20 </html>
```

Código XHTML 10.26: ric.xhtml

Faça alguns testes acessando a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/ric.xhtml>

Resposta do Complementar 10.5

Primeiro, implemente o managed bean que armazenará a data escolhida pelo usuário. No diretório managedbeans adicione a seguinte classe.

```

1 package managedbeans;
2
3 @ManagedBean
4 public class ReservaBean {
5
6   private Date data;
7
8   // GETTER E SETTER
9 }
```

Código Java 10.27: ReservaBean.java

No diretório WebContent, crie um arquivo chamado reserva.xhtml e implemente o formulário para o usuário digitar a data.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:h="http://java.sun.com/jsf/html"
6   xmlns:f="http://java.sun.com/jsf/core">
7 <h:head>
8   <title>K19 Treinamentos</title>
9 </h:head>
10
11 <h:body>
12   <h:outputText value="Escolha uma data entre 01/05/2014 e 30/08/2014." />
13   <h:form>
```

```

14 <h:outputLabel value="Data: (dd/MM/yyyy)" for="campo-data" />
15
16 <h:inputText id="campo-data" value="#{reservaBean.data}" required="true">
17   <f:convertDateTime pattern="dd/MM/yyyy" />
18
19   <f:validator validatorId="validators.ValidadorDeData" />
20
21   <f:attribute name="padraoDeData" value="dd/MM/yyyy" />
22   <f:attribute name="inicio" value="01/05/2014" />
23   <f:attribute name="fim" value="30/08/2014" />
24 </h:inputText>
25
26   <h:commandButton value="Enviar" />
27 </h:form>
28
29 <h:message for="campo-data" style="color: red" />
30 </h:body>
31 </html>

```

Código XHTML 10.30: reserva.xhtml

No pacote validators, crie uma classe chamada ValidadorDeData para implementar o validador.

```

1 package validators;
2
3 @FacesValidator("validators.ValidadorDeData")
4 public class ValidadorDeData implements Validator {
5
6   @Override
7   public void validate(FacesContext context, UIComponent component,
8     Object value) throws ValidatorException {
9
10    String padraoDeData = (String) component.getAttributes().get("padraoDeData");
11    String inicio = (String) component.getAttributes().get("inicio");
12    String fim = (String) component.getAttributes().get("fim");
13
14    Date data = (Date) value;
15    Date dataInicio;
16    Date dataFim;
17
18    try {
19      dataInicio = (Date) new SimpleDateFormat(padraoDeData).parse(inicio);
20      dataFim = (Date) new SimpleDateFormat(padraoDeData).parse(fim);
21    } catch (ParseException e) {
22      FacesMessage mensagem = new FacesMessage(
23        "Erro ao criar as datas de início e fim do intervalo.");
24      mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
25      throw new ValidatorException(mensagem);
26    }
27
28    if (data.before(dataInicio) || data.after(dataFim)) {
29      FacesMessage mensagem = new FacesMessage(
30        "A data escolhida precisa estar entre " + inicio + " e " + fim);
31      mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
32      throw new ValidatorException(mensagem);
33    }
34  }
35

```

Código Java 10.28: ValidadorDeData.java

Acesse a aplicação no endereço:

<http://localhost:8080/K19-Conversao-e-Validacao/reserva.xhtml>

Teste o formulário submetendo-o com diversas datas diferentes, inclusive com datas nos extremos do intervalo permitido.

Agora, modifique o arquivo `reserva.xhtml` de forma que o conversor de datas considere o fuso horário da cidade de São Paulo, como no trecho de código abaixo.

```
1 ...
2 <f:convertDateTime pattern="dd/MM/yyyy" timeZone="GMT -3" />
3 ...
```

Código XHTML 10.31: reserva.xhtml

Teste o formulário novamente.