



TREINAMENTOS

Desenvolvimento Web com JSF2 e JPA2

Desenvolvimento Web com JSF 2 e JPA 2

17 de novembro de 2010





Sumário

1	Banco de dados	1
1.1	Bases de dados (<i>Databases</i>)	2
1.2	Tabelas	4
1.3	Operações Básicas	7
1.4	Chaves Primária e Estrangeira	11
1.5	Consultas Avançadas	11
1.6	Exercícios	12
2	JDBC	27
2.1	Driver	27
2.2	JDBC	28
2.3	Instalando o Driver JDBC do MySQL Server	28
2.4	Criando uma conexão	28
2.5	Inserindo registros	29
2.6	Exercícios	29
2.7	SQL Injection	31
2.8	Exercícios	33
2.9	Listando registros	33
2.10	Exercícios	34
2.11	Fábrica de conexões (Factory)	35
2.12	Exercícios	36
3	JPA 2 e Hibernate	37
3.1	Múltiplas sintaxes da linguagem SQL	37
3.2	Orientação a Objetos VS Modelo Entidade Relacionamento	37
3.3	Ferramentas ORM	37
3.4	O que é JPA e Hibernate	38
3.5	Configuração	38
3.6	Mapeamento	39
3.7	Gerando o banco	40
3.8	Exercícios	41
3.9	Manipulando entidades	43
3.9.1	Persistindo	43
3.9.2	Buscando	43
3.9.3	Removendo	44
3.9.4	Atualizando	44

3.9.5	Listando	44
3.9.6	Transações	44
3.10	Exercícios	45
3.11	Repository	46
3.12	Exercícios	47
4	Web Container	49
4.1	Necessidades de uma aplicação web	49
4.2	Web Container	49
4.3	Especificação Java EE	50
4.4	Exercícios	50
4.5	Aplicação Web Java	50
4.6	Exercícios	51
4.7	Processando requisições	51
4.8	Servlet	51
4.8.1	Inserindo conteúdo na resposta	52
4.9	Exercícios	52
4.10	JSP	53
4.11	Exercícios	53
4.12	Frameworks	53
5	Visão Geral do JSF 2	55
5.1	Aplicação de exemplo	55
5.2	Managed Beans	55
5.2.1	GeradorDeApostasBean	56
5.3	Facelets e Componentes Visuais	56
5.3.1	Tela de entrada	56
5.3.2	Tela de Saída	57
5.4	Exercícios	58
6	Componentes Visuais	63
6.1	Formulários	63
6.2	Panel Grid	65
6.3	Panel Group	66
6.4	Tabelas	67
6.5	Namespaces	68
6.6	Esqueleto HTML	68
6.7	Exercícios	68
7	Facelets	71
7.1	Templating	71
7.2	Exercícios	72
7.3	Particionando as telas	73
7.4	Exercícios	74

8	Navegação	77
8.1	Navegação Estática Implícita	77
8.2	Navegação Estática Explícita	78
8.3	Exercícios	78
8.4	Navegação Dinâmica Implícita	80
8.5	Navegação Dinâmica Explícita	81
8.6	Exercícios	81
9	Managed Beans	85
9.1	Criando Managed Beans	85
9.2	Disponibilizando dados para as telas	86
9.3	Recebendo dados das telas	86
9.4	Definindo o tratamento das ações	86
9.5	Expression Language	87
9.5.1	Nome dos Managed Beans	87
9.5.2	Acessando as propriedades dos Managed Beans	87
9.6	Binding	87
9.7	Escopo	88
9.7.1	Request	88
9.7.2	Session	89
9.7.3	Application	90
9.7.4	View	90
9.8	Interdependência e Injeção	91
9.9	Exercícios	92
10	Conversão e Validação	95
10.1	Conversão	95
10.1.1	Conversão Padrão Implícita	95
10.1.2	Conversão Padrão Explícita	96
10.2	Mensagens de Erro	97
10.2.1	h:message	97
10.2.2	h:messages	98
10.2.3	Alterando as Mensagens de Erro	98
10.3	Exercícios	99
10.4	Validação	101
10.4.1	Validação Padrão	101
10.4.2	Campo Obrigatório (Required)	101
10.4.3	f:validateLongRange	101
10.4.4	f:validateDoubleRange	102
10.4.5	f:validateLength	102
10.4.6	f:validateRegex	102
10.4.7	Bean Validation	102
10.5	Exercícios	103

11	Eventos	107
11.1	Eventos de Aplicação (Application Events)	107
11.1.1	ActionEvent	107
11.1.2	ValueChangeEvent	108
11.2	Eventos de Ciclo de Vida (Lifecycle Events)	109
11.3	Exercícios	109
12	Ajax	113
12.1	Fazendo requisições AJAX	113
12.2	Recarregando alguns “pedaços” das telas	114
12.3	Processando alguns “pedaços” das telas	115
12.4	Palavras especiais	115
12.5	Exercícios	116
13	Projeto	119
13.1	Modelo	119
13.2	Exercícios	119
13.3	Persistência - Mapaemento	120
13.4	Exercícios	120
13.5	Persistência - Configuração	121
13.6	Exercícios	121
13.7	Persistência - Open Session in View	122
13.7.1	Gerenciando as fábricas de Entity Managers	122
13.7.2	Filtros	123
13.7.3	Gerenciando os Entity Managers	123
13.8	Exercícios	124
13.9	Persistência - Repositórios	125
13.10	Exercícios	126
13.11	Apresentação - Template	127
13.12	Exercícios	127
13.13	Cadastrando e Listando Seleções	129
13.14	Exercícios	130
13.15	Mensagens de Erro	132
13.16	Exercícios	132
13.17	Removendo Seleções	132
13.18	Exercícios	133
13.19	Otimizando o número de consultas	133
13.20	Exercícios	134
13.21	Cadastrando, Listando e Removendo Jogadores	135
13.22	Exercícios	135
13.23	Removendo Seleções com Jogadores	139
13.24	Exercícios	139
13.25	Controle de Acesso	140
13.26	Exercícios	140
13.27	Ordem dos filtros	143
13.28	Exercícios	143

13.29Controle de Erro	144
13.30Exercícios	144
13.31Enviando email	146
13.32Exercícios	147



Capítulo 1

Banco de dados

O nosso objetivo é desenvolver aplicações em **Java**. Essas aplicações necessitam armazenar as informações relacionadas ao seu domínio em algum lugar. Por exemplo, uma aplicação de gerenciamento de uma livraria deve armazenar os dados dos livros que ela comercializa. Uma forma de suprir essa necessidade seria armazenar essas informações em arquivos. Contudo, alguns fatores importantes nos levam a descartar tal opção.

A seguir, apresentamos as principais preocupações a serem consideradas ao trabalhar com dados:

Segurança: As informações potencialmente confidenciais devem ser controladas de forma que apenas usuários e sistemas autorizados tenham acesso a elas.

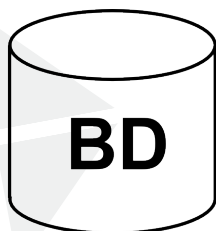
Integridade: Eventuais falhas de software ou hardware não devem corromper os dados.

Acesso: As funções de consulta e manipulação dos dados devem ser implementadas.

Concorrência: Usualmente, diversos sistemas e usuários acessarão as informações de forma concorrente. Apesar disso, os dados não podem ser perdidos ou corrompidos.

Considerando todos esses aspectos, concluímos que seria necessária a utilização de um sistema complexo para manusear as informações das nossas aplicações.

Felizmente, tal tipo de sistema já existe e é conhecido como **Sistema Gerenciador de Banco de Dados (SGBD)**.



- * **Segurança**
- * **Integridade**
- * **Acesso**
- * **Concorrência**

Sistemas gerenciadores de banco de dados

No mercado, há diversas opções de sistemas gerenciadores de banco de dados. A seguir, apresentamos os mais populares:

- Oracle
- SQL Server
- MySQL Server
- PostgreSQL

MySQL Server

Neste treinamento, utilizaremos o MySQL Server, que é mantido pela Oracle e vastamente utilizado no mercado. O MySQL Server pode ser obtido a partir do site:

<http://www.mysql.com>.

MySQL Query Browser

Para interagir com o MySQL Server, utilizaremos um cliente com interface gráfica chamado de MySQL Query Browser.

1.1 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.

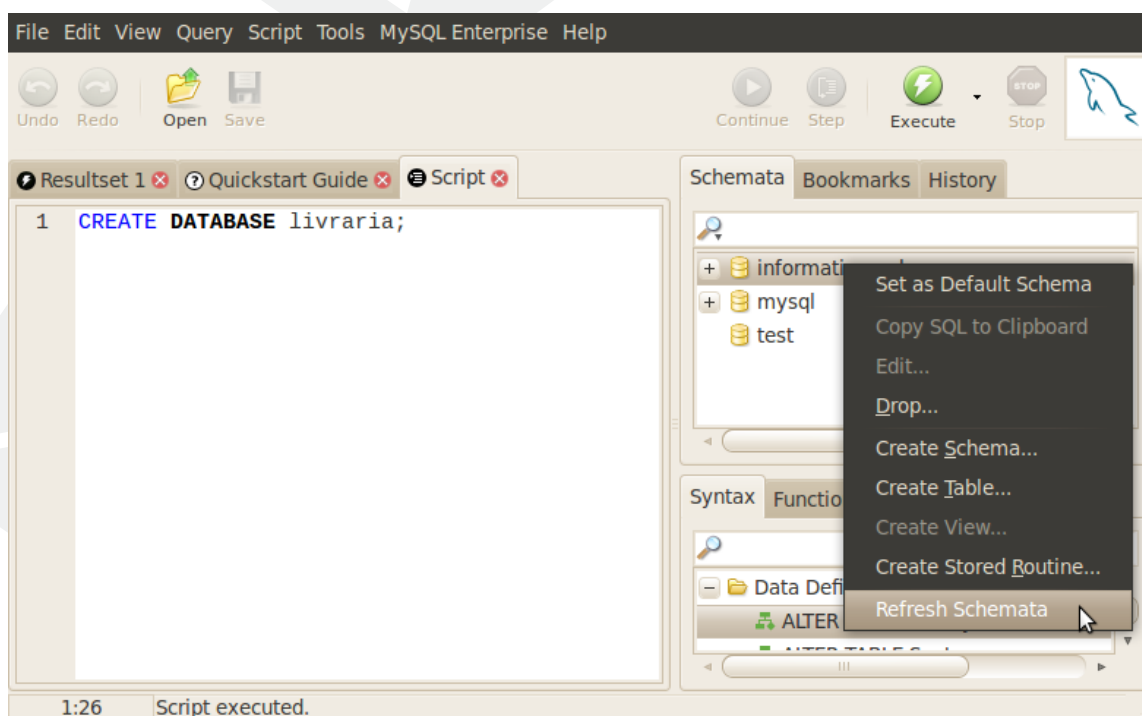
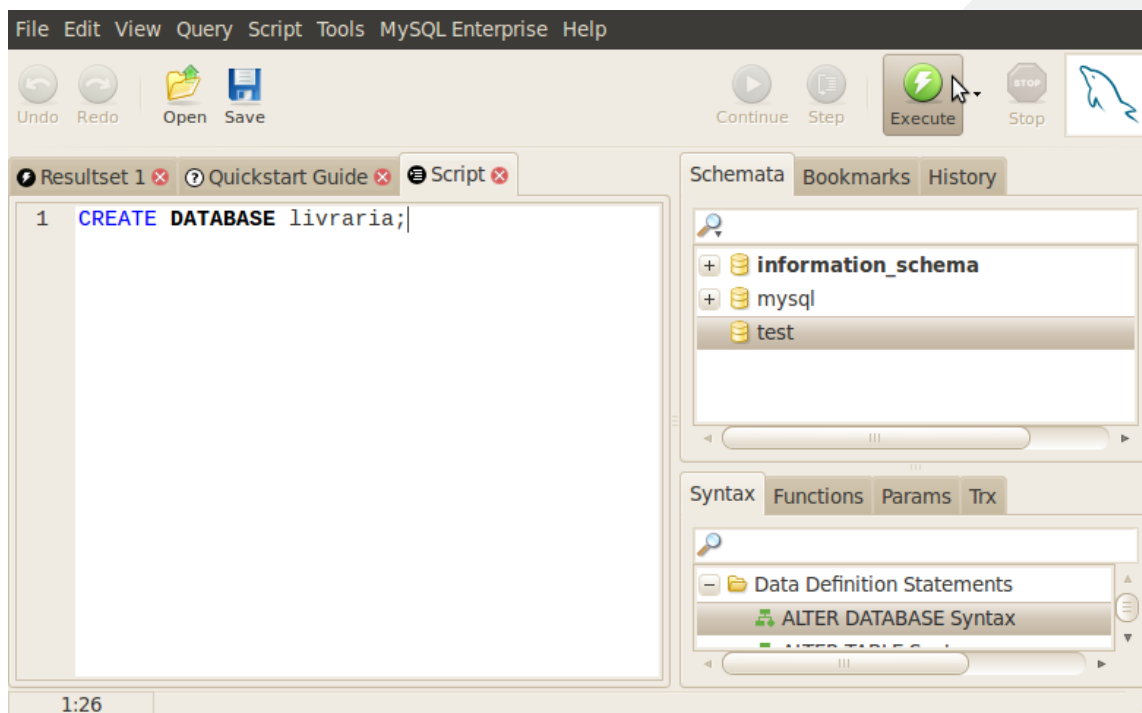
Caso os dados fossem mantidos sem nenhuma separação lógica, a organização ficaria prejudicada. Além disso, seria mais difícil implementar regras de segurança referentes ao acesso dos dados. Tais regras criam restrições quanto ao conteúdo acessível por cada usuário. Determinado usuário, por exemplo, poderia ter permissão de acesso aos dados dos clientes do banco, mas não às informações dos produtos da loja virtual, ou vice-versa.

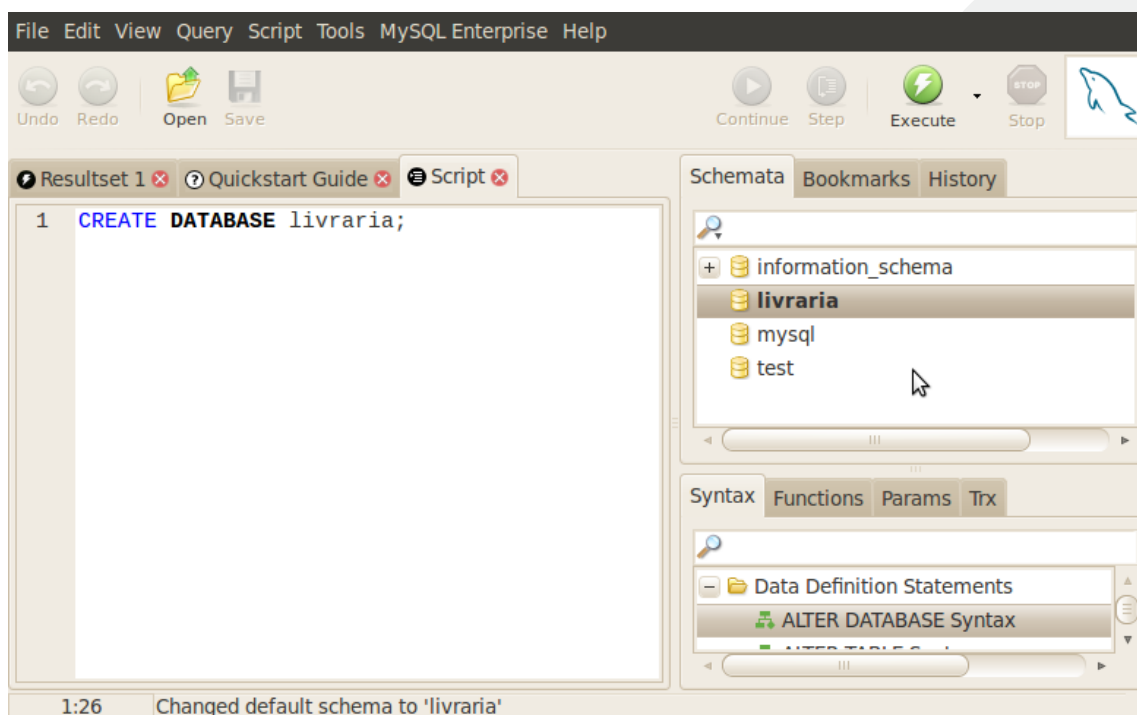
Então, por questões de organização e segurança, os dados devem ser armazenados separadamente no SGBD. Daí surge o conceito de **base de dados** (database).

Uma base de dados é um agrupamento lógico das informações de um determinado domínio, como, por exemplo, os dados da nossa livraria.

Criando uma base de dados no MySQL Server

Para criar uma base de dados no MySQL Server, utilizamos o comando CREATE DATABASE.





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 guardar algumas configurações.

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

1.2 Tabelas

Um servidor de banco de dados é dividido em bases de dados com o intuito de separar as informações de sistemas 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

Uma tabela é formada por **registros**(linhas) e os registros são formados por **campos**(colunas). Por exemplo, suponha 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.

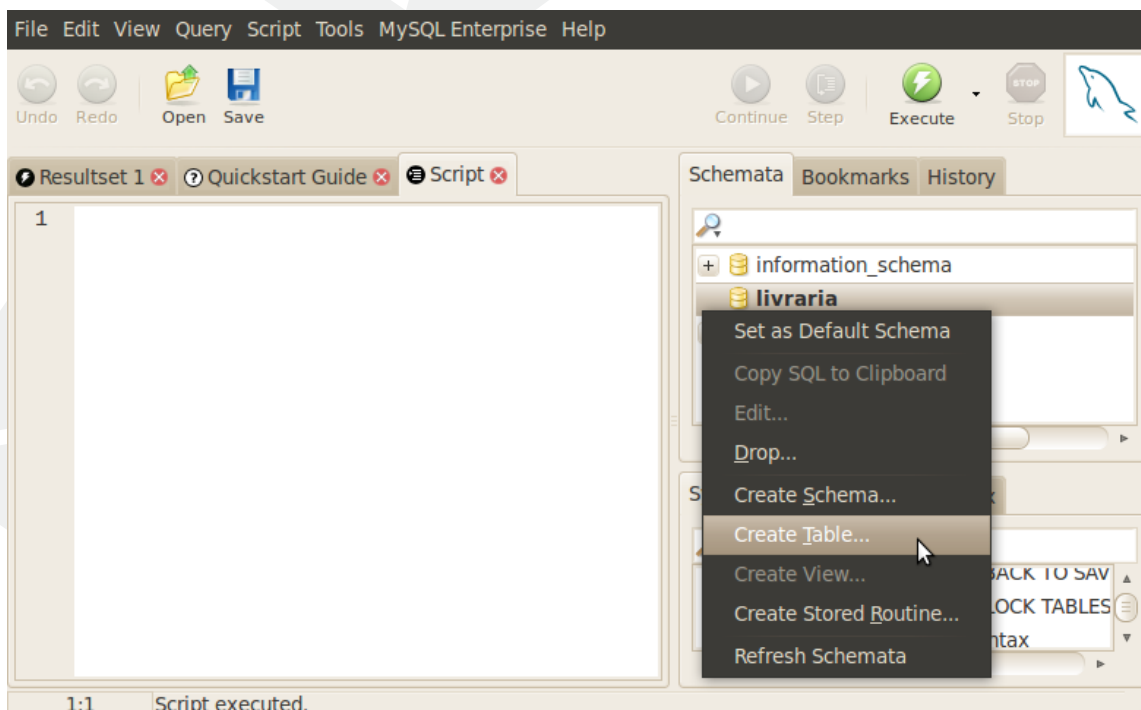


Table Name: Comment:

Columns and Indices Table Options Advanced Options

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

Column Details Indices Foreign Keys

Name: Data Type: Default Value: NULL

Column Options

- ☐ Primary Key
- ☐ Not NULL
- ☐ Auto Increment

Flags: ☐ BINARY

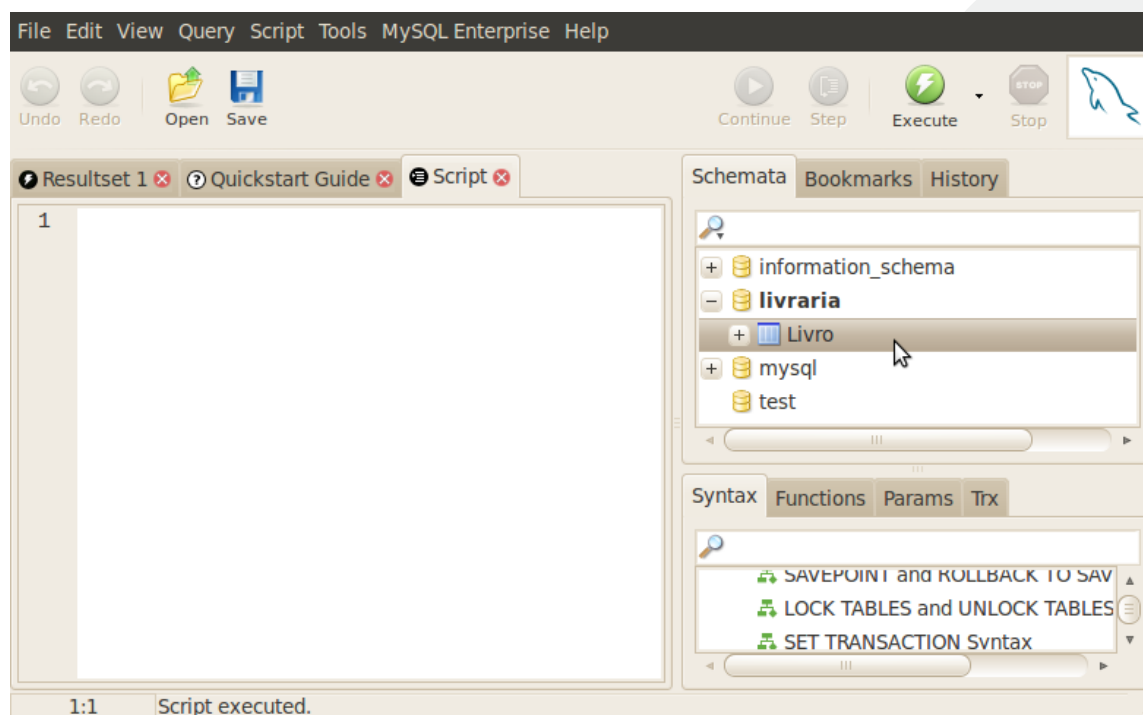
Character Set:

Collation:

Comment:

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` (  
  `titulo` VARCHAR(255) ,  
  `preco` DOUBLE  
)  
ENGINE = MyISAM;
```

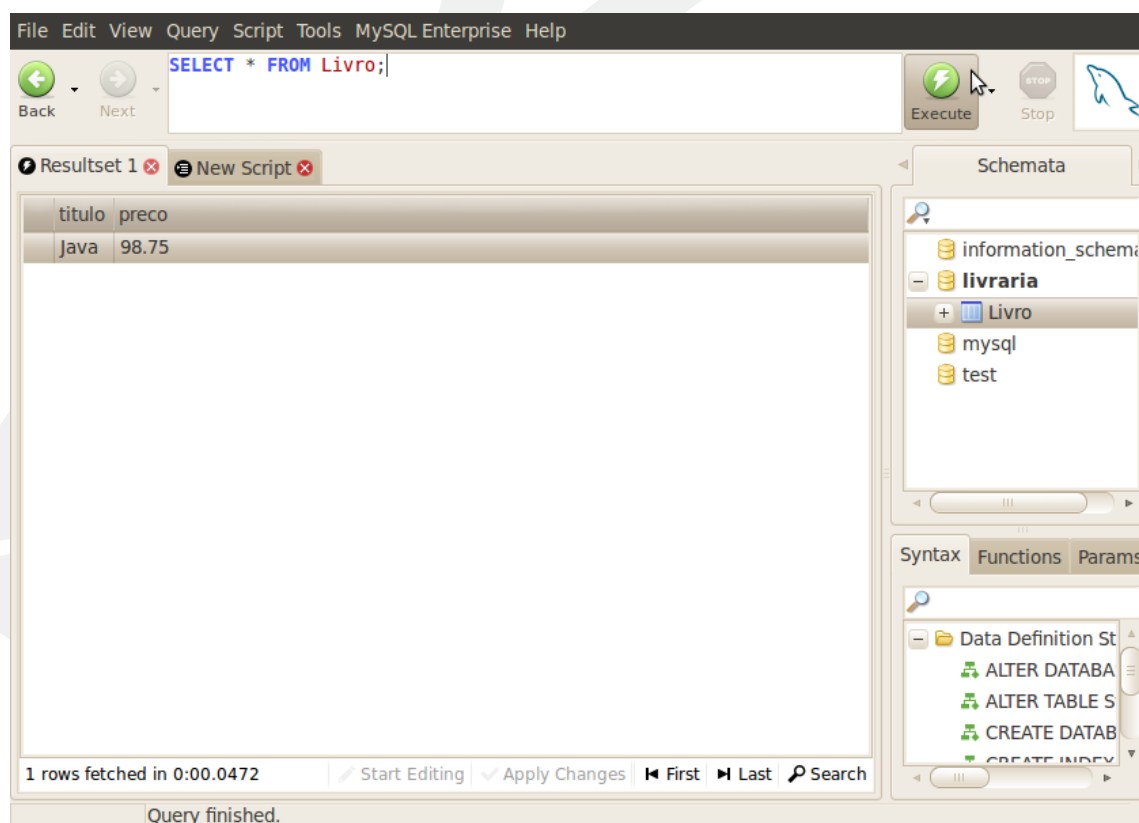
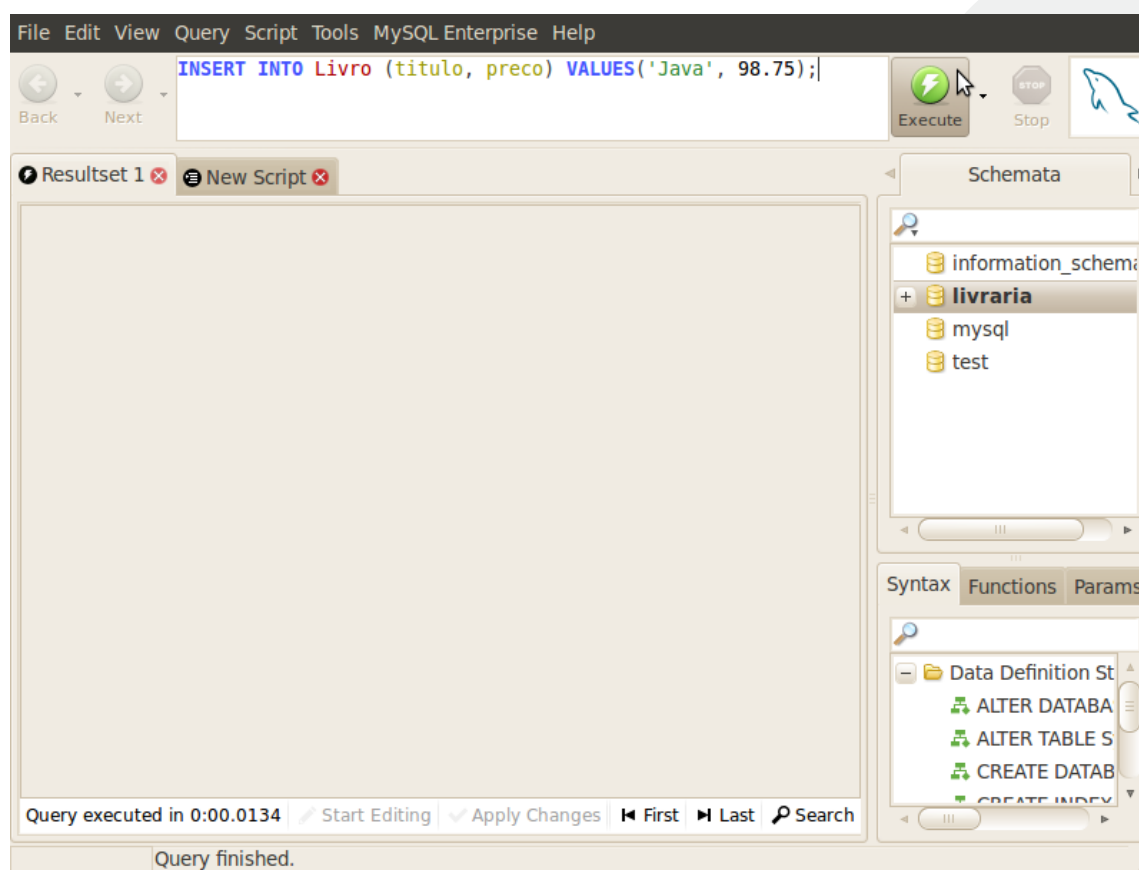


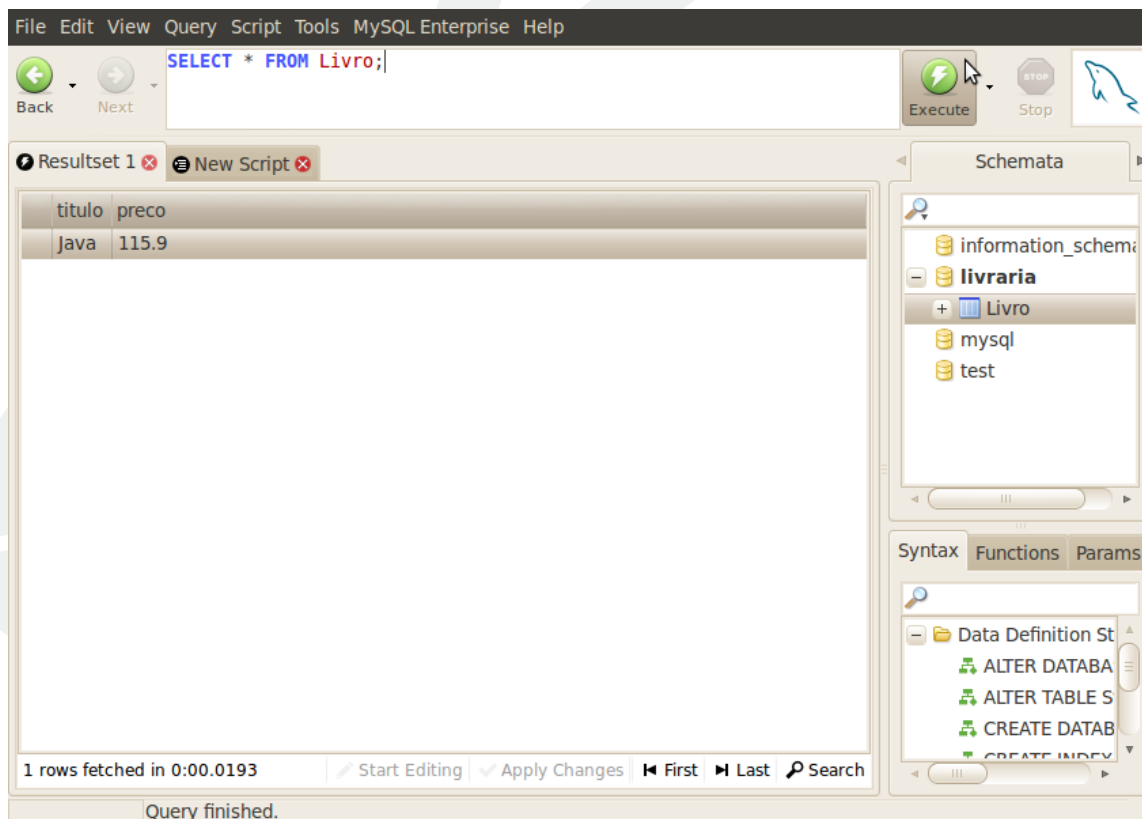
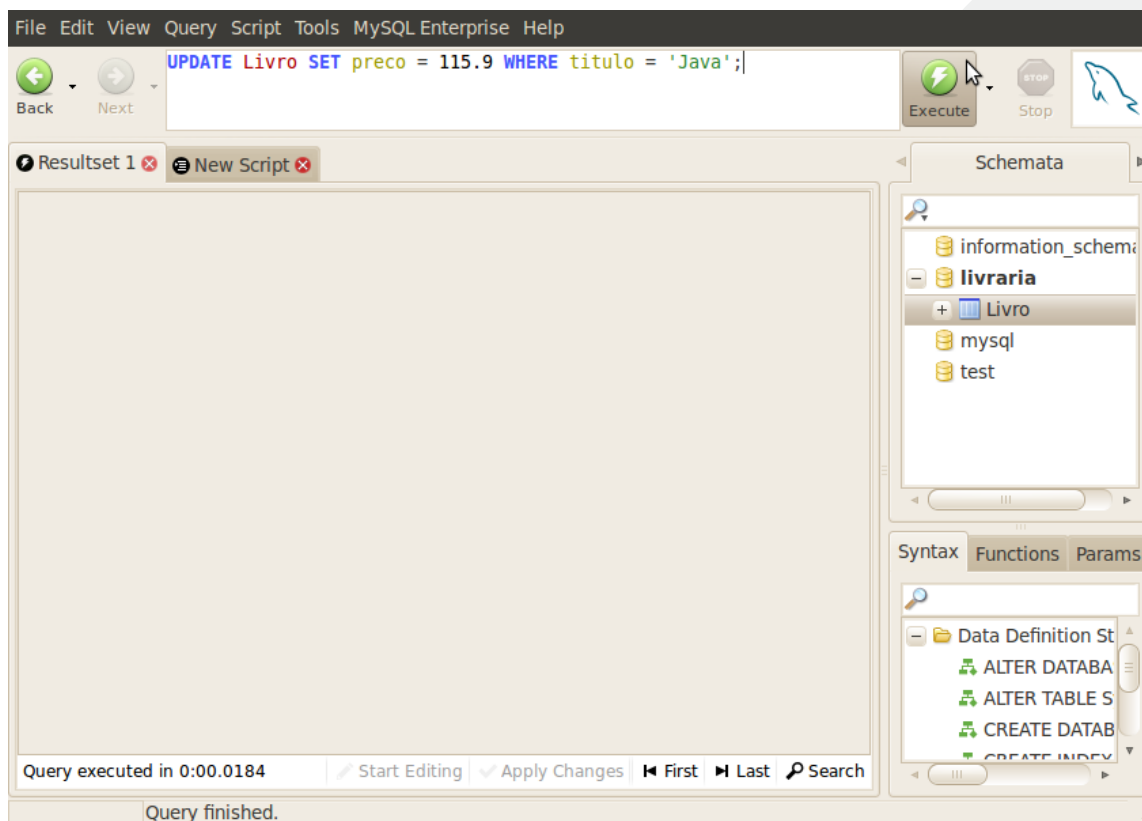
Se uma tabela não for mais desejada ela pode ser removida através do comando DROP TABLE.

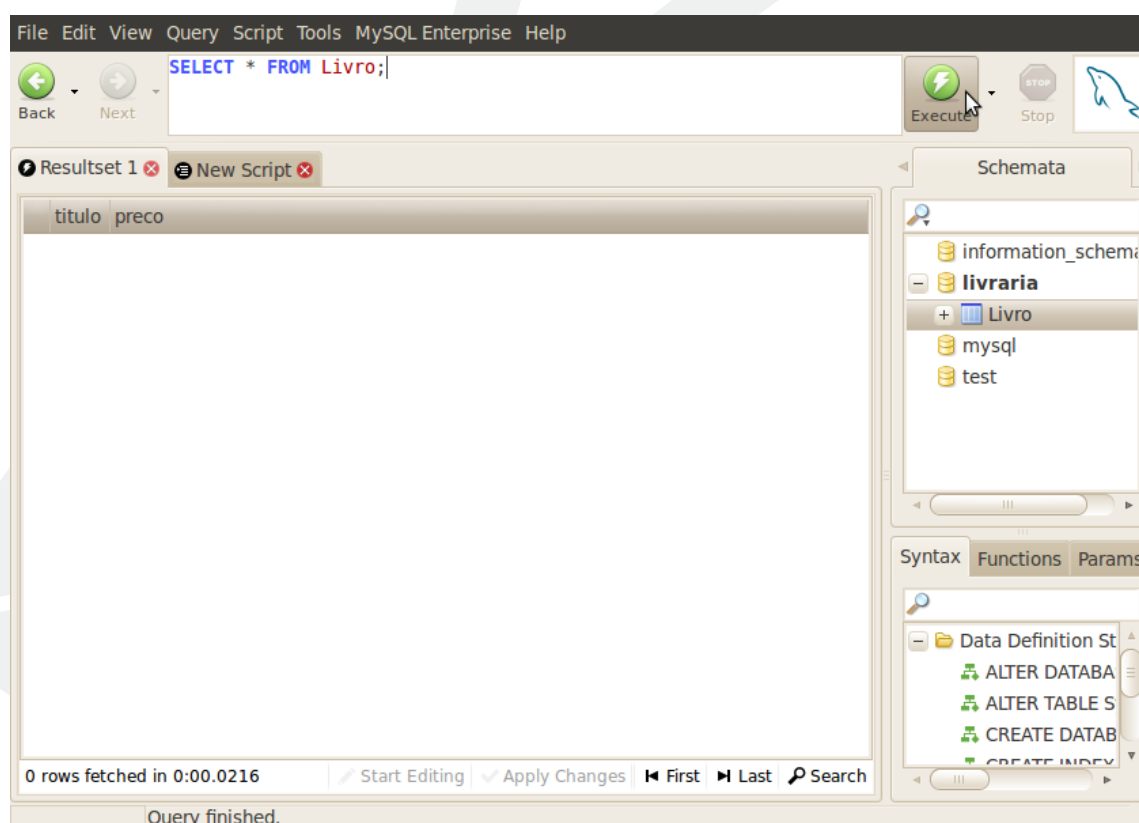
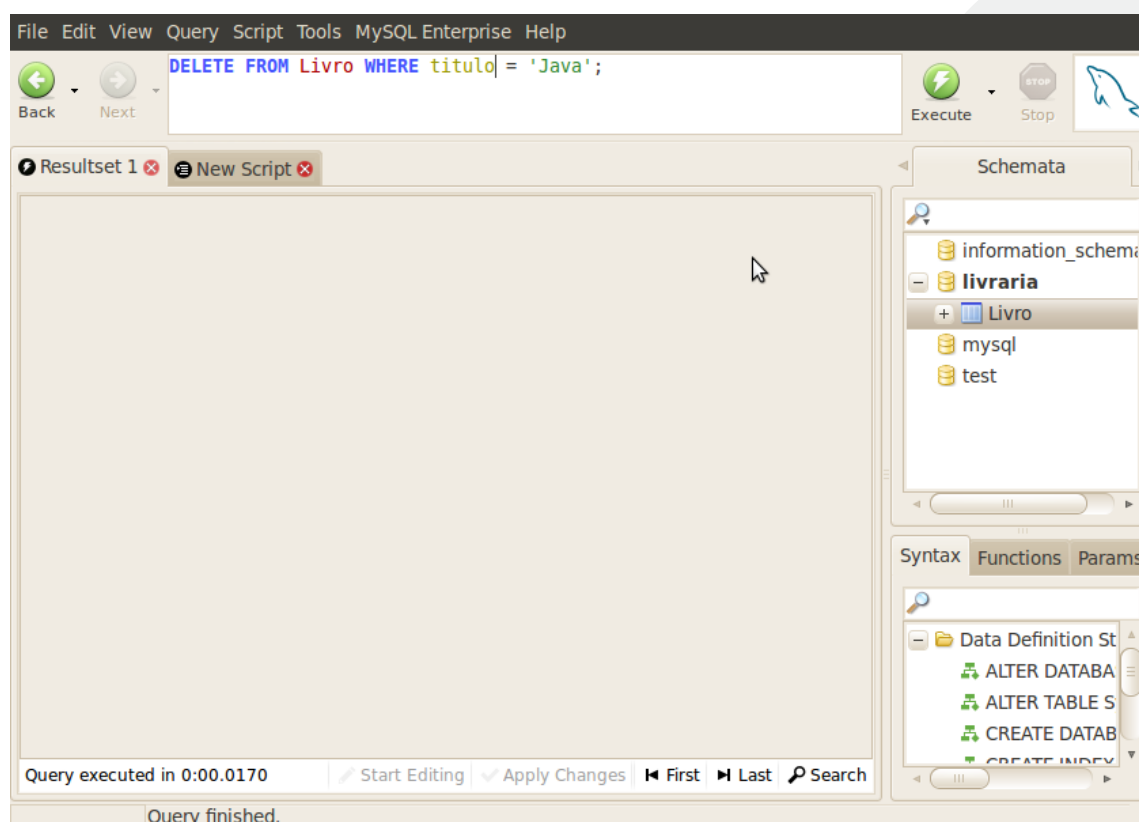
1.3 Operações Básicas

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

Essas operações são realizadas através da linguagem de consulta denominada **SQL**. Esta linguagem oferece quatro comandos básicos: INSERT, SELECT, UPDATE e DELETE. Estes comandos são utilizados para inserir, ler, alterar e remover registros respectivamente.







1.4 Chaves Primária e Estrangeira

Suponha que os livros da nossa livraria são separados por editoras. Uma editora possui nome e telefone. Para armazenar esses dados, uma nova tabela deve ser criada.

Nesse momento, teríamos duas tabelas (Livro e Editora). Eventualmente, será necessário 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 a qual ele pertence. 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 com editoras inexistentes, pois não há nenhum vínculo explícito entre as tabelas. Para solucionar estes 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** deve ser uma chave estrangeira associada ao campo **id**.

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.

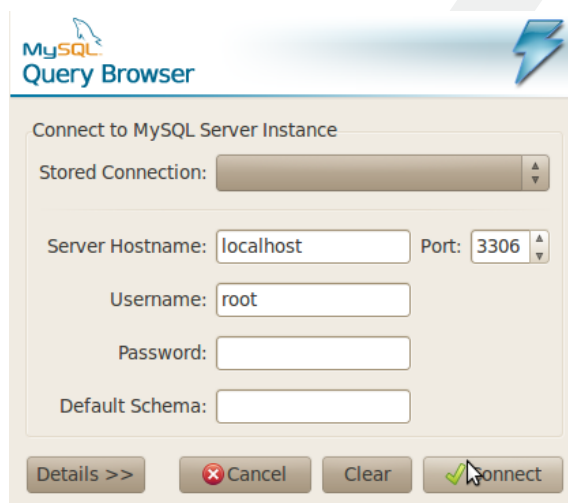
Com a definição da chave estrangeira, um livro não pode ser inserido com o valor do campo **editora_id** inválido. Caso tentássemos obteríamos uma mensagem de erro.

1.5 Consultas Avançadas

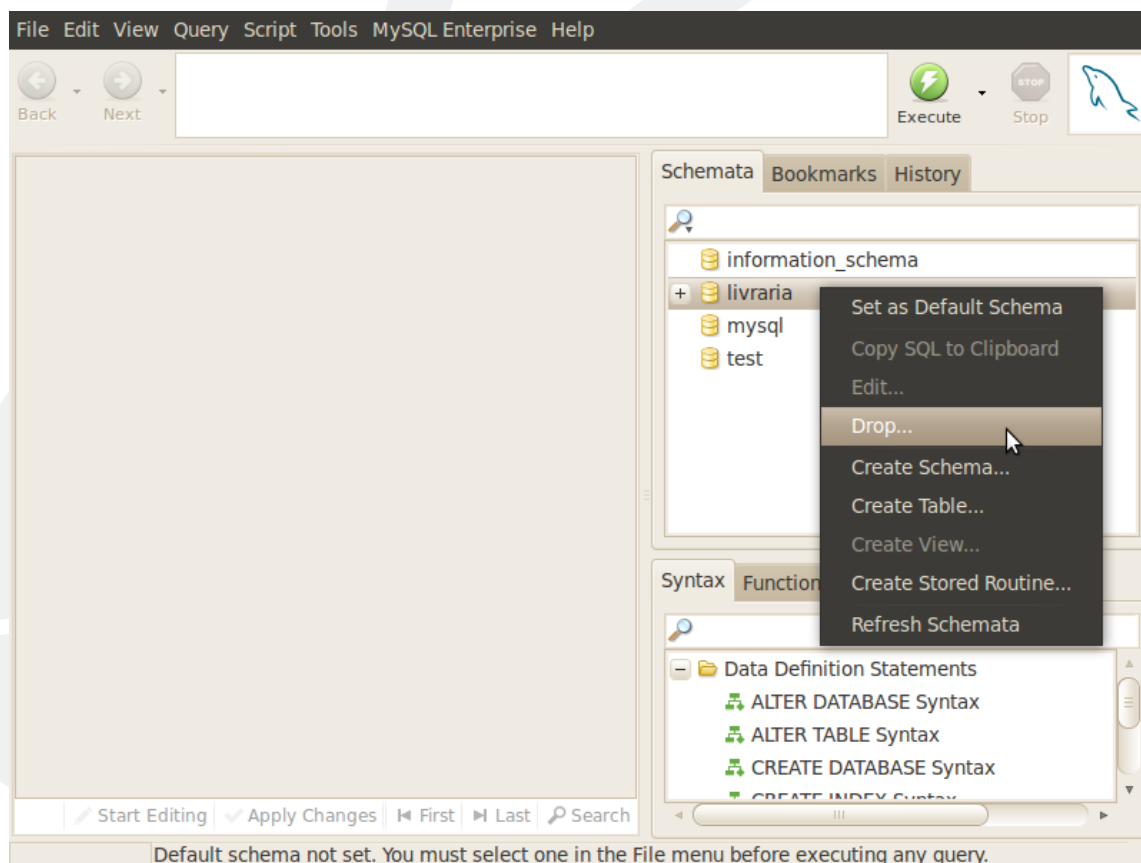
Com o conceito de chave estrangeira, podemos fazer consultas complexas envolvendo os registros de duas ou mais tabelas. Por exemplo, descobrir todos os livros de uma determinada editora.

1.6 Exercícios

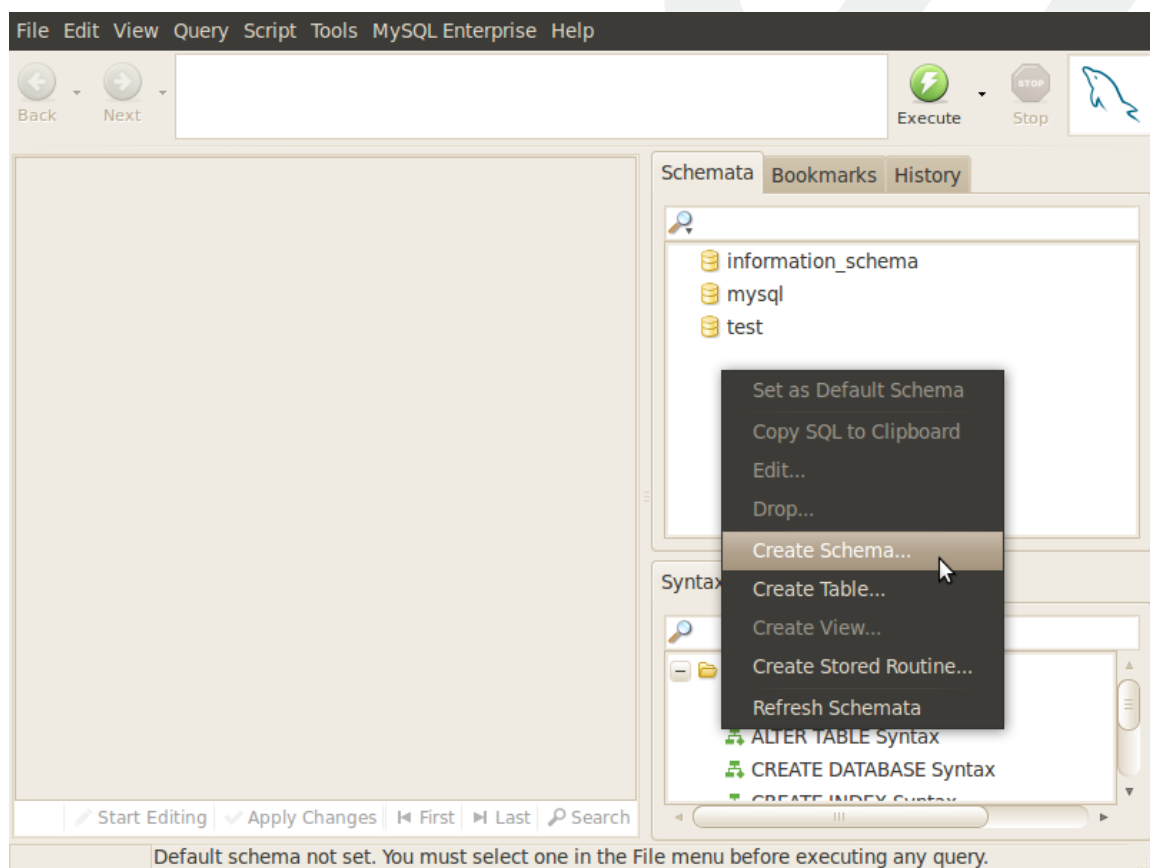
1. Abra o **MySQL Query Browser** utilizando **localhost** como Server Hostname, **root** como Username e **root** como Password.



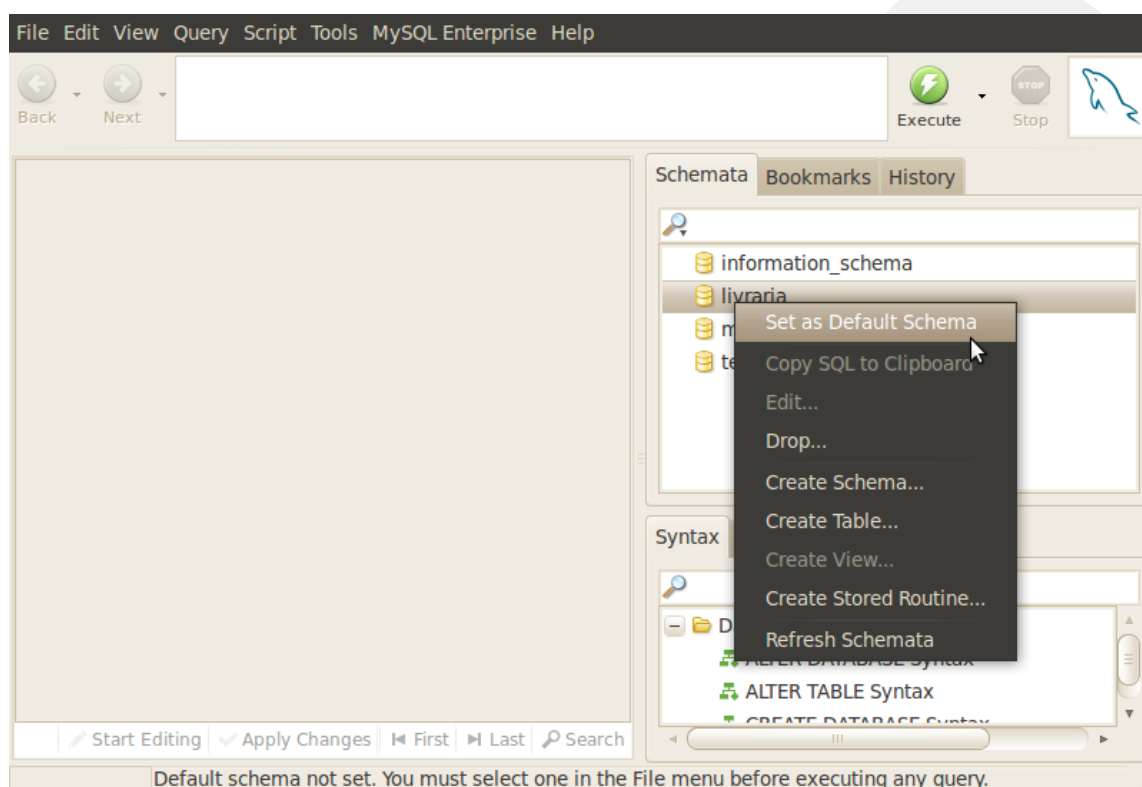
2. Caso exista uma base de dados chamada **Livraria**, remova-a conforme a figura abaixo:



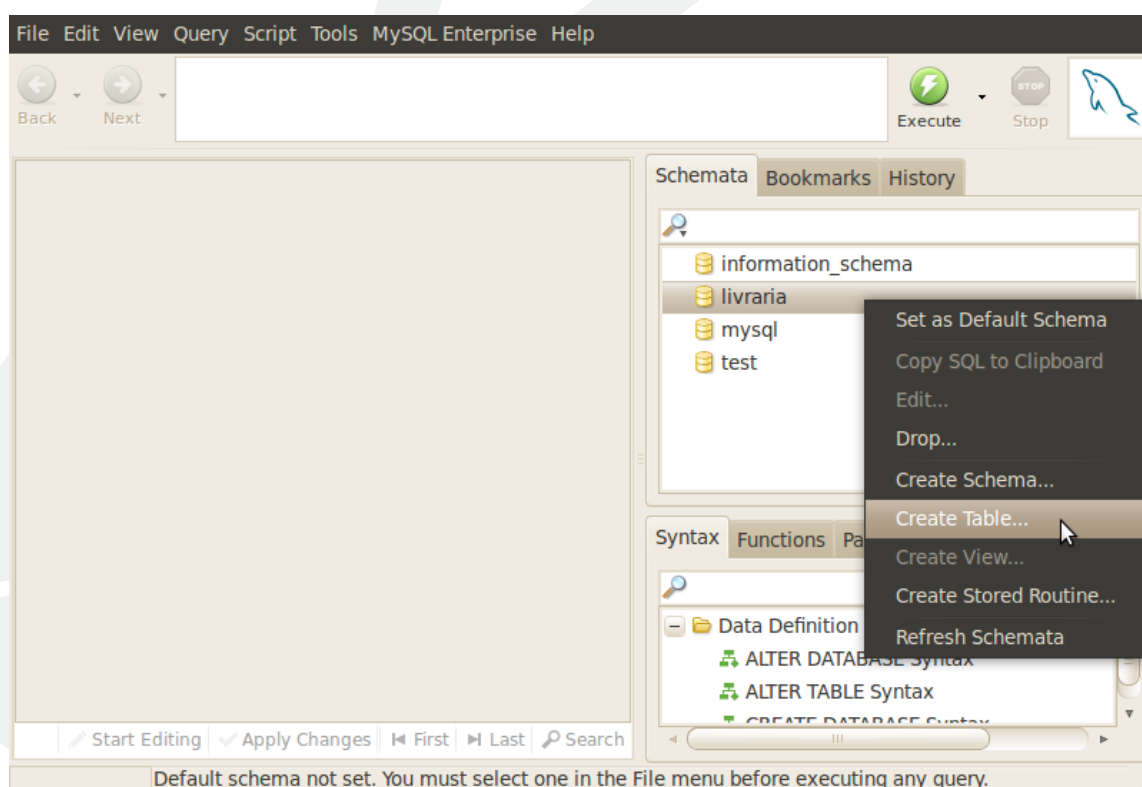
3. Crie uma nova base de dados chamada **livraria**, conforme mostrado na figura abaixo. Você vai utilizar esta base nos exercícios seguintes.



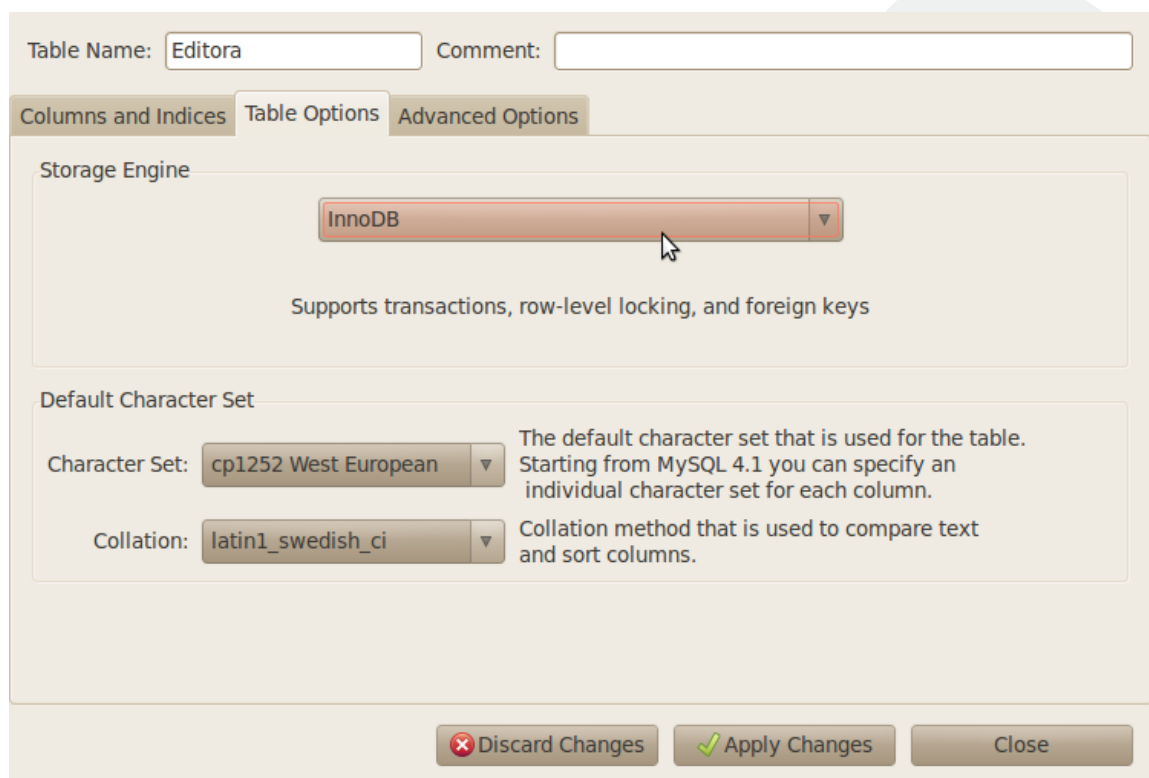
4. Selecione a base de dados livraria como padrão.



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



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



The screenshot shows the 'Table Options' tab of a MySQL database management interface. At the top, there are fields for 'Table Name' (containing 'Editora') and 'Comment'. Below these are three tabs: 'Columns and Indices', 'Table Options' (which is selected), and 'Advanced Options'. The 'Table Options' section contains two main areas. The first is 'Storage Engine', which has a dropdown menu set to 'InnoDB'. Below this dropdown, it states 'Supports transactions, row-level locking, and foreign keys'. The second area is 'Default Character Set', which contains two sub-sections. The first sub-section is 'Character Set', with a dropdown set to 'cp1252 West European'. To its right, a text box explains: '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.' The second sub-section is 'Collation', with a dropdown set to 'latin1_swedish_ci'. To its right, a text box explains: 'Collation method that is used to compare text and sort columns.' At the bottom of the dialog, there are three buttons: 'Discard Changes' (with a red X icon), 'Apply Changes' (with a green checkmark icon), and 'Close'.

Table Name: 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.

Discard Changes Apply Changes Close

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: 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"/>			

Column Details Indices Foreign Keys

Name: Data Type: Default Value: NULL

Column Options

☒ Primary Key
☒ Not NULL
☒ Auto Increment

Flags: ☐ UNSIGNED ☐ ZEROFILL

Character Set: Collation:

Comment:

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`.`Editora` (
  `id` BIGINT NOT NULL AUTO_INCREMENT,
  `nome` VARCHAR(255) NOT NULL,
  `email` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`id`)
)
ENGINE = InnoDB;
```

6. Crie uma tabela chamada **Livro** conforme as figuras abaixo:





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

The screenshot shows the 'Table Options' tab of a MySQL database management interface. At the top, there are input fields for 'Table Name' (containing 'Livro') and 'Comment'. Below these are three tabs: 'Columns and Indices', 'Table Options' (which is selected), and 'Advanced Options'. The 'Table Options' section contains two main areas. The first is 'Storage Engine', which has a dropdown menu currently set to 'InnoDB'. Below this dropdown, it states 'Supports transactions, row-level locking, and foreign keys'. The second area is 'Default Character Set', which includes two sub-sections. The first sub-section is 'Character Set', with a dropdown menu and a text box explaining: '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.' The second sub-section is 'Collation', with a dropdown menu and a text box explaining: 'Collation method that is used to compare text and sort columns.' At the bottom of the dialog, there are three buttons: 'Discard Changes' (with a red X icon), 'Apply Changes' (with a green checkmark icon), and 'Close'.

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: 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: Data Type: Default Value:

Column Options

- ☒ Primary Key
- ☒ Not NULL
- ☒ Auto Increment

Flags:

- ☐ UNSIGNED
- ☐ ZEROFILL

Character Set: Collation:

Comment:

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

Table Name: 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

fk_editora

Foreign Key Settings

Key Name:

On Delete:

On Update:

Refer. Table:

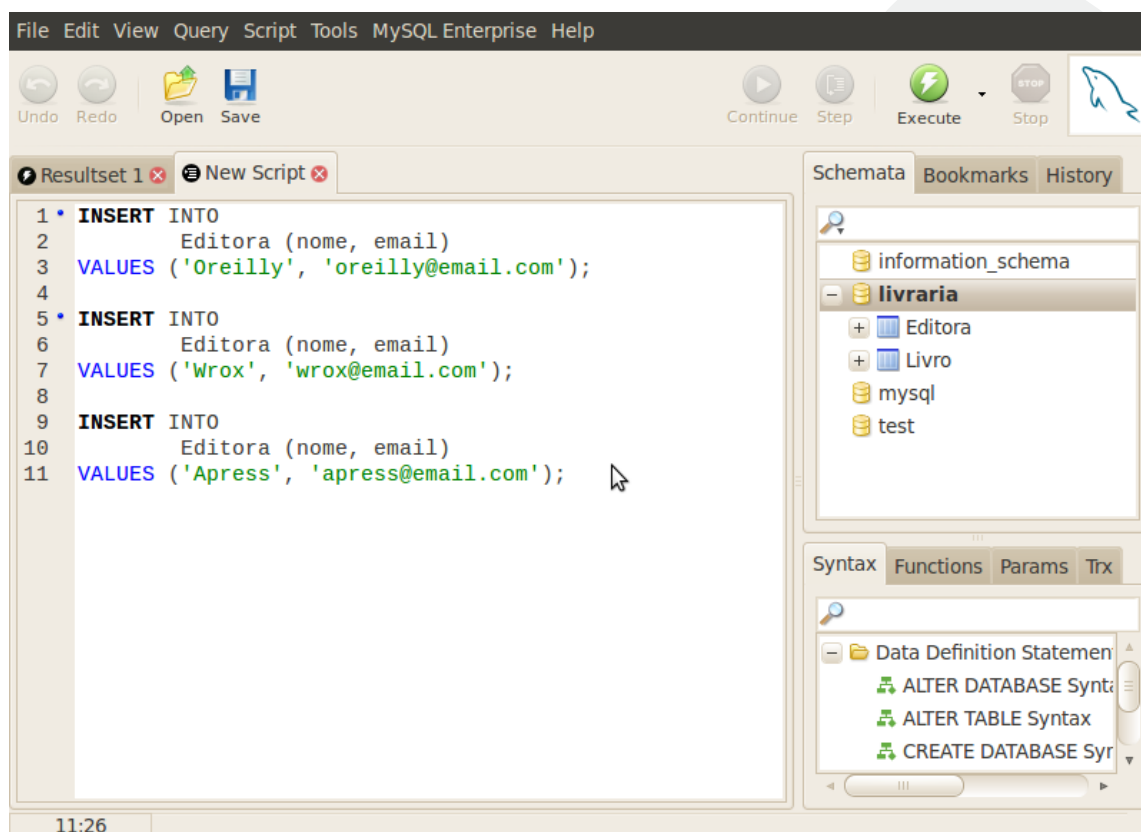
Column	Foreign Column
editora_id	id

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

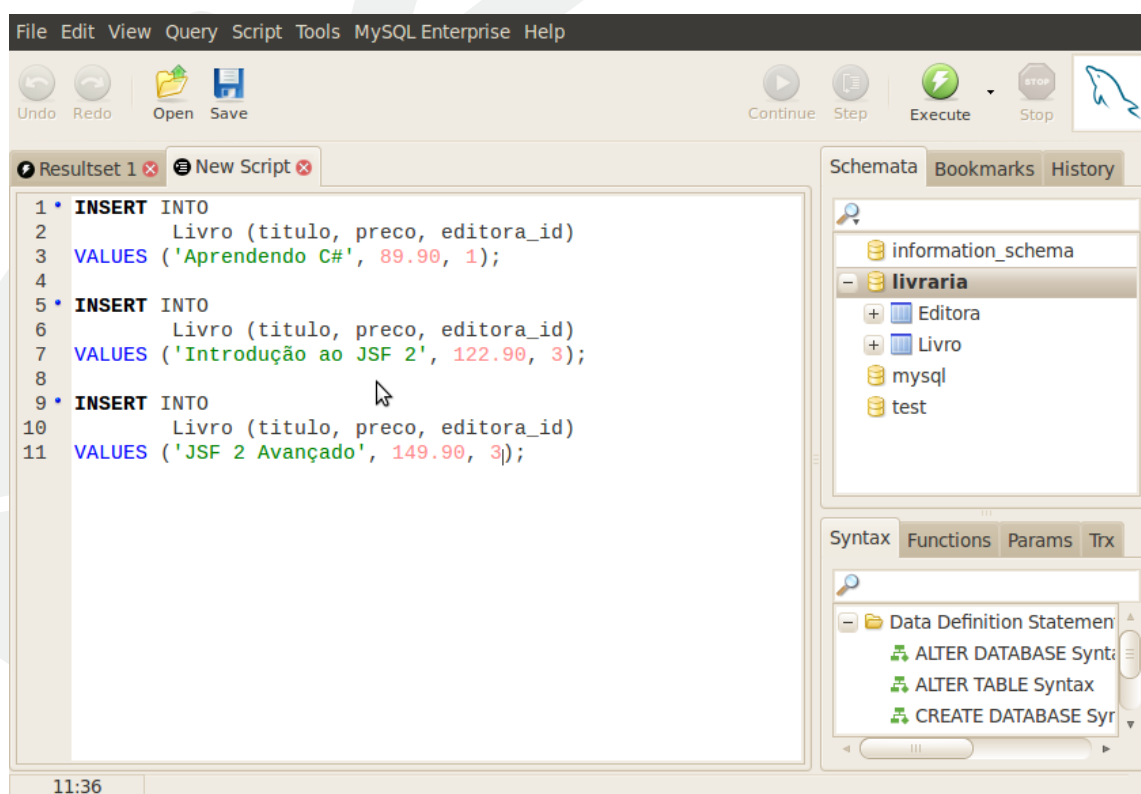
The following SQL commands will be executed to update the edited table. Please review it and press the Execute button to apply it.

```
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 (`editora_id`)
    REFERENCES `Editora` (`id`)
    ON DELETE RESTRICT
    ON UPDATE RESTRICT
)
ENGINE = InnoDB;
```

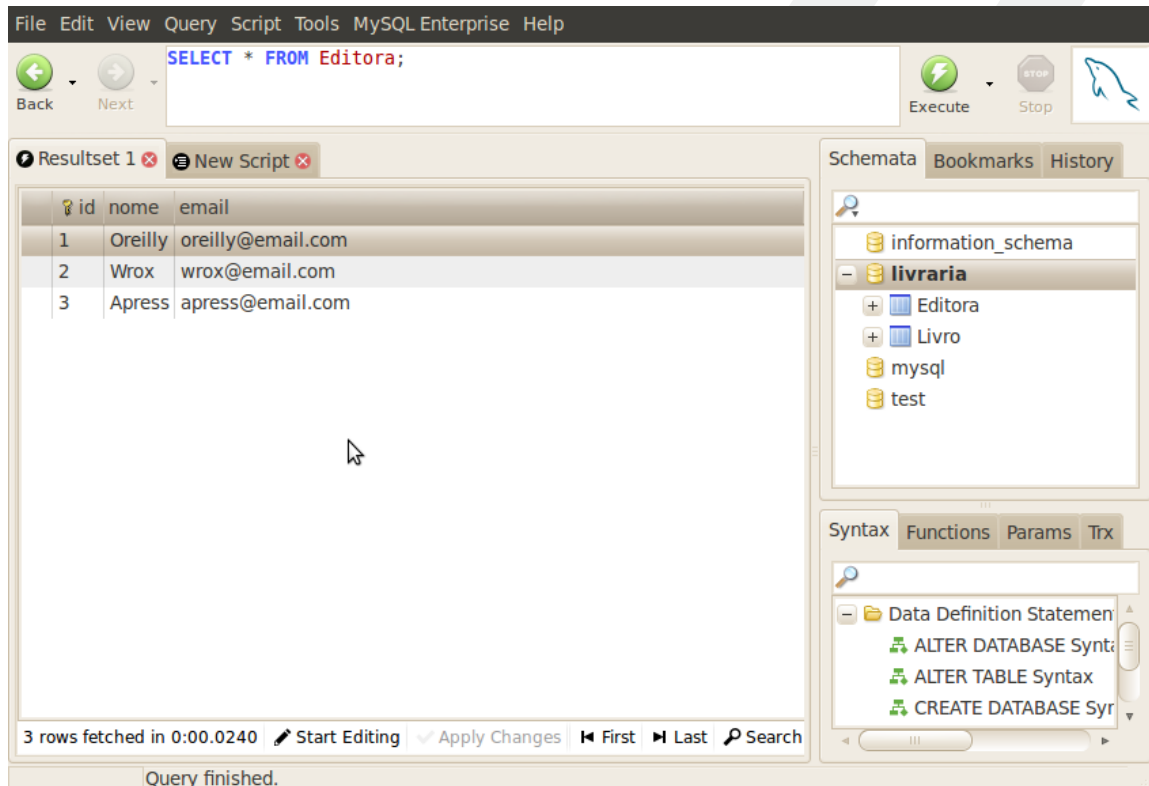
7. Adicione alguns registros na tabela Editora. Veja exemplos na figura abaixo:



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

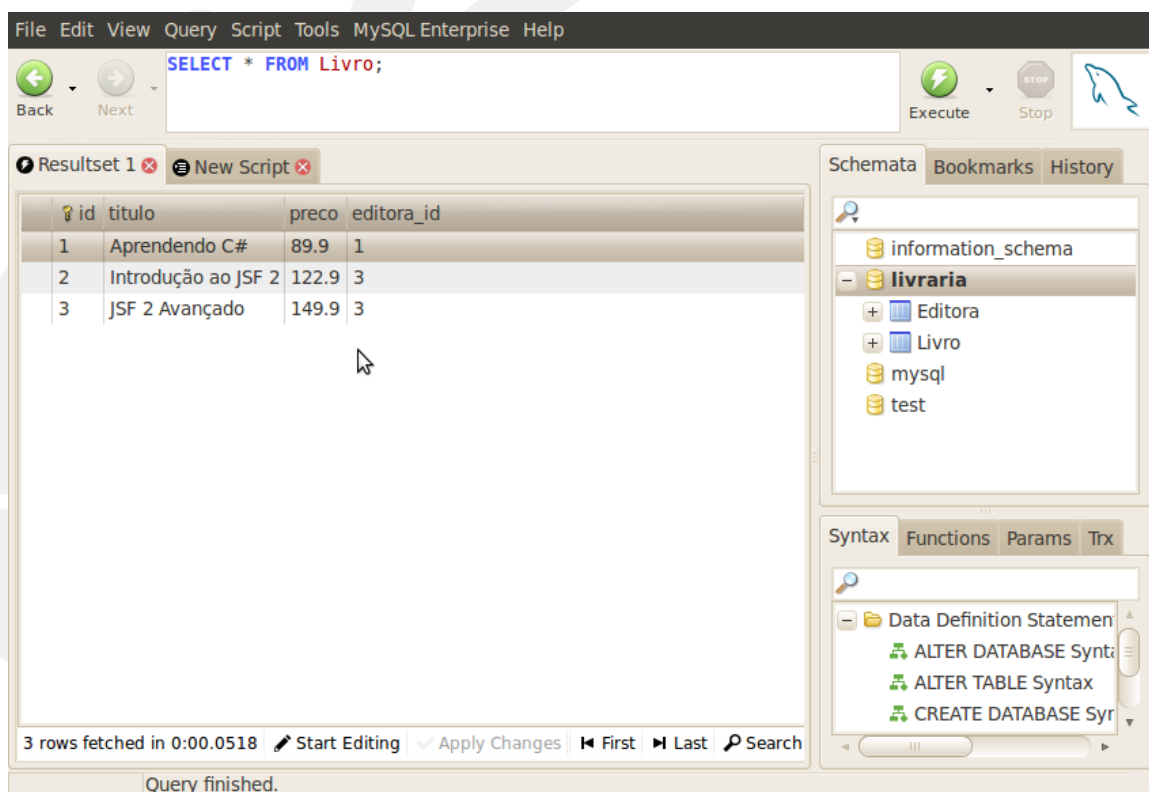


9. Consulte os registros da tabela Editora, e em seguida consulte a tabela Livro. Veja exemplos logo abaixo:



The screenshot shows the MySQL Enterprise console interface. The query editor at the top contains the SQL statement `SELECT * FROM Editora;`. Below the editor, the 'Resultset 1' tab is active, displaying a table with three rows of data. The table has columns 'id', 'nome', and 'email'. The data rows are: (1, Oreilly, oreilly@email.com), (2, Wrox, wrox@email.com), and (3, Apress, apress@email.com). The status bar at the bottom indicates '3 rows fetched in 0:00.0240' and 'Query finished.'

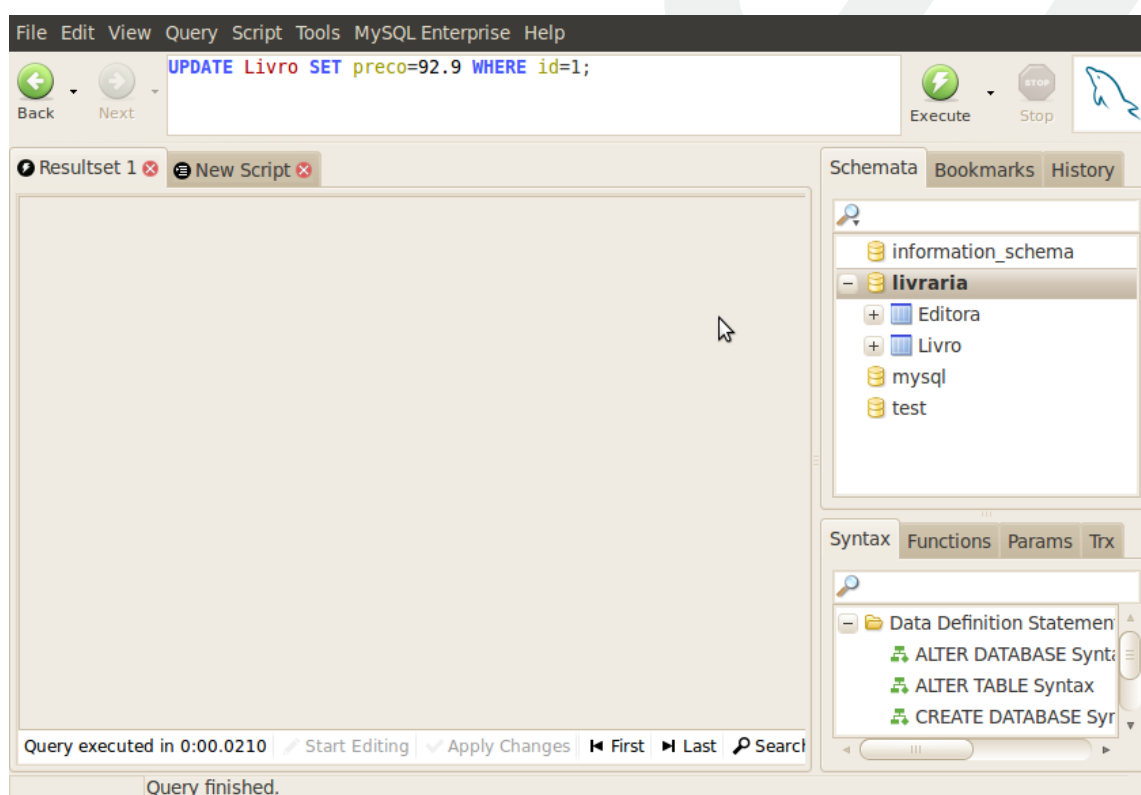
id	nome	email
1	Oreilly	oreilly@email.com
2	Wrox	wrox@email.com
3	Apress	apress@email.com



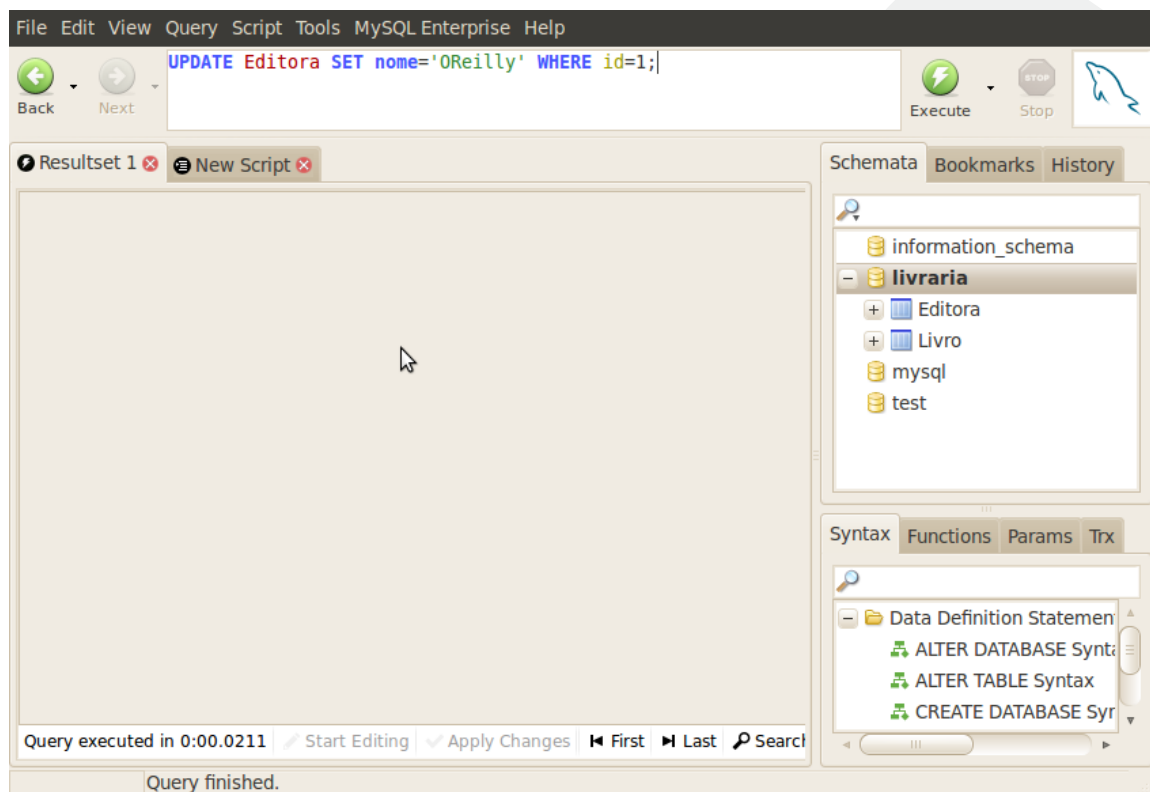
The screenshot shows the MySQL Enterprise console interface. The query editor at the top contains the SQL statement `SELECT * FROM Livro;`. Below the editor, the 'Resultset 1' tab is active, displaying a table with three rows of data. The table has columns 'id', 'titulo', 'preco', and 'editora_id'. The data rows are: (1, Aprendendo C#, 89.9, 1), (2, Introdução ao JSF 2, 122.9, 3), and (3, JSF 2 Avançado, 149.9, 3). The status bar at the bottom indicates '3 rows fetched in 0:00.0518' and 'Query finished.'

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

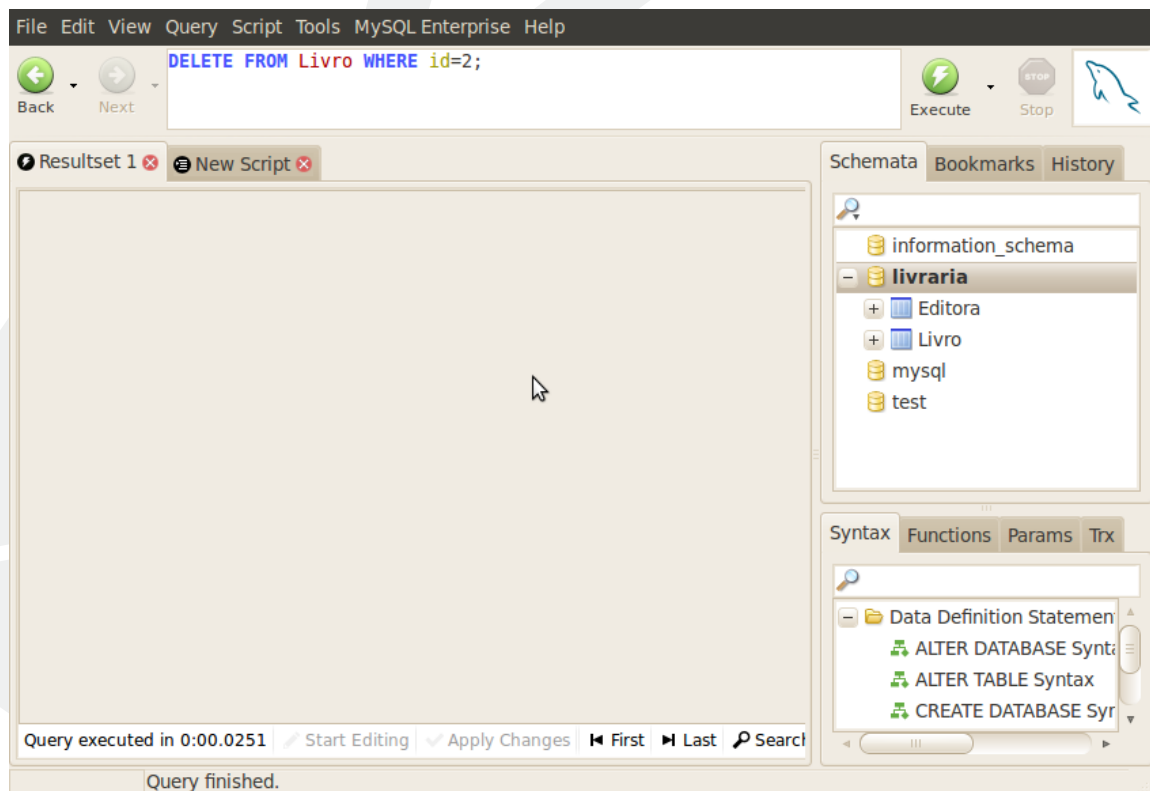
10. Altere alguns dos registros da tabela Livro. Veja o exemplo abaixo:



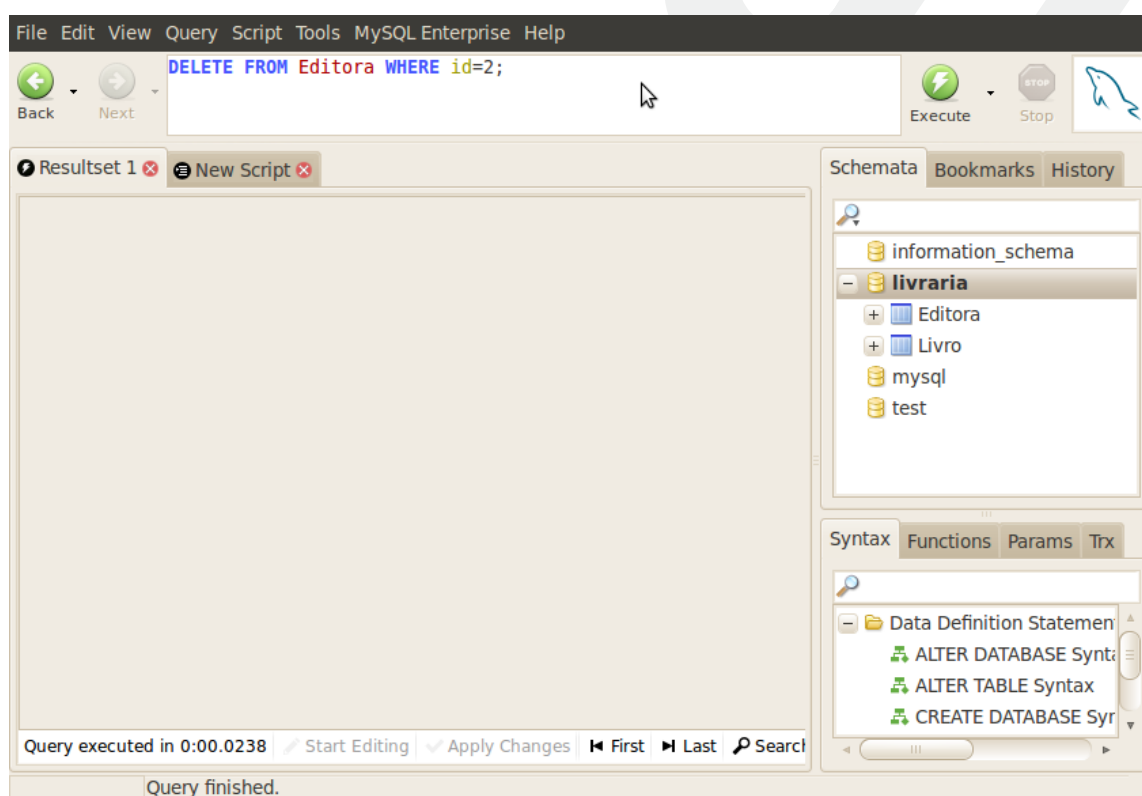
11. Altere alguns dos registros da tabela Editora. Veja o exemplo abaixo:



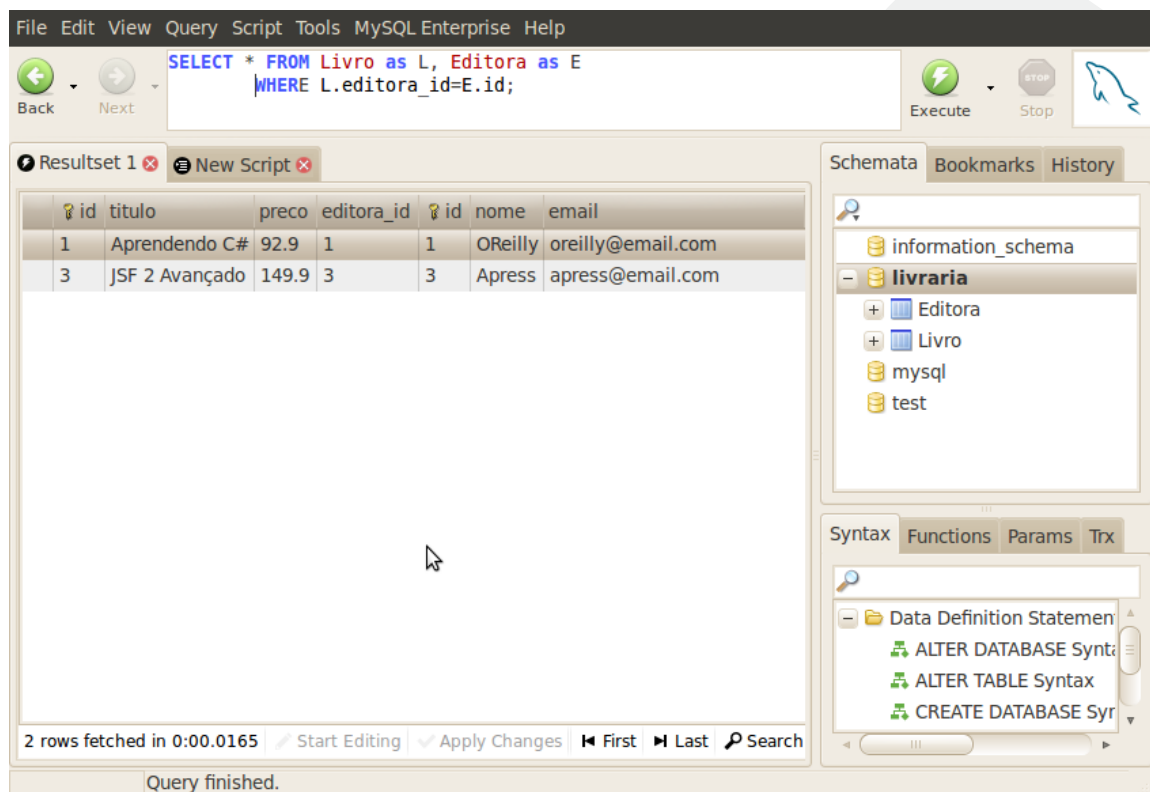
12. Remova alguns registros da tabela Livro. Veja o exemplo abaixo:



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 de uma determinada editora. Veja um exemplo na figura abaixo:



The screenshot shows the MySQL Enterprise GUI interface. The main window displays a query result set for the query: `SELECT * FROM Livro as L, Editora as E WHERE L.editora_id=E.id;`. The result set contains two rows of data. The status bar at the bottom indicates "2 rows fetched in 0:00.0165" and "Query finished."

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

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

Schema: **livraria**

- Editora
- Livro
- mysql
- test

Syntax: **Data Definition Statements**

- ALTER DATABASE Syntax
- ALTER TABLE Syntax
- CREATE DATABASE Syntax



Capítulo 2

JDBC

No capítulo anterior, aprendemos que utilizar bancos de dados é uma boa solução para o armazenamento dos dados de uma aplicação. Entretanto, você deve ter percebido que a interface de utilização do MySQL (e dos outros bancos de dados em geral) não é muito amigável. A desvantagem deste tipo de interface, é que ela exige que o usuário conheça a sintaxe da linguagem SQL para escrever as consultas. Além disso, quando o volume de dados é muito grande, é mais difícil visualizar os resultados.

Na prática uma aplicação com interface simples é desenvolvida para permitir que os usuários do sistema possam manipular os dados do banco, evitando desse modo que um usuário necessite conhecer SQL. Por isso, precisamos fazer com que essa aplicação consiga se comunicar com o banco de dados utilizado no sistema.

2.1 Driver

Como a aplicação precisa conversar com o banco de dados, ela deve trocar mensagens com o mesmo. O formato das mensagens precisa ser definido previamente. Por questões de economia de espaço, cada bit de uma mensagem tem um significado diferente. Resumidamente, o protocolo de comunicação utilizado é binário.

Mensagens definidas com protocolos binários são facilmente interpretadas por computadores. Por outro lado, são complexas para um ser humano compreender. Dessa forma, é mais trabalhoso e mais suscetível a erro desenvolver uma aplicação que converse com um banco de dados através de mensagens binárias.

Para resolver esse problema e facilitar o desenvolvimento de aplicações que devem se comunicar com bancos de dados, as empresas que são proprietárias desses bancos oferecem os **drivers de conexão**.

Os drivers de conexão atuam como “tradutores” de comandos escritos em uma determinada linguagem de programação para comandos no protocolo do banco de dados. Do ponto de vista do desenvolvedor da aplicação, não é necessário conhecer o complexo protocolo binário do banco.

Em alguns casos, o protocolo binário de um determinado banco de dados é fechado. Consequentemente, a única maneira de se comunicar com o banco de dados é através de um driver de conexão.

2.2 JDBC

Suponha que os proprietários dos bancos de dados desenvolvessem os drivers de maneira totalmente independente. Consequentemente, cada driver teria sua própria interface, ou seja, seu próprio conjunto de instruções. Dessa maneira, o desenvolvedor da aplicação precisa conhecer as instruções de cada um dos drivers dos respectivos bancos que ele for utilizar.

Para facilitar o trabalho do desenvolvedor da aplicação, foi criado o **JDBC** (Java Database Connectivity). O JDBC é uma API que generaliza a interface com os banco de dados. Assim, quando uma empresa proprietária de um banco de dados pretende desenvolver um driver para ser utilizado com a linguagem Java, ela segue a especificação JDBC com o intuito de incentivar a adoção do driver.

2.3 Instalando o Driver JDBC do MySQL Server

O driver oficial JDBC desenvolvido para funcionar com o MySQL se chama **MySQL Connector/J**. É necessário fazer o download do driver na seguinte url:

`http://www.mysql.com/downloads/connector/j/`.

É só descompactar o arquivo e depois incluir o **jar** com o driver no BUILD PATH da aplicação.

2.4 Criando uma conexão

Com o driver de conexão JDBC adicionado ao projeto, já é possível criar uma conexão com o banco de dados correspondente. Abaixo, estão os passos necessários para criar uma conexão.

- Escolher o driver de conexão;
- Definir a localização do banco de dados;
- Informar o nome da base de dados;
- Ter um usuário e senha cadastrados no banco de dados.

As informações sobre o driver, localização e nome da base de dados são definidas no que chamamos de **url de conexão** (ou string de conexão). O usuário e a senha, informamos no momento de criar uma conexão. Para criar esta conexão, utilizamos as classes **DRIVERMANAGER** (que instancia a conexão) e **CONNECTION** (que armazena a nossa conexão), ambas presentes no pacote **JAVA.SQL**.

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

2.5 Inserindo registros

Estabelecida a conexão com o banco de dados, já podemos executar comandos. Como primeiro exemplo, iremos inserir registros em uma tabela. O primeiro passo para executar um comando é defini-lo em linguagem SQL.

```
1 string textoDoComando = "INSERT INTO Editora (nome, email)" +  
2   "VALUES ('K19', 'contato@k19.com.br');";
```

Em seguida, devemos “pedir” para uma conexão JDBC através do método `PREPARESTATEMENT()` criar o comando que queremos executar. Este método devolve um objeto da interface `PREPAREDSTATEMENT`. O comando não é executado na chamada do método `PREPARESTATEMENT()` e sim posteriormente quando o método `EXECUTE()` for utilizado.

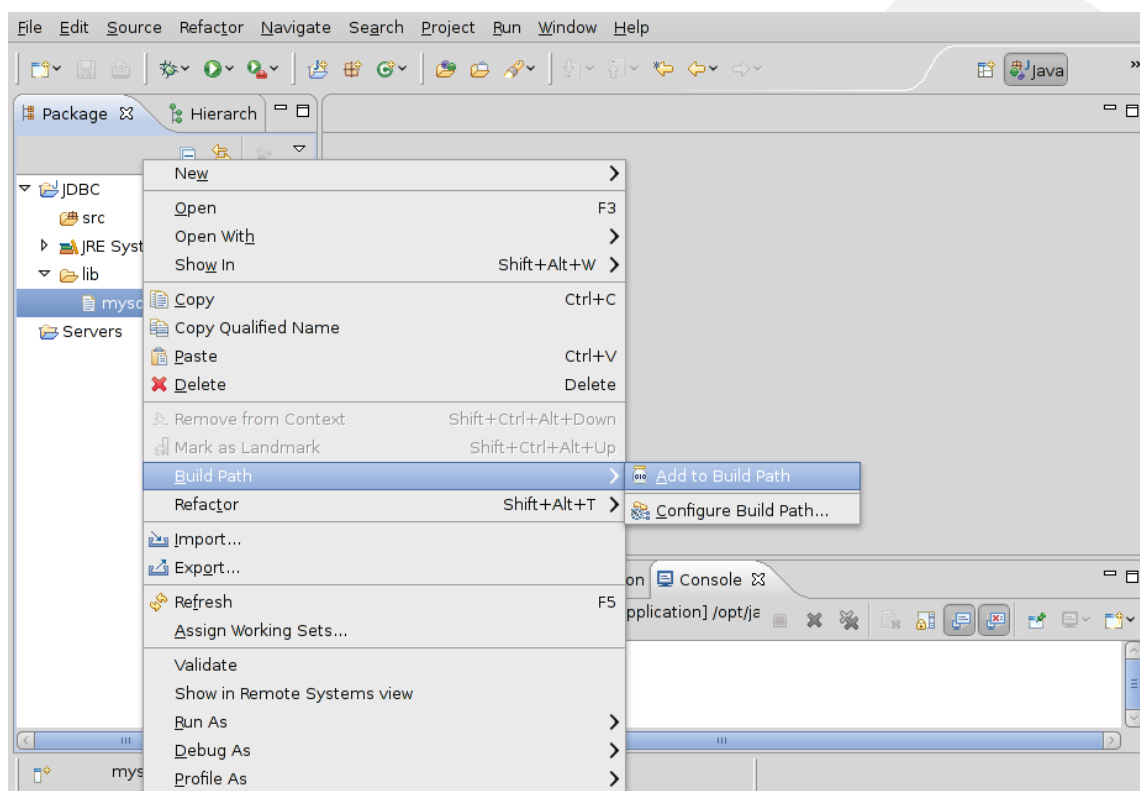
```
1 // criando comando  
2 PreparedStatement comando = conexao.prepareStatement(textoDoComando);  
3 // executando o comando  
4 comando.execute();  
5 comando.close();
```

Podemos utilizar a mesma conexão para executar diversos comandos. Quando não desejamos executar mais comandos, devemos fechar a conexão através do método `CLOSE()`. Fechar as conexões que não são mais necessárias é importante pois os SGBD's possuem um número limitado de conexões abertas.

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

2.6 Exercícios

1. Crie um projeto java no eclipse 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 pasta **lib** do projeto **JDBC**.
4. Adicione o arquivo **MYSQL-CONNECTOR-JAVA-5.1.1.JAR** ao **build path** (veja figura).



5. Crie uma nova classe, com o nome **InserEditora**, 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
8     public static void main(String[] args) {
9         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
10        String usuario = "root";
11        String senha = "root";
12
13        Scanner entrada = new Scanner(System.in);
14
15        try {
16            System.out.println("Abrindo conexao...");
17            Connection conexao =
18                DriverManager.getConnection(stringDeConexao, usuario, senha);
19
20            System.out.println("Digite o nome da editora: ");
21            String nome = entrada.nextLine();
22
23            System.out.println("Digite o email da editora: ");
24            String email = entrada.nextLine();
25
26            String textoDoComando = "INSERT INTO Editora (nome, email) " +
27                "VALUES ('" + nome + "', '" + email + "')";
28
29            PreparedStatement comando = conexao.prepareStatement(textoDoComando);
30
31            System.out.println("Executando comando...");
32            comando.execute();
33
34            System.out.println("Fechando conexao...");
35            conexao.close();
36        }
37        catch (Exception e) {
38            e.printStackTrace();
39        }
40    }
41 }
```

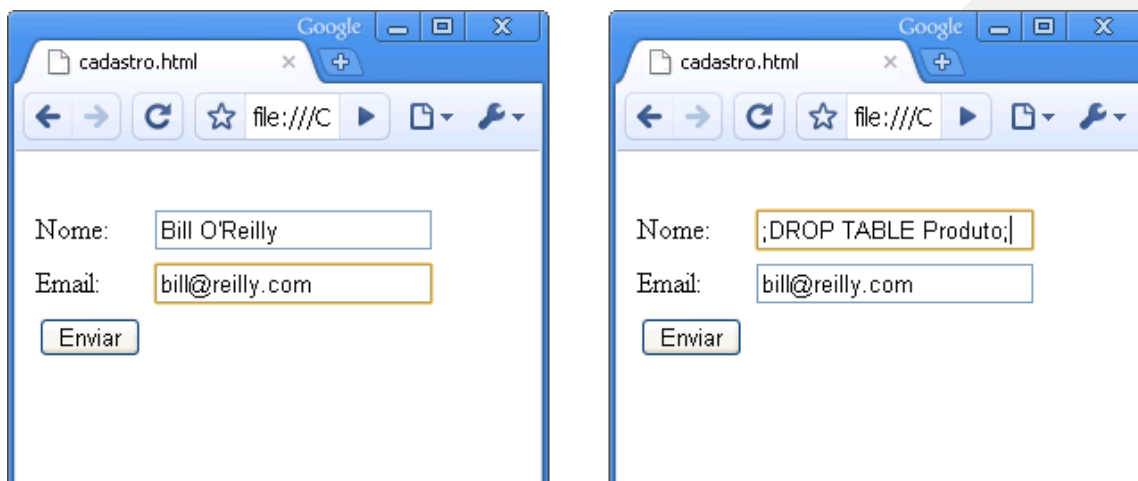
Rode e veja se o registro foi inserido com sucesso na base de dados.

6. (Opcional) Analogamente, ao exercício anterior crie um programa para inserir livros.

2.7 SQL Injection

Apesar de funcional, a implementação da inserção de registros feita anteriormente apresenta uma falha grave. Os dados obtidos do usuário através do teclado não são tratados antes de serem enviados para o banco de dados.

Esses dados podem conter algum carácter especial que altere o comportamento desejado da consulta impedindo que algum registro seja inserido corretamente ou até abrindo uma brecha para que um usuário mal intencionado execute alguma consulta SQL para utilizar de maneira inadequada os dados da nossa aplicação.



O problema de SQL Injection pode ser resolvido manualmente. Basta fazer “escape” dos caracteres especiais, por exemplo: ponto-e-vírgula e apóstrofo. No MySQL Server, os caracteres especiais devem ser precedidos pelo carácter “\”. Então seria necessário acrescentar \ em todas as ocorrências de caracteres especiais nos valores passados pelo usuário.

A desvantagem desse processo é que, além de trabalhoso, é diferente para cada banco de dados, pois o “\” não é padronizado e cada banco tem o seu próprio método de escape de caracteres especiais.

Para tornar mais prática a comunicação com o banco de dados, o próprio driver faz o tratamento das sentenças SQL. Esse processo é denominado **sanitize**.

```
1 // lendo a entrada feita pelo usuario
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 // texto do comando inclui parâmetros
9 String textoDoComando = "INSERT INTO Editora (nome, email) " +
10     "VALUES (?, ?)";
11
12 // criação e adição de parâmetros ao comando
13 PreparedStatement comando = conexao.prepareStatement(textoDoComando);
14 comando.setString(1, nome);
15 comando.setString(2, email);
```

Observe que a sentença SQL foi definida com parâmetros através do carácter “?”. Antes de executar o comando, é necessário atribuir valores aos parâmetros. Isso é feito com o método SETSTRING, que recebe o índice (posição) do parâmetro na consulta e o valor correspondente. De maneira similar, temos os métodos SETINT, SETDOUBLE, SETDATE, etc, variando conforme o tipo do campo que foi definido no banco.

Os métodos acima mostrados, realizam a tarefa de “sanitizar”(limpar) os valores enviados pelo usuário.

2.8 Exercícios

7. Tente causar um erro de SQL Injection na classe feita no exercício de inserir editoras. (Dica: tente entradas com aspas simples)
8. Altere o código para eliminar o problema do SQL Injection. Você deve deixar a classe com o código abaixo:

```
1 public class InserirEditora {
2     public static void main(String[] args) {
3
4         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
5         String usuario = "root";
6         String senha = "root";
7
8         Scanner entrada = new Scanner(System.in);
9
10        try {
11            System.out.println("Abrindo conexao...");
12            Connection conexao = DriverManager.getConnection(stringDeConexao, usuario, ↵
                senha);
13
14            System.out.println("Digite o nome da editora: ");
15            String nome = entrada.nextLine();
16
17            System.out.println("Digite o email da editora: ");
18            String email = entrada.nextLine();
19
20            String textoDoComando = "INSERT INTO Editora (nome, email) " +
21                "VALUES (?, ?)";
22
23            PreparedStatement comando = conexao.prepareStatement(textoDoComando);
24            comando.setString(1, nome);
25            comando.setString(2, email);
26
27            System.out.println("Executando comando...");
28            comando.execute();
29
30            System.out.println("Fechando conexao...");
31            conexao.close();
32        }
33        catch (Exception e) {
34            e.printStackTrace();
35        }
36    }
37 }
```

9. Agora tente causar novamente o problema de SQL Injection ao inserir novas editoras.

2.9 Listando registros

O processo para executar um comando de consulta é bem parecido com o processo de inserir registros.

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

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`, 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");
```

O código acima mostra como os campos do primeiro registro da consulta são recuperados. Agora, para recuperar os 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");
```

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 }
```

2.10 Exercícios

10. Insira algumas editoras utilizando a classe `INSEREEDITORAS` que você criou nos exercícios acima.
11. Adicione uma nova classe ao projeto chamada **ListaEditoras**. O objetivo é listar as editoras que foram salvas no banco. Adicione o seguinte código à esta classe.

```
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
8     public static void main(String[] args) {
9         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
10        String usuario = "root";
11        String senha = "root";
12
13        try {
14            System.out.println("Abrindo conexao...");
15            Connection conexao =
16                DriverManager.getConnection(stringDeConexao, usuario, senha);
17
18            String textoDoComando = "SELECT * FROM Editora;";
19
20            PreparedStatement comando = conexao.prepareStatement(textoDoComando);
21
22            System.out.println("Executando comando...");
23            ResultSet resultado = comando.executeQuery();
24
25            System.out.println("Resultados encontrados: \n");
26            while (resultado.next()) {
27                System.out.printf("%d : %s - %s\n",
28                    resultado.getInt("id"),
29                    resultado.getString("nome"),
30                    resultado.getString("email"));
31            }
32
33            System.out.println("\nFechando conexao...");
34            conexao.close();
35        }
36        catch (Exception e) {
37            e.printStackTrace();
38        }
39    }
40 }
```

2.11 Fábrica de conexões (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, em cada ocorrência da url de conexão. A probabilidade de algum ponto não ser corrigido é grande.

Para diminuir o trabalho de manutenção, nós poderíamos criar uma classe responsável pela criação e distribuição de conexões. Nessa e somente nessa classe estaria definida a url de conexão. Dessa forma, qualquer alteração no modo em que nos conectamos à base de dados, só acarreta mudanças nesta classe.

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

Agora podemos obter uma nova conexão apenas chamando `FABRICADECONEXAO.CRIACONEXAO()`. 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.

2.12 Exercícios

12. Adicione uma nova classe chamada `FABRICADECONEXAO` e adicione o seguinte código:

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.SQLException;
4
5 public class FabricaDeConexao {
6
7     public static Connection CriaConexao() {
8         String stringDeConexao = "jdbc:mysql://localhost:3306/livraria";
9         String usuario = "root";
10        String senha = "root";
11
12        Connection conexao = null;
13
14        try {
15            conexao = DriverManager.getConnection(stringDeConexao, usuario, senha);
16        } catch (SQLException e) {
17            e.printStackTrace();
18        }
19        return conexao;
20    }
21 }
```

13. Altere as classes `INSEREEDITORA` e `LISTAEDITORAS` para que elas utilizem a fábrica de conexão. Execute-as novamente.
14. (Opcional) Implemente um teste que remove uma editora pelo **id**.
15. (Opcional) Implemente um teste que altera os dados de uma editora pelo **id**.

Capítulo 3

JPA 2 e Hibernate

3.1 Múltiplas sintaxes da linguagem SQL

No capítulo anterior, você aprendeu a utilizar a especificação JDBC para fazer uma aplicação Java interagir com um banco de dados. Essa interação é realizada através de consultas escritas em SQL. Uma desvantagem dessa abordagem, é que a sintaxe da linguagem SQL, apesar de parecida, pode variar conforme o banco de dados que está sendo utilizado. Desse modo, os desenvolvedores teriam que aprender as diferenças entre as sintaxes do SQL correspondentes aos banco de dados que ele utilizará.

Seria bom se, ao invés de programar direcionado a um determinado banco de dados, pudéssemos programar de uma maneira mais genérica, voltado à uma interface ou especificação, assim poderíamos escrever o código independente de SQL.

3.2 Orientação a Objetos VS Modelo Entidade Relacionamento

Outro problema na comunicação entre uma aplicação Java e um banco de dados é o conflito de paradigmas. O banco de dados é organizado seguindo o modelo entidade relacionamento, enquanto as aplicações Java, geralmente, utilizam o paradigma orientado a objetos.

A transição de dados entre o modelo entidade relacionamento 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.

3.3 Ferramentas ORM

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

Outra consequência, ao utilizar uma ferramenta de ORM, é que não é necessário escrever consultas em SQL, pois a própria ferramenta gera as consultas de acordo com a sintaxe da

linguagem SQL correspondente ao banco que está sendo utilizado.

A principal ferramenta ORM para Java utilizada no mercado de TI é o Hibernate. Mas, existem outras que possuem o mesmo objetivo.

3.4 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 de 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: **TopLink**, **EclipseLink** e **OpenJPA**. Optamos por utilizar o Hibernate por ele ser o mais antigo e mais utilizado atualmente.

Caso você queira utilizar outro framework ORM, poderá aplicar os conceitos aqui aprendidos justamente por que 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.

3.5 Configuração

Antes de começar a utilizar o Hibernate, é necessário baixar no 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 esta:

(<http://www.hibernate.org/>)

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 xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↵
5     ns/persistence/persistence_1_0.xsd"
6   version="1.0">
7
8   <persistence-unit name="K19" transaction-type="RESOURCE_LOCAL">
9     <provider>org.hibernate.ejb.HibernatePersistence</provider>
10    <properties>
11      <property name="hibernate.dialect" value="org.hibernate.dialect.↵
12        MySQL5InnoDBDialect"/>
13      <property name="hibernate.hbm2ddl.auto" value="create"/>
14      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver↵
15        "/>
16      <property name="javax.persistence.jdbc.user" value="usuario"/>
17      <property name="javax.persistence.jdbc.password" value="senha"/>
18      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://↵
19        localhost:3306/k19"/>
20    </properties>
21  </persistence-unit>
22</persistence>
```

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

3.6 Mapeamento

Um dos principais objetivos dos frameworks ORM é estabelecer o mapeamento entre os conceitos do modelo orientado a objetos e os conceitos do modelo entidade relacionamento. Este mapeamento pode ser definido através de xml ou de maneira mais prática com anotações Java. Quando utilizamos anotações, evitamos a criação de extensos arquivos em xml.

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 que 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. E 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 que deve ser utilizado na tabela. Veja o exemplo:

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

@Column Utilizado para alterar o nome da coluna que será utilizado 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;
```

@Transient Serve para indicar um atributo que 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     // ...
```

3.7 Gerando o banco

Uma das vantagens de 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 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 configurando a propriedade **HIBERNATE.HBM2DDL.AUTO** no arquivo **PERSISTENCE.XML**. Podemos, por exemplo, fazer com

que o Hibernate sempre sobrescreva as tabelas existentes, basta configurar 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"/>
```

3.8 Exercícios

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**.
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 xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
5   version="1.0">
6
7   <persistence-unit name="livraria" 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="root"/>
15       <property name="javax.persistence.jdbc.password" value="root"/>
16       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/livraria"/>
17     </properties>
18   </persistence-unit>
19 </persistence>

```

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 }

```

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**.

```

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

```

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");
5
6         factory.close()
7     }
8 }
```

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

3.9 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();
```

3.9.1 Persistindo

Para armazenar as informações de um objeto no banco de dados basta 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);
```

3.9.2 Buscando

Para obter um objeto que contenha informações do banco de dados basta utilizar o método `FIND()` ou o `GETREFERENCE()` do `ENTITYMANAGER`.

```
1 Editora editora1 = manager.find(Editora.class, 1L);
2 Editora editora2 = manager.getReference(Editora.class, 2L);
```

A diferença entre os dois métodos básicos de busca `FIND()` e `GETREFERENCE()` é que o primeiro recupera os dados desejados imediatamente já o segundo posterga até a primeira chamada de um método `GET` do objeto.

3.9.3 Removendo

Para remover um registro correspondente a um objeto basta utilizar o método REMOVE() do ENTITYMANAGER.

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

3.9.4 Atualizando

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

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

3.9.5 Listando

Para obter uma listagem com todos os objetos referentes aos registros de uma tabela, devemos utilizar a linguagem de consulta do JPA, a JPQL que é muito parecida com a linguagem SQL. A vantagem do JPQL em relação ao SQL é que a sintaxe é a mesma para bancos de dados diferentes.

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

3.9.6 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();
```

Com a transação aberta podemos sincronizar os dados com o banco através do método FLUSH() ou COMMIT().

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

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

3.10 Exercícios

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");
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         factory.close();
25     }
26 }
```

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");
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 }

```

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

```

```
1 EntityManagerFactory factory = Persistence.createEntityManagerFactory("K12");
2
3 EntityManager manager = factory.createEntityManager();
4
5 EditoraRepository editoraRepository = new EditoraRepository(manager);
6
7 List<Editora> editoras = editoraRepository.buscaTodas();
```

3.12 Exercícios

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        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 }
```

18. Altere a classe **InserEditoraComJPA** 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");
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         factory.close();
27     }
28 }
```

19. (Opcional) Altere a classe **ListaEditorasComJPA** para que ela utilize o repositório de editoras.

Capítulo 4

Web Container

4.1 Necessidades de uma aplicação web

As aplicações web são acessadas pelos navegadores (browsers). A comunicação entre os navegadores e as aplicações web é realizada através de requisições e respostas definidas pelo protocolo **HTTP**. Portanto, ao desenvolver uma aplicação web, devemos estar preparados para receber requisições HTTP e enviar respostas HTTP.

Além disso, na grande maioria dos casos, as aplicações web devem ser acessadas por diversos usuários simultaneamente. Dessa forma, o desenvolvedor web precisa saber como permitir o acesso simultâneo.

Outra necessidade das aplicações web é gerar conteúdo dinâmico. Por exemplo, quando um usuário de uma aplicação de email acessa a sua caixa de entrada, ele deseja ver a listagem atualizada dos seus emails. Portanto, é fundamental que a listagem dos emails seja gerada no momento da requisição do usuário. O desenvolvedor web precisa utilizar algum mecanismo eficiente que permita que o conteúdo que os usuários requisitam seja gerado dinamicamente.

Trabalhar diretamente com as requisições e repostas HTTP e criar um mecanismo eficiente para permitir o acesso simultâneo e para gerar conteúdo dinâmico não são tarefas simples. Na verdade, é extremamente trabalhoso implementar essas características. Por isso, a plataforma Java oferece uma solução para diminuir o trabalho no desenvolvimento de aplicações web.

4.2 Web Container

Uma aplicação web Java deve ser implantada em um **Web Container** para obter os recursos fundamentais que as aplicações web necessitam. Um Web Container é responsável pelo envio e recebimento de mensagens HTTP, permite que as aplicações implantadas nele sejam acessadas simultaneamente por múltiplos usuários de uma maneira eficiente e oferece mecanismos para que as aplicações gerem conteúdo dinamicamente.

Os dois Web Container's 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.

4.3 Especificação Java EE

Como é comum na plataforma Java, para padronizar a interface dos recursos oferecidos pelos Web Container's, especificações foram definidas. Essas especificações fazem parte do conjunto de especificações do **Java EE**. O Java EE é uma especificação que agrupa diversas outras especificações.

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

4.4 Exercícios

1. Na Área de Trabalho, entre na pasta **K19-Arquivos** e copie **glassfish-3.0.1-with-hibernate.zip** para o seu Desktop. Descompacte este arquivo na própria Área de Trabalho.
2. Ainda na Área de Trabalho, entre na pasta **glassfishv3/glassfish/bin** e execute o script **startserv** para executar o glassfish.
3. Verifique se o glassfish está executando através de um navegador acessando a url:
`http://localhost:8080.`
4. Pare 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. Execute o glassfish pela view **servers** e verifique se ele está funcionando acessando através de um navegador a url:
`http://localhost:8080.`
7. Pare o glassfish pela view **servers**.

4.5 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/ *pasta raiz pode ter qualquer nome*
 - ▷ WEB-INF/
 - ▷ classes/
 - ▷ lib/
 - ▷ web.xml

A pasta **K19-App** é raiz da nossa aplicação, o nome dessa pasta pode ser definido pelo desenvolvedor. A pasta **WEB-INF** deve ser criada dentro da pasta raiz de todas as aplicações Web Java, o conteúdo dessa pasta não pode ser acessado diretamente pelos usuários. A pasta **classes** deve ser criada dentro da pasta WEB-INF, o código compilado das aplicações Web Java deve ser salvo nessa pasta. A pasta **lib** deve ser criada dentro da pasta WEB-INF, todos os jar's das bibliotecas que serão utilizadas devem ser colocados nessa pasta. O arquivo **web.xml** contém configurações do Web Container e deve ser criado na pasta WEB-INF. Os arquivos dentro da pasta raiz da aplicação mas fora da pasta WEB-INF podem ser acessados diretamente pelos usuários através de um navegador.

As IDE's já criam toda a estrutura de pastas exigidas pelos Web Container's. Então, na prática, não temos o trabalho de criar esses diretórios manualmente.

4.6 Exercícios

8. Crie um projeto no eclipse do tipo **Dynamic Web Project** chamado **App-Web-Java** selecionando na última tela a opção **Generate web.xml deployment descriptor**.
9. Execute o projeto no glassfish clicando com o botão direito no nome do projeto e escolhendo a opção **Run on Server** dentro de **Run As**.
10. Verifique o funcionamento da nossa aplicação acessando a url:
`http://localhost:8080/K19-App/` através de um navegador.

4.7 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 da nossa aplicação deve ser gerado. Há duas maneiras fundamentais de gerar conteúdo dinâmico: programando uma **Servlet** ou um **JSP**. Na verdade, os JSPs são “traduzidos” para Servlets automaticamente pelos Web Container's. Então, escrever um JSP é apenas uma maneira diferente de escrever uma Servlet.

4.8 Servlet

Para criar uma Servlet, podemos seguir os seguintes passos básicos:

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 de 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 }
```

Quando um usuário fizer uma requisição para url definida através da anotação `@WEB-SERVLET`, o método `service()` será executado. Esse método recebe dois parâmetros: o primeiro é uma referência para um objeto da classe **HttpServletRequest** que guarda todos os dados da requisição HTTP realizada pelo navegador do usuário; o segundo é uma referência para um objeto da classe **HttpServletResponse** que permite que a resposta HTTP que será enviada para o navegador do usuário seja construída pela aplicação.

4.8.1 Inserindo conteúdo na resposta

Para inserir conteúdo na resposta HTTP que será enviada para o navegador do usuário, devemos utilizar o método `getWriter()`. Em geral, o conteúdo inserido na resposta HTTP é texto **HTML**.

```
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 }
```

4.9 Exercícios

11. Crie um pacote chamado **servlets** no projeto **K19-App**.
12. Crie uma classe chamada **OlaMundo** no pacote **servlets** da seguinte forma:

```
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 }
```

13. Verifique o funcionamento da Servlet acessando através de um navegador a url:

`http://localhost:8080/K19-App/OlaMundo`

4.10 JSP

Outra maneira de criar uma Servlet é escrever um JSP. Lembrando que os JSPs são “traduzidos” para Servlets automaticamente pelos Web Containers.

Para criar um JSP basta adicionar um arquivo **.jsp**.

```
1 <!-- olaMundo.jsp -->
2 <html>
3   <body>
4     <!-- scriptlet -->
5     <% java.util.Random geradorDeNumeros = new java.util.Random(); %>
6     <!-- expression -->
7     <%= geradorDeNumeros.nextInt(100) %>
8   </body>
9 </html>
```

Um arquivo JSP pode conter texto HTML e código Java. O código Java pode aparecer em diversos lugares. Por exemplo, dentro de **scriptlets** ou **expressions**.

4.11 Exercícios

14. Crie um arquivo JSP chamado **olaMundo.jsp** na pasta **WebContent** com o seguinte conteúdo.

```
1 <!-- olaMundo.jsp -->
2 <html>
3   <body>
4     <!-- scriptlet -->
5     <% java.util.Random geradorDeNumeros = new java.util.Random(); %>
6     <!-- expression -->
7     <%= geradorDeNumeros.nextInt(100) %>
8   </body>
9 </html>
```

15. Verifique o funcionamento do JSP acessando através de um navegador a url:

`http://localhost:8080/K19-App/olaMundo.jsp`

4.12 Frameworks

Hoje em dia, é improvável que uma empresa decida começar um projeto utilizando diretamente Servlets e JSPs pois a produtividade é pequena e a manutenção é difícil. Por isso bibliotecas que definem um modelo de programação baseado no padrão **MVC** são utilizadas nos projetos. Essas bibliotecas são os chamados **Frameworks web**. Eis uma lista dos principais Frameworks web para aplicações web Java:

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

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



Capítulo 5

Visão Geral do JSF 2

O JSF 2 oferece muitos recursos para o desenvolvimento de uma aplicação web Java. Cada um desses recursos por si só já são suficientemente grandes e podem ser abordados em separado.

Porém, no primeiro contato com JSF 2, é interessante ter uma visão geral dos recursos principais e do relacionamento entre eles sem se aprofundar em muito nos detalhes individuais de cada recurso.

Portanto, neste capítulo, mostraremos de forma sucinta e direta o funcionamento e os conceitos principais do JSF 2. Nos próximos capítulos, discutiremos de maneira mais detalhada as diversas partes do JSF 2.

5.1 Aplicação de exemplo

Inspirados na sorte de um dos nossos alunos que ganhou na Loto Mania utilizando um programa que ele fez baseado em algumas dicas que o instrutor Rafael Cosentino deu a ele para gerar números aleatórios em Java, vamos montar uma pequena aplicação em JSF 2 que gera apostas de loteria.

5.2 Managed Beans

Os Managed Beans são objetos utilizados nas aplicações JSF e possuem três responsabilidades principais:

1. Receber os dados enviados pelos usuários através das telas da aplicação.
2. Executar as lógicas para tratar as requisições dos usuários.
3. Disponibilizar os dados que devem ser apresentados nas telas da aplicação.

Para definir o funcionamento de um Managed Bean no JSF 2, devemos seguir os seguintes passos:

1. Criar uma classe com a anotação **@ManagedBean**.

2. Definir atributos com os correspondentes getters e setters para poder receber dados das telas ou enviar dados para as telas.
3. Definir métodos para implementar as lógicas de tratamento das possíveis requisições dos usuários.

5.2.1 GeradorDeApostasBean

Na aplicação que gera apostas de loteria, devemos criar um Managed Bean para receber alguns parâmetros que devem ser definidos pelos usuários para gerar as apostas corretamente.

```
1 @ManagedBean
2 public class GeradorDeApostasBean {
3     private int quantidadeDeNumeros;
4
5     private int tamanhoDaAposta;
6
7     private int quantidadeDeApostas;
8
9     // getters e setters
10 }
```

Devemos acrescentar no GERADORDEAPOSTASBEAN um método para implementar a lógica de gerar as apostas. Este método deve devolver no final o “nome” da tela que apresentará as apostas geradas para os usuários.

```
1 public String geraApostas() {
2     // Aqui deve ser implementa a lógica para gerar as apostas
3     return "lista-de-apostas";
4 }
```

Por fim, devemos adicionar um atributo no GERADORDEAPOSTASBEAN para disponibilizar as apostas geradas para a tela que irá apresentá-las aos usuários.

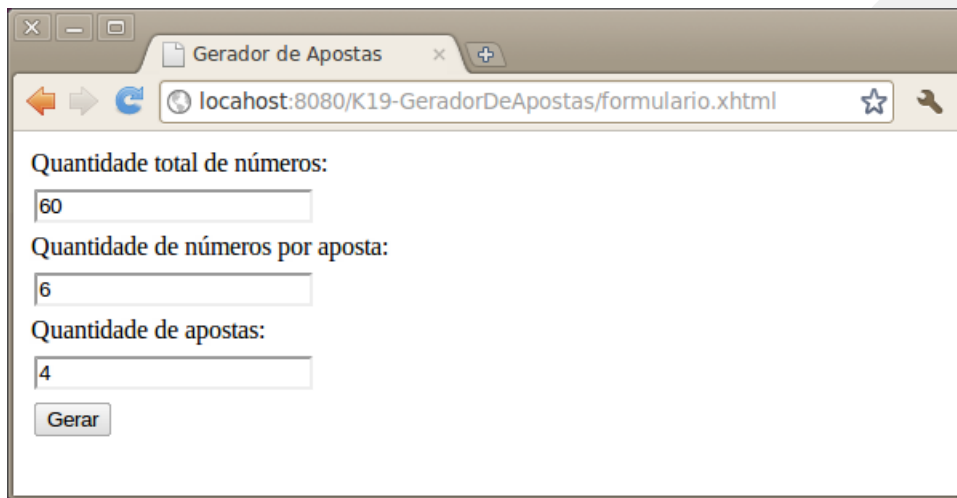
```
1 private List<List<Integer>> apostas;
2
3 // getters e setters
```

5.3 Facelets e Componentes Visuais

As telas das aplicações JSF 2 podem ser definidas através de arquivos **xhtml**. Esses arquivos são processados pela engine do **Facelets** que faz parte do JSF 2. Os componentes visuais que formam as telas da aplicação são inseridos através de tags xhtml.

5.3.1 Tela de entrada

Na nossa aplicação de gerar apostas, devemos definir uma tela para os usuários passarem os parâmetros necessários para que as apostas sejam geradas.



Gerador de Apostas

localhost:8080/K19-GeradorDeApostas/formulario.xhtml

Quantidade total de números:

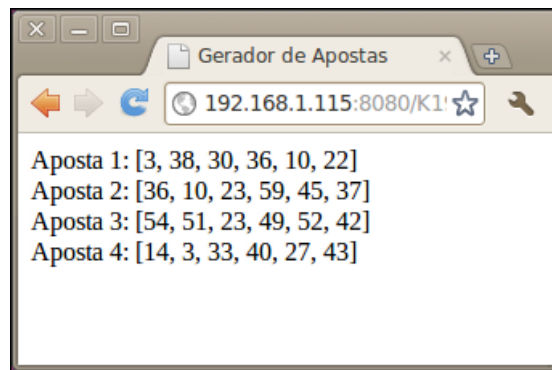
Quantidade de números por aposta:

Quantidade de apostas:

```
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>Gerador de Apostas</title>
9 </h:head>
10 <h:body>
11   <h:form>
12     <h:panelGrid>
13       <h:outputLabel value="Quantidade total de números:"/>
14       <h:inputText value="#{geradorDeApostasBean.quantidadeDeNumeros}"/>
15
16       <h:outputLabel value="Quantidade de números por aposta:"/>
17       <h:inputText value="#{geradorDeApostasBean.tamanhoDaAposta}"/>
18
19       <h:outputLabel value="Quantidade de apostas:"/>
20       <h:inputText value="#{geradorDeApostasBean.quantidadeDeApostas}"/>
21
22       <h:commandButton action="#{geradorDeApostasBean.geraApostas}"
23         value="Gerar"/>
24     </h:panelGrid>
25   </h:form>
26 </h:body>
27 </html>
```

5.3.2 Tela de Saída

Na tela de saída, devemos apresentar aos usuários uma listagem das apostas que foram criadas pelo GERADORDEAPOSTASBEAN.



```
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>Gerador de Apostas</title>
10 </h:head>
11 <h:body>
12   <ui:repeat var="aposta" value="#{geradorDeApostasBean.apostas}"
13     varStatus="status">
14     <h:outputText value="Aposta #{status.index + 1}: "/>
15     <h:outputText value="#{aposta}"/>
16     <br/>
17   </ui:repeat>
18 </h:body>
19 </html>
```

5.4 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **K19-Loteria**. Siga as imagens abaixo para configurar corretamente o projeto.

Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

Project location
☒ Use default location
Location:

Target runtime

Dynamic web module version

Configuration


Configures a Dynamic Web application to use JSF v2.0

EAR membership
☐ Add project to an EAR
EAR project name:

Working sets
☐ Add project to working sets

Java
Configure project for building a Java application.

Source folders on build path:

 src

Default output folder:

Web Module
Configure web module settings.

Context root:

Content directory:

☒ Generate web.xml deployment descriptor

? < Back Next > Cancel Finish

JSF Capabilities

⚠ Library configuration is disabled. Further classpath changes may be required later.

JSF Implementation Library

Type:

This facet requires JSF implementation library to be present on project classpath. By disabling library configuration, user takes on responsibility of configuring classpath appropriately via alternate means.

JSF Configuration File:

JSF Servlet Name:

JSF Servlet Class Name:

URL Mapping Patterns: Add... Remove

? < Back Next > Cancel Finish

2. Na pasta **src** do projeto **K19-Loteria** crie um pacote chamado **managedbeans**.
3. No pacote **managedbeans** crie uma classe chamada **GeradorDeApostasBean** com o seguinte conteúdo.

```
1  @ManagedBean
2  public class GeradorDeApostasBean {
3      private int quantidadeDeNumeros;
4
5      private int tamanhoDaAposta;
6
7      private int quantidadeDeApostas;
8
9      private List<List<Integer>> apostas;
10
11     public String geraApostas() {
12         // Prepara uma lista com todos os números
13         ArrayList<Integer> numeros = new ArrayList<Integer>();
14         for (int j = 1; j <= this.quantidadeDeNumeros; j++) {
15             numeros.add(j);
16         }
17
18         // Cria uma sublista da lista de números
19         List<Integer> subList = numeros.subList(0, this.tamanhoDaAposta);
20
21         // Lista de apostas vazia
22         this.apostas = new ArrayList<List<Integer>>();
23
24         // Gera as apostas
25         for (int i = 0; i < this.quantidadeDeApostas; i++) {
26             Collections.shuffle(numeros);
27             List<Integer> aposta = new ArrayList<Integer>(subList);
28             this.apostas.add(aposta);
29         }
30         return "lista-de-apostas";
31     }
32
33     // GETTERS AND SETTERS
34 }
```

4. Na pasta **WebContent**, crie um arquivo chamado **formulario.xhtml** e monte a tela de entrada do gerador de apostas.

```

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>Gerador de Apostas</title>
9 </h:head>
10 <h:body>
11   <h:form>
12     <h:panelGrid>
13       <h:outputLabel value="Quantidade total de números:"/>
14       <h:inputText value="#{geradorDeApostasBean.quantidadeDeNumeros}"/>
15
16       <h:outputLabel value="Quantidade de números por aposta:"/>
17       <h:inputText value="#{geradorDeApostasBean.tamanhoDaAposta}"/>
18
19       <h:outputLabel value="Quantidade de apostas:"/>
20       <h:inputText value="#{geradorDeApostasBean.quantidadeDeApostas}"/>
21
22       <h:commandButton action="#{geradorDeApostasBean.geraApostas}"
23         value="Gerar"/>
24     </h:panelGrid>
25   </h:form>
26 </h:body>
27 </html>

```

5. Na pasta **WebContent**, crie um arquivo chamado **lista-de-apostas.xhtml** para apresentar as apostas geradas.

```

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>Gerador de Apostas</title>
10 </h:head>
11 <h:body>
12   <ui:repeat var="aposta" value="#{geradorDeApostasBean.apostas}"
13     varStatus="status">
14     <h:outputText value="Aposta #{status.index + 1}: "/>
15     <h:outputText value="#{aposta}"/>
16     <br/>
17   </ui:repeat>
18 </h:body>
19 </html>

```

6. Preencha o formulário e clique no botão de gerar apostas para verifique o funcionamento da nossa aplicação. Acesse a url:

<http://localhost:8080/K19-Loteria/formulario.xhtml>

Capítulo 6

Componentes Visuais

A arquitetura de componentes visuais do JSF permite que novos componentes além dos que estão definidos na especificação sejam desenvolvidos por terceiros. Isso permitiu que bibliotecas de componentes extremamente “ricos” visualmente fossem desenvolvidas. Inclusive, em geral, essas bibliotecas utilizam recursos como o AJAX para melhorar a interatividade com o usuário. A mais famosa dessas bibliotecas é a **RichFaces** cujo site oficial pode ser visitado através da url:

(<http://www.jboss.org/richfaces>).

Porém, não devemos deixar nossos olhos nos enganarem. Os componentes visuais do JSF vão além da questão visual. Esses componentes podem ser reaproveitados em diversos pontos da mesma aplicação ou em aplicações diferentes mostrando de forma concreta o poder da Orientação a Objetos e o desenvolvimento Orientado a Componentes.

Além disso, a arquitetura do JSF permite que outros tipos de componentes sejam conectados aos componentes visuais. Por exemplo, podemos conectar componentes validadores aos componentes visuais para que os dados vindos dos usuários sejam verificados de acordo com alguma regra.

Neste capítulo apresentaremos os principais componentes visuais definidos pela especificação do JSF.

6.1 Formulários

Os formulários são necessários em todas as telas que precisam receber dados dos usuários. O componente visual **h:form** cria formulários.

Nome:

Sobre:

Sexo: ☒ Masculino ☐ Feminino

País:

Escolha uma senha:

Li e estou de acordo com os termos de uso. ☒

```

1 <h:form>
2   <h:outputLabel value="Nome: " for="input-nome"/>
3   <h:inputText id="input-nome"/>
4
5   <br/>
6
7   <h:outputLabel value="Sobre: " for="input-sobre"/>
8   <h:inputTextarea id="input-sobre"/>
9
10  <br/>
11
12  <h:outputLabel value="Sexo: "/>
13  <h:selectOneRadio>
14    <f:selectItem itemLabel="Masculino" itemValue="M"/>
15    <f:selectItem itemLabel="Feminino" itemValue="F"/>
16  </h:selectOneRadio>
17
18  <h:outputLabel value="País: "/>
19  <h:selectOneMenu>
20    <f:selectItem itemLabel="Argentina" itemValue="ar"/>
21    <f:selectItem itemLabel="Brasil" itemValue="br"/>
22    <f:selectItem itemLabel="Espanha" itemValue="es"/>
23  </h:selectOneMenu>
24
25  <br/>
26
27  <h:outputLabel value="Escolha uma senha: " for="input-senha"/>
28  <h:inputSecret id="input-senha"/>
29
30  <br/>
31
32  <h:outputLabel value="Li e estou de acordo com os termos de uso."
33    for="checkbox-termo"/>
34  <h:selectBooleanCheckbox id="checkbox-termo" value="aceito"/>
35
36  <br/>
37
38  <h:commandButton value="Cadastrar"/>
39 </h:form>

```

Na linha 2, utilizamos o componente visual **h:outputLabel** para criar um rótulo para o campo no qual deve ser digitado o nome da pessoa que se cadastra. Observe que o atributo **for** conecta o rótulo ao **id** do campo do nome. Dessa forma, se o usuário clicar no rótulo o cursor de digitação aparecerá nesse campo.

Na linha 3, utilizamos o componente visual **h:inputText** para criar o campo do nome. Definimos o atributo **id** para que o campo pudesse ser conectado ao rótulo logo acima.

Na linha 9, para criar um campo para texto maiores, utilizamos o componente **h:inputTextarea**. O funcionamento dele é muito semelhante ao do **h:inputText**. A diferença básica é que a área de digitação do **h:inputTextarea** é maior.

Na linha 13, aplicamos o componente **h:selectOneRadio** para criar um radio button que permite o usuário escolher o sexo (masculino e feminino). As opções do radio button são definidas pelo componente **f:selectItem** utilizado nas linhas 14 e 15.

Na linha 19, criamos um combo box para que o usuário escolha qual é o país de origem da pessoa que será cadastrada. Assim como no radio button, as opções do combo box são definidas com o componente **f:selectItem**.

Na linha 28, utilizamos o componente **h:inputSecret** para criar um campo de texto que não mostra na tela o valor que já foi digitado. Esse componente é útil para senhas.

Na linha 34, inserimos na tela um check box para saber se o usuário aceita ou não os termos de cadastro. O componente utilizado para isso foi o **h:selectBooleanCheckbox**.

Por fim, na linha 38, acrescentamos um botão para o usuário confirmar o cadastro.

6.2 Panel Grid

O componente **h:panelGrid** é utilizado para organizar outros componentes em tabelas de uma forma prática. Basicamente, para utilizar este componente, devemos definir quantas colunas queremos e ele automaticamente distribui os componentes em um número suficiente de linhas.

Nome:

Sobre:

Sexo: ☒ Masculino ☐ Feminino

País:

Escolha uma senha:

Li e concordo. ☒

```

1 <h:form>
2   <h:panelGrid columns="2">
3     <h:outputLabel value="Nome: " for="input-nome"/>
4     <h:inputText id="input-nome"/>
5
6     <h:outputLabel value="Sobre: " for="input-sobre"/>
7     <h:inputTextarea id="input-sobre"/>
8
9     <h:outputLabel value="Sexo: "/>
10    <h:selectOneRadio>
11      <f:selectItem itemLabel="Masculino" itemValue="M"/>
12      <f:selectItem itemLabel="Feminino" itemValue="F"/>
13    </h:selectOneRadio>
14
15    <h:outputLabel value="País: "/>
16    <h:selectOneMenu>
17      <f:selectItem itemLabel="Argertina" itemValue="ar"/>
18      <f:selectItem itemLabel="Brasil" itemValue="br"/>
19      <f:selectItem itemLabel="Espanha" itemValue="es"/>
20    </h:selectOneMenu>
21
22    <h:outputLabel value="Escolha uma senha: " for="input-senha"/>
23    <h:inputSecret id="input-senha"/>
24
25    <h:outputLabel value="Li e estou de acordo com os termos de uso."
26      for="checkbox-termo"/>
27    <h:selectBooleanCheckbox id="checkbox-termo" value="aceito"/>
28
29    <h:commandButton value="Cadastrar"/>
30  </h:panelGrid>
31 </h:form>

```

6.3 Panel Group

Em certas situações não conseguimos colocar dois ou mais componentes em um determinado local. Por exemplo, em uma célula de um panel Grid. Nesses casos, devemos aplicar o componente **h:panelGroup**. A ideia é inserir dois ou mais componentes em um panel Group e depois inserir o panel Group no lugar que só aceita um componente.

Nome:	<input type="text" value="Rafael Cosentino"/>
Sobre:	<input type="text" value="Líder de Treinamentos da K19"/>
Sexo:	<input checked="" type="radio"/> Masculino <input type="radio"/> Feminino
País:	<input type="text" value="Brasil"/>
Escolha uma senha:	<input type="password"/>
Li e concordo. <input checked="" type="checkbox"/>	<input type="button" value="Cadastrar"/>

```

1 <h:form>
2   <h:panelGrid columns="2">
3     <h:outputLabel value="Nome: " for="input-nome"/>
4     <h:inputText id="input-nome"/>
5
6     <h:outputLabel value="Sobre: " for="input-sobre"/>
7     <h:inputTextarea id="input-sobre"/>
8
9     <h:outputLabel value="Sexo: "/>
10    <h:selectOneRadio>
11      <f:selectItem itemLabel="Masculino" itemValue="M"/>
12      <f:selectItem itemLabel="Feminino" itemValue="F"/>
13    </h:selectOneRadio>
14
15    <h:outputLabel value="País: "/>
16    <h:selectOneMenu>
17      <f:selectItem itemLabel="Argentina" itemValue="ar"/>
18      <f:selectItem itemLabel="Brasil" itemValue="br"/>
19      <f:selectItem itemLabel="Espanha" itemValue="es"/>
20    </h:selectOneMenu>
21
22    <h:outputLabel value="Escolha uma senha: " for="input-senha"/>
23    <h:inputSecret id="input-senha"/>
24
25    <h:panelGroup>
26      <h:outputLabel value="Li e estou de acordo com os termos de uso."
27        for="checkbox-termo"/>
28      <h:selectBooleanCheckbox id="checkbox-termo" value="aceito"/>
29    </h:panelGroup>
30
31    <h:commandButton value="Cadastrar"/>
32  </h:panelGrid>
33 </h:form>

```

6.4 Tabelas

Podemos criar tabelas utilizando o componente **h:dataTable** com dados de alguma coleção. Basicamente, a diferença dos Data Tables e dos Panel Grids é que os Data Tables iteram diretamente nos ítemes de coleções.

Apostas
[36, 2, 41, 22, 40, 14]
[51, 56, 28, 39, 9, 45]
[14, 57, 34, 37, 10, 33]

```

1 <h:dataTable value="#{geradorDeApostasBean.apostas}" var="aposta" border="1">
2   <h:column>
3     <f:facet name="header">
4       <h:outputText value="Apostas"/></h:outputText>
5     </f:facet>
6     <h:outputText value="#{aposta}"/></h:outputText>
7   </h:column>
8 </h:dataTable>

```

Na linha 2, o componente **h:column** é utilizado para adicionar uma coluna na tabela definida

com o **h:dataTable**.

Na linha 3, aplicamos o componente **f:facet** para adicionar um cabeçalho na coluna correspondente.

6.5 Namespaces

Para poder aplicar as tags que definem as telas das aplicações JSF, precisamos adicionar os **namespaces** correspondentes às bibliotecas de tags que desejamos utilizar.

```
1 <html 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     <!-- Conteúdo Aqui -->
7
8 </html>
```

Os dois principais namespaces são: **http://java.sun.com/jsf/html** e **http://java.sun.com/jsf/core**. O primeiro é o namespace da biblioteca de tags do JSF que geram conteúdo HTML especificamente. O segundo corresponde a biblioteca de tags do JSF que não está atrelada a um tipo de visualização, ou seja, são tags mais genéricas.

Outro namespace importante é o **http://java.sun.com/jsf/facelets** que contém tags que nos ajudam a reaproveitar o código das telas. Veremos em outro capítulo o funcionamento das tags desse namespace.

6.6 Esqueleto HTML

Um documento HTML possui um esqueleto constituído por algumas tags principais. Para refletir essa estrutura devemos inserir algumas tags nos documentos XHTML que definem as telas JSF.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:h="http://java.sun.com/jsf/html">
3
4 <h:head>
5
6 </h:head>
7 <h:body>
8
9 </h:body>
10 </html>
```

6.7 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **ComponentesVisuais** seguindo os passos vistos no exercício do capítulo 5.

2. Adicione um arquivo na pasta **WebContent** do projeto **ComponentesVisuais** para criar uma tela utilizando os principais componentes de formulários. Este arquivo deve se chamar **formulario.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
8 <h:head>
9   <title>Cadastro de Usuário</title>
10 </h:head>
11 <h:body>
12   <h:form>
13     <h:outputLabel value="Nome: " for="input-nome"/>
14     <h:inputText id="input-nome"/>
15
16     <h:outputLabel value="Sobre: " for="input-sobre"/>
17     <h:inputTextarea id="input-sobre"/>
18
19     <h:outputLabel value="Sexo: "/>
20     <h:selectOneRadio>
21       <f:selectItem itemLabel="Masculino" itemValue="M"/>
22       <f:selectItem itemLabel="Feminino" itemValue="F"/>
23     </h:selectOneRadio>
24
25     <h:outputLabel value="País: "/>
26     <h:selectOneMenu>
27       <f:selectItem itemLabel="Argentina" itemValue="ar"/>
28       <f:selectItem itemLabel="Brasil" itemValue="br"/>
29       <f:selectItem itemLabel="Espanha" itemValue="es"/>
30     </h:selectOneMenu>
31
32     <h:outputLabel value="Escolha uma senha: " for="input-senha"/>
33     <h:inputSecret id="input-senha"/>
34
35     <h:outputLabel value="Li e concordo." for="checkbox-termo"/>
36     <h:selectBooleanCheckbox id="checkbox-termo" value="aceito"/>
37
38     <h:commandButton value="Cadastrar"/>
39   </h:form>
40 </h:body>
41 </html>
```

Acesse a url adequada para visualizar o formulário.

3. Utilize o componente **h:panelGrid** para alinhar melhor os itens do formulário criado no exercício anterior.

```
1 <h:form>
2   <h:panelGrid columns="2">
3
4     <!-- CONTEÚDO DO FORMULÁRIO -->
5
6   </h:panelGrid>
7 </h:form>
```

Verifique o resultado através de um navegador.

4. Utilize o componente **h:panelGroup** para agrupar dois ou mais componentes em uma célula do Panel Grid criado no exercício anterior.
5. Na pasta **src** crie um pacote chamado **managedbeans**. Nesse pacote, adicione uma classe com o seguinte conteúdo para modelar um simples Managed Bean que gera palavras.

```
1 @ManagedBean
2 public class PalavrasBean {
3     private List<String> palavras = new ArrayList<String>();
4
5     public PalavrasBean() {
6         this.palavras.add("K19 Treinamentos");
7         this.palavras.add("Rafael Cosentino");
8         this.palavras.add("Alexandre Macedo");
9         this.palavras.add("Jonas Hirata");
10    }
11
12
13    public List<String> getPalavras() {
14        return this.palavras;
15    }
16
17    public void setPalavras(List<String> palavras) {
18        this.palavras = palavras;
19    }
20 }
```

6. Crie uma tela aplicando o componente **h:dataTable** para apresentar as palavras geradas pelo Managed Bean do 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
8 <h:head>
9     <title>Palavras apenas palavras</title>
10 </h:head>
11 <h:body>
12     <h:dataTable value="#{palavrasBean.palavras}" var="palavra" border="1">
13         <h:column>
14             <f:facet name="header">
15                 <h:outputText value="Palavras"></h:outputText>
16             </f:facet>
17             <h:outputText value="#{palavra}"></h:outputText>
18         </h:column>
19     </h:dataTable>
20 </h:body>
21 </html>
```

Veja a tabela acessando a url correspondente.

Capítulo 7

Facelets

Certamente, você já ouviu alguém falar da importância da reutilização no desenvolvimento de software. Nesse contexto, os objetivos do reaproveitamento de software são: diminuir o tempo e o custo para desenvolver e facilitar a manutenção também diminuindo gastos e tempo.

Levando a ideia do reaproveitamento adiante, algumas pessoas desenvolveram um projeto chamado **Facelets** que tem como principal objetivo facilitar todo o processo de desenvolvimento e manutenção das telas de uma aplicação JSF. O Facelets já faz parte do JSF 2 sendo a engine padrão dessa tecnologia para a definição das telas das aplicações web.

7.1 Templating

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 de uma aplicação JSF e defini-lo através de um esqueleto (template) que possua trechos dinâmicos que possam ser preenchidos posteriormente.

A criação de um template é simples, basta criar um arquivo **xhtml** adicionando todos os componentes visuais que são fixos e devem aparecer em um determinado conjunto de telas. Para os trechos dinâmicos, devemos aplicar o componente **ui:insert** criando um espaço que pode ser preenchido depois.


```

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="conteudo"> 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>

```

Na linha 16, o atributo **name** do componente **ui:insert** é utilizado para identificar o espaço que será preenchido depois.

Após escrever o template, devemos criar as telas que o utilizarão. Essas telas são definidas também através de arquivos **xhtml**. Para indicar que desejamos utilizar um template, devemos aplicar o componente **ui:composition**. Para preencher um espaço deixado no template, devemos inserir o componente **ui:define** no código.

```

1 <html 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 <ui:composition template="/template.xhtml">
6   <ui:define name="conteudo">
7     <h:outputText value="Conteúdo da página que utiliza o template.xhtml"/>
8   </ui:define>
9 </ui:composition>
10 </html>

```

7.2 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **Facelets** seguindo os passos vistos no exercício do capítulo 5.
2. Crie um template na pasta **WebContent** chamado **template.xhtml**. Copie o arquivo **k19-logo.png** da pasta **K19-Arquivos/imagens** que está na Área de Trabalho para pasta **WebContent** do projeto **Facelets**.

```
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="conteudo"> 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>
```

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

```
1 <html 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 <ui:composition template="/template.xhtml">
7   <ui:define name="conteudo">
8     <h:form>
9       <h:outputLabel value="Nome: " for="campo-nome"/>
10      <h:inputText id="campo-nome"/>
11      <h:commandButton value="Enviar"/>
12    </h:form>
13  </ui:define>
14 </ui:composition>
15 </html>
```

Verifique o resultado acessando a url:

<http://localhost:8080/Facelets/teste-template.xhtml>

7.3 Particionando as telas

Com o intuito de organizar melhor o código das telas ou definir “pedaços” de telas que possam ser reaproveitados, podemos dividir o conteúdo de uma tela ou de um template em vários arquivos através do componente **ui:include**.

O recurso de separar uma tela ou um template em vários arquivos se torna mais interessante e útil quando temos a possibilidade de passar dados do arquivo principal para os secundários. Essa passagem de dados é realizada através do componente **ui:param**.

Por exemplo, estamos desenvolvendo uma aplicação e desejamos colocar o nome do usuário e um link para fazer logoff no canto superior direito sempre que alguém estiver logado ou um link para a página de login caso contrário.

Veja o fragmento que teríamos que acrescentar nos arquivos principais:

```

1 <div id="header">
2   <c:if test="#{usuarioBean.logado}">
3     <ui:include src="usuario-logado.xhtml">
4       <ui:param name="usuario" value="#{usuarioBean.usuario}" />
5     </ui:include>
6   </c:if>
7
8   <c:if test="#{!usuarioBean.logado}">
9     <ui:include src="usuario-nao-logado.xhtml" />
10  </c:if>
11 </div>

```

Veja os fragmentos que teríamos que acrescentar nos arquivos secundários:

```

1 <h:outputText value="Olá #{usuario.nome}" />
2 <h:commandButton action="usuarioBean.logout" value="Log out" />

```

```

1 <h:outputLabel value="Usuário: " for="campo-usuario" />
2 <h:inputText id="campo-usuario" />
3 <h:outputLabel value="Senha: " for="campo-senha" />
4 <h:inputSecret id="campo-senha" />
5 <h:commandButton action="usuarioBean.login" value="Log in" />

```

7.4 Exercícios

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

```

1 public class Instrutor {
2     private String nome;
3     private String dataDeNascimento;
4
5     // GETTERS AND SETTERS
6 }

```

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 **Facelets** e adicione nele uma classe chamada **InstrutorBean** com seguinte código:

```

1 @ManagedBean
2 public class InstrutorBean {
3
4     private List<Instrutor> instrutores = new ArrayList<Instrutor>();
5
6     public InstrutorBean() {
7         Instrutor rafael = new Instrutor();
8         rafael.setNome("Rafael Cosentino");
9         rafael.setDataDeNascimento("30/10/1984");
10
11         Instrutor marcelo = new Instrutor();
12         marcelo.setNome("Marcelo Martins");
13         marcelo.setDataDeNascimento("02/04/1985");
14
15         this.instrutores.add(rafael);
16         this.instrutores.add(marcelo);
17     }
18
19     public List<Instrutor> getInstrutores() {
20         return instrutores;
21     }
22
23     public void setInstrutores(List<Instrutor> instrutores) {
24         this.instrutores = instrutores;
25     }
26 }

```

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 **Facelets** e se chamar **instrutor-info.xhtml**.

```

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

```

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

```
1 <html 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 <ui:composition template="/template.xhtml">
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>
17 </html>
```

Veja o resultado acessando a url:

<http://localhost:8080/Facelets/listagem-de-instrutores.xhtml>

Capítulo 8

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.

8.1 Navegação Estática Implícita

Na navegação estática implícita, quando o usuário clica em algum botão ou link, um sinal (outcome) fixo definido no próprio botão ou link é enviado para o JSF. Este sinal é uma string que será utilizada pelo tratador de navegação do JSF para definir a próxima tela que será apresentada ao usuário.

Nas navegações implícitas, os nomes dos arquivos que definem as telas de resposta são montados com as strings dos outcomes. Por exemplo, se o usuário clica em um botão ou link de uma tela definida por um arquivo chamado **pagina1.xhtml** que envia o outcome “**pagina2**” então ele será redirecionado para a tela definida pelo arquivo **pagina2.xhtml** dentro do mesmo diretório que está o arquivo **pagina1.xhtml**.

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 Página 1</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>
```

```

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 Página 2</title>
9 </h:head>
10 <h:body>
11   <h1>K19 Página 2</h1>
12   <h:form>
13     <h:commandLink action="paginal">
14       <h:outputText value="Página 1"/>
15     </h:commandLink>
16   </h:form>
17 </h:body>
18 </html>

```

8.2 Navegação Estática Explícita

Na navegação implícita, os outcomes são os nomes dos arquivos que definem as telas. Para ter a liberdade de definir os nomes dos arquivos independentemente dos outcomes, podemos utilizar a navegação explícita. Porém, nesse tipo de navegação, devemos acrescentar algumas linhas no arquivo de configurações do JSF, o **faces-config.xml**.

```

1 <navigation-rule>
2   <from-view-id>paginal.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>

```

O código acima define que quando a tela do arquivo **pagina1.xhtml** emitir o sinal(outcome) “proxima” a transição deve ser realizada para a tela do arquivo **pagina2.xhtml**. Na tela do arquivo **pagina1.xhtml**, basta acrescentar um botão ou link que emita o sinal “next”.

```

1 <h:commandButton value="Próxima tela" action="proxima"/>

```

8.3 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **Navegacao** seguindo os passos vistos no exercício do capítulo 5.
2. Na pasta **WebContent** do projeto **Navegacao**, crie um arquivo chamado **pagina1.xhtml** com o seguinte código:

```
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 Página 1</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>
```

3. Novamente, na pasta **WebContent** do projeto **Navegacao**, crie um arquivo chamado **pagina2.xhtml** com o seguinte código:

```
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 Página 2</title>
9 </h:head>
10 <h:body>
11     <h1>K19 Página 2</h1>
12     <h:form>
13         <h:commandLink action="paginal">
14             <h:outputText value="Página 1"/>
15         </h:commandLink>
16     </h:form>
17 </h:body>
18 </html>
```

4. Navegue através dos links e botões da url:

`http://localhost:8080/Navegacao/paginal.xhtml`

5. Configure uma navegação explícita no arquivo **faces-config.xml**.

```
1 <navigation-rule>
2     <from-view-id>paginal.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>
```

6. Adicione um botão na tela do arquivo **pagina1.xhtml** que emita o sinal “**proxima**” abaixo do outro botão.


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

7. Navegue através dos links e botões da url:

<http://localhost:8080/Navegacao/paginal.xhtml>

8.4 Navegação Dinâmica Implícita

Na maioria dos casos, não queremos fixar nas telas os outcomes que elas podem enviar para o JSF. Normalmente, a escolha dos outcomes são realizadas dentro dos Managed Beans.

Na navegação dinâmica, quando um usuário clica em um botão ou link, um Managed Bean é chamado para escolher um outcome e enviar para o JSF. Para isso, associamos os botões ou os links a métodos dos Managed Beans.

O arquivo **cara-ou-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 Cara ou Coroa</title>
9 </h:head>
10 <h:body>
11   <h1>K19 Cara ou Coroa</h1>
12   <h:form>
13     <h:commandButton value="Lançar Moeda" action="#{managedBean.proxima}"/>
14   </h:form>
15 </h:body>
16 </html>
```

Nos Managed Beans, temos que definir a lógica de escolha dos outcomes.

```
1 @javax.faces.bean.ManagedBean
2 public class ManagedBean {
3
4     public String proxima() {
5         if (Math.random() < 0.5) {
6             return "cara";
7         } else {
8             return "coroa";
9         }
10    }
11 }
```

Os Managed Beans devem devolver uma string com o outcome escolhido. Se o outcome devolvido pelo Managed Bean não estiver configurado no **faces-config.xml** o tratador de navegação do JSF assumirá a navegação implícita, ou seja, o valor devolvido é o nome do arquivo que será processado para gerar a tela de resposta.

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 Cara ou Coroa</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>
```

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 Cara ou Coroa</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>
```

8.5 Navegação Dinâmica Explícita

Para implementar a navegação dinâmica explícita, basta seguir os passos da navegação dinâmica implícita e acrescentar as regras de navegação no arquivo de configurações do JSF.

8.6 Exercícios

8. Implemente um Managed Bean de forma aleatória escolha entre dois outcomes. Crie um pacote chamado **managedbeans** no projeto **Navegacao** e adicione uma classe chamada **ManagedBean**

```
1 @javax.faces.bean.ManagedBean
2 public class ManagedBean {
3
4     public String proxima(){
5         if(Math.random() < 0.5){
6             return "cara";
7         } else {
8             return "coroa";
9         }
10    }
11 }
```

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 **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 Cara ou Coroa</title>
9 </h:head>
10 <h:body>
11     <h1>K19 Cara ou Coroa</h1>
12     <h:form>
13         <h:commandButton value="Lançar Moeda" action="#{managedBean.proxima}"/>
14     </h:form>
15 </h:body>
16 </html>
```

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 Cara ou Coroa</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>
```

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 Cara ou Coroa</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>
```

11. Navegue através dos links e botões da url:

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



Capítulo 9

Managed Beans

Através das telas de uma aplicação, os usuários podem basicamente: enviar dados para o sistema; visualizar informações da aplicação; e disparar ações ou eventos. Só que as telas não realizam todas essas tarefas sozinhas. Nas aplicações web JSF, há objetos que oferecem todo o suporte que as telas necessitam. Esses objetos são chamados de Managed Beans.

As responsabilidades principais dos Managed Beans são:

1. Fornecer dados que serão apresentados aos usuários nas telas
2. Receber os dados enviados pelos usuários através das telas
3. Processar o tratamentos das ações e eventos disparados pelos usuários.

9.1 Criando Managed Beans

Para criar um Managed Bean, devemos escrever uma classe Java simples sem nenhum vínculo direto com classes ou interfaces do JSF. Dizemos, que as classes que implementam os Managed Beans são POJOs (Plain Old Java Object). A simplicidade é a maior vantagem de utilizar POJOs.

As classes dos Managed Beans precisam ser registradas no JSF. Na versão 2 do JSF, o registro é feito através da anotação **@ManagedBean**.

```
1 package managedbeans;
2
3 @ManagedBean
4 class MeuManagedBean {
5
6 }
```

Antes do JSF 2, o registro dos Managed Beans era realizado somente através do arquivo de configurações do JSF, o **faces-config.xml**.

```
1 <managed-bean>
2   <managed-bean-name>meuManagedBean</managed-bean-name>
3   <managed-bean-class>managedbeans.MeuManagedBean</managed-bean-class>
4   <managed-bean-scope>request</managed-bean-scope>
5 </managed-bean>
```

As duas possibilidades, anotação ou xml, estão disponíveis no JSF 2.

9.2 Disponibilizando dados para as telas

Basta criar métodos **getters** nas classes dos Managed Beans para disponibilizar dados para as telas.

```
1 package managedbeans;
2
3 @ManagedBean
4 class MeuManagedBean {
5     private String informacao;
6
7     public String getInformacao() {
8         return this.informacao;
9     }
10 }
```

9.3 Recebendo dados das telas

Basta criar métodos **setters** nas classes dos Managed Beans para receber dados das telas.

```
1 package managedbeans;
2
3 @ManagedBean
4 class MeuManagedBean {
5     private String informacao;
6
7     public void setInformacao(String informacao) {
8         this.informacao = informacao;
9     }
10 }
```

9.4 Definindo o tratamento das 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.

```
1 @ManagedBean
2 class MeuManagedBean {
3     public void logica() {
4         // implementação
5     }
6 }
```

Esses métodos podem ser VOID quando desejamos manter os usuários na mesma tela ou devolver STRING quando desejamos realizar uma navegação entre telas.

9.5 Expression Language

De alguma forma as telas precisam referenciar os Managed Beans com os quais elas desejam interagir. Há uma linguagem no JSF que podemos utilizar no código das telas que é apropriada para realizar a interação entre as páginas e os Managed Beans. Essa linguagem é chamada de **Expression Language**. Dentro do código de uma tela, delimitamos os trechos escritos em Expression Language através dos símbolos `#{ }`.

9.5.1 Nome dos Managed Beans

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 minúscula. Porém podemos alterar o nome acrescentando um argumento na anotação.

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

9.5.2 Acessando as propriedades dos Managed Beans

As propriedades dos Managed Beans são acessadas tanto para leitura quanto para escrita da maneira mais natural possível, pelo nome. Suponha o seguinte Managed Bean:

```
1 @ManagedBean(name="teste")
2 class ManagedBean {
3     private String informacao;
4
5     public String getInformacao() {
6         return this.informacao;
7     }
8
9     public void setInformacao(String informacao) {
10        this.informacao = informacao;
11    }
12 }
```

A propriedade `INFORMACAO` deve ser acessada da seguinte forma utilizando Expression Language:

```
1 #{teste.informacao}
```

9.6 Binding

Os componentes que estão nas telas podem ser “ligados” aos Managed Beans. Normalmente, essa ligação é estabelecida através de algum atributo das tags dos componentes. Por

exemplo, suponha que queremos ligar um campo de texto a uma propriedade de um Managed Bean, o código seria mais ou menos assim:

```
1 <h:inputText value="#{teste.informacao}"/>
```

O atributo VALUE do H:INPUTTEXT cria o vínculo entre o input e a propriedade INFORMACAO do Managed Bean TESTE. Dessa forma, quando o usuário preencher algum valor nesse campo, esse dado será armazenado no atributo INFORMACAO através do método SETINFORMACAO().

Outro exemplo, suponha que desejamos associar um método do nosso Managed Bean a um botão de uma tela qualquer. O código seria mais ou menos assim:

```
1 @ManagedBean
2 class MeuManagedBean {
3     public void logica() {
4         // implementação
5     }
6 }
```

```
1 <h:commandButton action="#{meuManagedBean.logica}" value="Executar"/>
```

9.7 Escopo

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.

9.7.1 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. Ou seja, os dados não são mantidos de uma requisição para outra.

O JSF utiliza o escopo Request como padrão. Dessa forma, se o desenvolvedor não definir nenhum escopo para um determinado Managed Bean o escopo Request será adotado automaticamente.

Mesmo sendo o padrão, podemos deixar explícito a escolha do escopo Request através da anotação **@RequestScoped** ou da tag **managed-bean-scope**.

```
1 package managedbeans;
2
3 @ManagedBean
4 @RequestScoped
5 class MeuManagedBean {
6
7 }
```

```
1 <managed-bean>
2   <managed-bean-name>meuManagedBean</managed-bean-name>
3   <managed-bean-class>managedbeans.MeuManagedBean</managed-bean-class>
4   <managed-bean-scope>request</managed-bean-scope>
5 </managed-bean>
```

Antes do JSF 2, havia somente a opção da configuração através de xml.

9.7.2 Session

Certas informações devem ser mantidas entre as requisições de um determinado usuário em um determinado navegador. Por exemplo, suponha uma aplicação que utiliza a ideia de 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 este usuário.

Daí surge o escopo Session. Cada usuário possui um espaço na memória do servidor que é chamado de **Session**, ou seja, existe uma Session para cada usuário. Tecnicamente, é possível existir duas ou mais Sessions de um mesmo usuário, por exemplo, se ele estiver utilizando dois 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, o usuário fez logout) ou o Web Container decide apagar a Session de um usuário pois este não faz requisições a “muito” 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 managedbeans;
2
3 @ManagedBean
4 @SessionScoped
5 class MeuManagedBean {
6
7 }
```

```
1 <managed-bean>
2   <managed-bean-name>meuManagedBean</managed-bean-name>
3   <managed-bean-class>managedbeans.MeuManagedBean</managed-bean-class>
4   <managed-bean-scope>session</managed-bean-scope>
5 </managed-bean>
```

Antes do JSF 2, havia somente a opção da configuração através de xml.

Temos que tomar um cuidado maior ao utilizar o escopo Session pois podemos acabar sobrecarregando 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.

9.7.3 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.

Os dados dessas instâncias podem ser utilizados nas telas de todos os usuários durante toda a execução da aplicação.

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

```
1 package managedbeans;
2
3 @ManagedBean
4 @ApplicationScoped
5 class MeuManagedBean {
6
7 }
```

```
1 <managed-bean>
2   <managed-bean-name>meuManagedBean</managed-bean-name>
3   <managed-bean-class>managedbeans.MeuManagedBean</managed-bean-class>
4   <managed-bean-scope>application</managed-bean-scope>
5 </managed-bean>
```

Antes do JSF 2, havia somente a opção da configuração através de xml.

9.7.4 View

O escopo View foi adicionado no JSF 2. A ideia é manter determinados dados enquanto o usuário não mudar de tela. As instância 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 managedbeans;
2
3 @ManagedBean
4 @ViewScoped
5 class MeuManagedBean {
6
7 }
```

```
1 <managed-bean>
2   <managed-bean-name>meuManagedBean</managed-bean-name>
3   <managed-bean-class>managedbeans.MeuManagedBean</managed-bean-class>
4   <managed-bean-scope>view</managed-bean-scope>
5 </managed-bean>
```

Antes do JSF 2, havia somente a opção da configuração através de xml.

9.8 Interdependência e Injeção

Instâncias de Managed Beans podem “conversar” entre si para dividir o processamento das requisições dos usuários de acordo a especialidade de cada uma delas. Para que duas instâncias “conversem”, uma deve possuir a referência da outra.

Como a criação e eliminação das instâncias dos Managed Beans são responsabilidade do JSF, ele é o mais indicado para administrar as referências entre as instâncias do Managed Beans.

Basicamente, o que o desenvolvedor deve fazer é indicar ao JSF quais instâncias devem ser conectadas através de referências.

Como exemplo, suponha dois Managed Beans:

```
1 @ManagedBean
2 class PrimeiroManagedBean {
3
4 }
```

```
1 @ManagedBean
2 class SegundoManagedBean {
3
4 }
```

Suponha também que o primeiro precisa chamar o segundo para resolver algum problema. Do ponto de vista da Orientação a Objetos, bastaria declarar um atributo na classe do primeiro Managed Bean relacionando-o ao segundo.

```
1 @ManagedBean
2 class PrimeiroManagedBean {
3   private SegundoManagedBean segundoManagedBean;
4 }
```

Porém, como é o JSF que vai administrar as ligações entre os objetos, devemos indicar através de anotações ou de xml o vínculo dos dois Managed Beans.

```
1 @ManagedBean
2 class PrimeiroManagedBean {
3   @ManagedProperty(value="#{segundoManagedBean}")
4   private SegundoManagedBean segundoManagedBean;
5 }
```

```

1 <managed-bean>
2   <managed-bean-name>primeiroManagedBean</managed-bean-name>
3   <managed-bean-class>managedbeans.PrimeiroManagedBean</managed-bean-class>
4   <managed-bean-scope>request</managed-bean-scope>
5   <managed-property>
6     <property-name>segundoManagedBean</property-name>
7     <value>#{segundoManagedBean}</value>
8   </managed-property>
9 </managed-bean>
10
11 <managed-bean>
12   <managed-bean-name>segundoManagedBean</managed-bean-name>
13   <managed-bean-class>managedbeans.SegundoManagedBean</managed-bean-class>
14   <managed-bean-scope>request</managed-bean-scope>
15 </managed-bean>

```

Quando as instâncias dos Managed Beans são criadas pelo JSF ele resolve todas as dependências conectando os objetos.

9.9 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **ManagedBean** seguindo os passos vistos no exercício do capítulo 5.
2. Na pasta **src**, faça um pacote chamado **managedbeans**.
3. No pacote **managedbeans**, adicione a seguinte classe:

```

1 package managedbeans;
2
3 @ManagedBean(name="impostometro")
4 public class ImpostometroBean {
5
6     private double total;
7
8     private double valor;
9
10    public void adicionaImposto() {
11        this.total += this.valor;
12    }
13
14    // GETTERS AND SETTERS
15 }

```

4. Crie uma tela com o suporte do Managed Bean **impostometro**. O arquivo da tela deve se chamar **impostometro.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>Impostometro</title>
9 </h:head>
10 <h:body>
11   <h1>Impostometro</h1>
12   <h:form>
13     <h:panelGrid>
14       <h:outputText value="Total: #{impostometro.total}" />
15       <h:outputLabel value="Digite o imposto a ser adicionado: "
16         for="campo-imposto" />
17       <h:inputText id="campo-imposto" value="#{impostometro.valor}" />
18       <h:commandButton value="Adicionar"
19         action="#{impostometro.adicionaImposto}" />
20     </h:panelGrid>
21   </h:form>
22 </h:body>
23 </html>
```

Adicione alguns valores através do formulário. Observe que o total não acumula os valores adicionados em requisições anteriores. Por quê?

5. Altere o escopo do **impostometro** para **Session**. Teste novamente.

```
1 @ManagedBean(name="impostometro")
2 @SessionScoped
3 public class ImpostometroBean {
```

6. Crie um Managed Bean para guardar os valores de algumas taxas.

```
1 package managedbeans;
2
3 @ManagedBean(name="taxas")
4 public class TaxasBean {
5
6   private double selic = 3.5;
7
8   // GETTERS AND SETTERS
9 }
```

7. Faça um Managed Bean que calcula juros baseado na taxa selic. Para isso, ele deve ser relacionado com o Managed Bean do exercício anterior.

```
1 package managedbeans;
2
3 @ManagedBean(name="calculadora")
4 public class CalculadoraBean {
5
6     @ManagedProperty(value="#{taxas}")
7     private TaxasBean taxas;
8
9     private double montante;
10
11     private double juros;
12
13     public void calculaJuros() {
14         this.juros = this.montante * this.taxas.getSelic() / 100;
15     }
16
17     // GETTERS AND SETTERS
18 }
```

8. Construa uma tela que utilize os Managed Beans criados nos exercícios anteriores.

```
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>Calculadora de Imposto</title>
9 </h:head>
10 <h:body>
11     <h1>Calculadora de Imposto</h1>
12     <h:form>
13         <h:panelGrid>
14             <h:outputText value="Selic: #{taxas.selic}" />
15             <h:outputText value="Juros: #{calculadora.juros}" />
16             <h:outputLabel value="Digite um montante: " />
17             <h:inputText value="#{calculadora.montante}" />
18             <h:commandButton value="Calcular"
19                 action="#{calculadora.calculaJuros}" />
20         </h:panelGrid>
21     </h:form>
22 </h:body>
23 </html>
```

Capítulo 10

Conversão e Validação

10.1 Conversão

Quando um usuário preenche um formulário, os valores escolhidos 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, suponha 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 chegam para as aplicações 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 para os usuários. Novamente, os dados devem ser convertidos antes de enviados para os navegadores.

Felizmente, o JSF oferece um mecanismo automatizado de conversão de dados. Veremos a seguir o funcionamento desse mecanismo.

10.1.1 Conversão Padrão Implícita

Para os tipos fundamentais da linguagem Java, o JSF define conversores padrões e os aplica de maneira implícita, ou seja, não precisamos fazer nada para o processo de conversão acontecer. Os tipos fundamentais do Java são:

- BigDecimal
- BigInteger
- Boolean
- Byte
- Character

- Double
- Float
- Integer
- Long
- Short

```
1 @ManagedBean
2 public class MeuManagedBean {
3
4     private double numero;
5
6     // GETTERS AND SETTERS
7 }
```

```
1 <!-- O valor digitado nesse campo será convertido para double -->
2 <h:inputText value="#{meuManagedBean.numero}" />
```

10.1.2 Conversão Padrão Explícita

Em alguns casos, os conversores padrões aplicados implicitamente nos tipos fundamentais do Java não são suficientes. Por exemplo, para trabalhar com valores monetários precisamos de outros conversores. Por isso, o JSF define dois outros conversores padrões para serem aplicados de maneira explícita.

f:convertNumber

A tag **f:convertNumber** permite que conversões mais sofisticadas sejam feitas em valores numéricos.

Estipulando duas casas decimais no mínimo:

```
1 <h:outputText value="#{managedbean.valor}" >
2     <f:convertNumber minFractionDigits="2" />
3 </h:outputText>
```

1.60

Defindo a formatação através de expressão regular:

```
1 <h:outputText value="#{managedbean.valor}" >
2     <f:convertNumber pattern="#0.000" />
3 </h:outputText>
```

1.600

Apresentando os dados em porcentagem:

```
1 <h:outputText value="#{managedbean.valor}" >
2   <f:convertNumber type="percent" />
3 </h:outputText>
```

160%

Utilizando o símbolo do Real para valores monetários:

```
1 <h:outputText value="#{managedbean.valor}" >
2   <f:convertNumber currencySymbol="R$" type="currency" />
3 </h:outputText>
```

R\$1.60

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**.

```
1 <h:outputText value="#{managedbean.date}" >
2   <f:convertDateTime pattern="dd/MM/yyyy" />
3 </h:outputText>
```

28/10/2010

10.2 Mensagens de Erro

Eventualmente, as informações preenchidas pelos usuários em formulários não são adequadas impedindo a conversão dos dados. Nesses casos, geralmente, desejamos apresentar para os usuários mensagens relacionadas aos erros no preenchimento das informações.

10.2.1 h:message

Para adicionar nas telas erros relacionados a um determinado campo, devemos utilizar a componente **h:message**. Primeiro, temos que definir um **id** para o campo desejado. Depois,

associar o **h:message** a esse **id**.

```
1 @ManagedBean
2 public class MeuManagedBean {
3
4     private double numero;
5
6     // GETTERS AND SETTERS
7 }
```

```
1 <h:inputText value="#{meuManagedBean.numero}" id="campo-numero"/>
2 <h:message for="campo-numero"/>
```

form:campo-numero: 'K19 Treinamentos' must be a number between 4.9E-324 and 1.7976931348623157E308 Example: 1999999

10.2.2 h:messages

A tag **h:message** permite que os erros dos diversos campos de um formulário sejam colocados um a um na tela. Inclusive, podemos colocar as mensagens de erro de campo em lugares diferentes na página que será apresentada ao usuário.

Em alguns casos, simplesmente, queremos colocar todos os erros de todos os campos de um formulário juntos na tela. Para isso, devemos aplicar a tag **h:messages**.

```
1 <h:messages/>
```

10.2.3 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>

Essas mensagens estão definidas em inglês. Normalmente, queremos personalizar essas mensagens. Para isso, devemos seguir dois passos:

1. Criar um arquivo de mensagens.
2. Registrar o arquivo de mensagens.

Criando um Arquivo de Mensagens

Um arquivo de mensagens é um conjunto de chaves e valores. Sendo que cada chave se refere a um tipo de erro e está associada a um valor que é o texto que será apresentado nas telas através das tags **h:message** ou **h:messages**.

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.BOOLEAN={1}: '{0}' must be 'true' or 'false'.
```

Os arquivos de mensagens devem possuir o sufixo **properties**.

Registrando um Arquivo de Mensagens

Suponha que você tenha criado um arquivo de mensagem chamado **Messages.properties** num pacote chamado **resources**. Para registrá-lo, você deve acrescentar uma configuração no arquivo **faces-config.xml**.

```
1 <application>
2   <message-bundle>resources.Messages</message-bundle>
3 </application>
```

10.3 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **ConversaoValidacao** seguindo os passos vistos no exercício do capítulo 5.
2. Acrescente um pacote na pasta **src** chamado **managedbean** e adicione a seguinte classe nesse pacote:

```

1 @ManagedBean
2 public class FuncionarioBean {
3
4     private double salario;
5
6     private int codigo;
7
8     private Date aniversario;
9
10    // GETTERS AND SETTERS
11 }

```

3. Crie uma tela para cadastrar funcionários. Adicione um arquivo na pasta **WebContent** chamado **cadastro.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 <h:head>
8     <title>Cadastro de Funcionário</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.salario}">
17                 <!-- Sistema numérico do Brasil -->
18                 <f:convertNumber locale="pt_BR"/>
19             </h:inputText>
20             <h:message for="campo-salario"/>
21
22             <h:outputLabel value="Código: " for="campo-codigo"/>
23             <h:inputText id="campo-codigo" value="#{funcionarioBean.codigo}"/>
24             <h:message for="campo-codigo"/>
25
26             <h:outputLabel value="Data: (dd/MM/yyyy)" for="campo-aniversario"/>
27             <h:inputText id="campo-aniversario"
28                 value="#{funcionarioBean.aniversario}"
29                 <f:convertDateTime pattern="dd/MM/yyyy"/>
30             </h:inputText>
31             <h:message for="campo-aniversario"/>
32
33             <h:commandButton value="Cadastrar"/>
34         </h:panelGrid>
35         <h:messages/>
36     </h:form>
37 </h:body>
38 </html>

```

Preencha o formulário várias vezes com valores errados e observe os erros.

4. Faça um pacote chamado **resources** na pasta **src** e adicione um arquivo de mensagens nesse pacote 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.
```

5. Adicione a configuração necessária no **faces-config.xml** para utilizar o arquivo de mensagens criado no exercício anterior.

```
1 <application>  
2   <message-bundle>resources.Messages</message-bundle>  
3 </application>
```

Observação: a tag **<application>** deve ser colocada dentro de **faces-config**.

6. Preencha o formulário várias vezes com valores errados e observe as novas mensagens.

10.4 Validação

Muitas vezes, apenas a conversão de dados não é suficiente para verificar se uma informação preenchida em um formulário por um usuário está correta. Por exemplo, suponha um campo para os usuários digitarem uma idade. Como visto anteriormente, o valor digitado será tratado como texto até chegar na aplicação e lá poderá ser convertido para INT. Essa conversão não verifica se o número obtido é negativo. Porém, nesse caso, não seria correto obter números negativos pois a idade de uma pessoa é sempre positiva. Dessa forma, depois da conversão dos dados, mais uma etapa deve ser realizada para validar as informações.

10.4.1 Validação Padrão

O JSF também define validadores padrões para serem aplicados nos dados obtidos dos usuários. A seguir veremos a aplicação desses validadores:

10.4.2 Campo Obrigatório (Required)

A validação mais comum de todas é a de verificar se um determinado campo não deixou de ser preenchido. Podemos aplicar essa validação utilizando o atributo **required** dos inputs.

```
1 <h:inputText value="#{managedbean.nome}" id="campo-nome" required="true"/>  
2 <h:message for="campo-nome"/>
```

10.4.3 f:validateLongRange

O validador **f:validateLongRange** é utilizado para verificar se um valor numérico inteiro pertence a um determinado intervalo de números.

```
1 <h:inputText value="#{managedbean.idade}" id="campo-idade">
2   <f:validateLongRange minimum="10" maximum="130" />
3 </h:inputText>
4 <h:message for="campo-idade"/>
```

10.4.4 f:validateDoubleRange

O validador **f:validateDoubleRange** é utilizado para verificar se um valor numérico real pertence a um determinado intervalo de números.

```
1 <h:inputText value="#{managedbean.preco}" id="campo-preco">
2   <f:validateDoubleRange minimum="20.57" maximum="200.56" />
3 </h:inputText>
4 <h:message for="campo-preco"/>
```

10.4.5 f:validateLength

O validador **f:validateLength** é utilizado para verificar se uma string possui uma quantidade mínima ou máxima de letras.

```
1 <h:inputText value="#{managedbean.nome}" id="campo-nome">
2   <f:validateLength minimum = "6" maximum = "20"/>
3 </h:inputText>
4 <h:message for="campo-nome"/>
```

10.4.6 f:validateRegex

O validador **f:validateRegex** é utilizado para verificar se um texto respeita uma determinada expressão regular.

```
1 <h:inputText value="#{managedbean.nome}" id="campo-nome">
2   <f:validateRegex pattern="[a-zA-Z]{6,20}" />
3 </h:inputText>
4 <h:message for="campo-nome"/>
```

10.4.7 Bean Validation

Uma nova abordagem para definir as validações foi adicionada no JSF2. A ideia é declarar as regras de validação nas classes de modelo ao invés de inserí-las nas telas. A grande vantagem das validações definidas nas classes de 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 de modelo.

```
1 public class Funcionario {  
2     @NotNull  
3     private String nome;  
4  
5     ...  
6 }
```

@NotNull: o valor não pode ser nulo.

@Min: define um valor mínimo.

@Max: define um valor máximo.

@Size: define um valor mínimo e máximo.

@Pattern: aplica uma expressão regular na validação.

Podemos acrescentar mensagens de erro através das anotações.

```
1 public class Funcionario {  
2     @NotNull(message="O nome não pode ser nulo")  
3     private String nome;  
4     ...  
5 }
```

10.5 Exercícios

7. No projeto **ConversaoValidacao**, crie uma tela para cadastrar produtos de uma loja virtual. O arquivo que você criará deve se chamar **cadastra-produto.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
8 <h:head>
9   <title>Cadastro de Produto</title>
10 </h:head>
11
12 <h:body>
13   <h:form>
14     <h:panelGrid columns="3">
15       <h:outputLabel value="Nome: " for="produto-nome"/>
16       <h:inputText id="produto-nome"
17         required="true"
18         value="#{produtoBean.produto.nome}"/>
19       <h:message for="produto-nome" />
20
21       <h:outputLabel value="Preço: " for="produto-preco"/>
22       <h:inputText id="produto-preco"
23         required="true"
24         value="#{produtoBean.produto.preco}">
25         <f:validateDoubleRange minimum="0" />
26       </h:inputText>
27       <h:message for="produto-preco" />
28
29       <h:commandButton value="Cadastrar" />
30     </h:panelGrid>
31   </h:form>
32 </h:body>
33 </html>

```

8. Crie a classe de modelo para definir os produtos. Adicione essa classe dentro de um pacote chamado **model**.

```

1 public class Produto {
2   private String nome;
3   private double preco;
4
5   // GETTERS AND SETTERS
6 }

```

9. Depois, implemente o Managed Bean que dará suporte à tela de cadastro de produtos. Faça a classe **ProdutoBean** dentro do pacote **managedbeans**.

```

1 @ManagedBean
2 public class ProdutoBean {
3   private Produto produto = new Produto();
4
5   // GETTERS AND SETTERS
6 }

```

10. Teste o formulário preenchendo diversos valores e observe as mensagens de erro.

11. Retire a validação realizada com a tag **f:validateDoubleRange**. Acrescente uma validação com as anotações da especificação de Bean Validation na classe PRODUTO.

```
1 public class Produto {  
2     private String nome;  
3  
4     @Min(value = 0)  
5     private double preco;  
6  
7     // GETTERS AND SETTERS  
8 }
```

12. Teste novamente o formulário.
13. (Opcional) Altere as mensagens de erros do formulário de cadastro de produto.



Capítulo 11

Eventos

Normalmente, as pessoas preferem utilizar aplicações que ofereçam maior grau de interatividade. Consequentemente, as empresas buscam sistemas mais interativos para controlar os seus negócios e serem utilizados pelos seus funcionários.

A interatividade de uma aplicação está diretamente relacionada a sua capacidade de percepção e reação. O nível mais alto de interatividade aconteceria se uma aplicação pudesse perceber e reagir aos pensamentos dos usuários.

O JSF não oferece esse nível de interação, na verdade, nenhuma tecnologia oferece. Mas, veremos que ele oferece um mecanismo bem sofisticado para aumentar a interatividade com os usuários. Esse mecanismo é baseado na ideia de eventos. No JSF, há duas categorias fundamentais de eventos: eventos de aplicação e eventos de ciclo de vida.

Os eventos de aplicação correspondem às ações dos usuários que são pertinentes às aplicações. Por exemplo, um usuário pressiona um botão ou altera o valor de preenchido em um campo de um formulário.

Os eventos de ciclo de vida correspondem às transições entre as diversas etapas do processamento de uma requisição ou às transições entre os estados dos componentes do JSF ou da própria aplicação.

11.1 Eventos de Aplicação (Application Events)

Como dito anteriormente, os eventos de aplicação correspondem às ações dos usuários que interessam para as aplicações. O JSF suporta dois tipos de eventos de aplicação: **ActionEvent** e **ValueChangeEvent**.

11.1.1 ActionEvent

Os ActionEvents são gerados por botões ou links quando esses são pressionados pelos usuários. O tratamento dos ActionEvents pode ser definido por métodos dentro dos Managed Beans. Esses métodos são classificados em dois tipos: **Action Method** ou **Action Listener Method**. Um Action Method deve ser utilizado quando desejamos efetuar uma navegação (mudar de tela) após o tratamento do evento. Caso contrário, devemos utilizar um Action Listener Method.

Action Method

Um Action Method deve devolver uma String que será utilizada como outcome para processar uma navegação. Veja um exemplo de Action Method.

```
1 <h:commandButton value="Salva" action="#{produtoBean.salva}" />
```

```
1 public String salva() {  
2     // implementação  
3     return "lista-produtos";  
4 }
```

No **h:commandButton** definimos qual é o Action Method que queremos associar ao botão através do atributo **action**.

Action Listener Method

Um Action Listener Method precisa ser **void** e aceita um **ActionEvent** como argumento. Um **ActionEvent** contém informações sobre o evento disparado.

```
1 <h:commandButton value="Salva" actionListener="#{produtoBean.salva}" />
```

```
1 public void salva(ActionEvent event) {  
2     // implementação  
3 }
```

No **h:commandButton** definimos qual é o Action Listener Method que queremos associar ao botão através do atributo **actionListener**.

11.1.2 ValueChangeEvent

Os **ValueChangeEvent** são gerados quando os usuários modificam o valor preenchido em um campo de um formulário. O tratamento desse tipo de evento pode ser realizado por métodos de um Managed Bean. Esses métodos são chamados de **Value Change Listeners**.

Um **Value Change Listener** precisa ser **void** e aceita um **ValueChangeEvent** como argumento. Um **ValueChangeEvent** contém informações sobre o evento disparado.

```
1 <h:outputLabel value="Preço: " />  
2 <h:inputText valueChangeListener="#{produtoBean.mudaPrecoListener}" />
```

```
1 public void mudaPrecoListener(ValueChangeEvent event) {  
2     // implementação  
3 }
```

No **h:inputText** definimos qual é o Value Change Listener queremos associar ao campo através do atributo **valueChangeListener**.

11.2 Eventos de Ciclo de Vida (Lifecycle Events)

Os eventos de ciclo de vida são utilizados quando desejamos executar procedimentos antes ou depois de uma determinada etapa do processamento de uma requisição ou a cada mudança de estado de um componente do JSF e da própria aplicação. Vamos discutir a respeito dos **Phase Events** que correspondem às transições entre as etapas do processamento das requisições.

Um Phase Event é tratado por um **Phase Listener**. Como o próprio JSF dispara automaticamente os Phase Events, devemos apenas criar um **Phase Listener** e registrá-lo.

Para criar um Phase Listener, devemos escrever uma classe que implemente a interface **PhaseListener**.

```
1 package listeners;
2
3 public class MeuPhaseListener implements PhaseListener {
4     public void beforePhase(PhaseEvent pe) {
5         // implementação
6     }
7     public void afterPhase(PhaseEvent pe) {
8         // implementação
9     }
10    public PhaseId getPhaseId() {
11        return PhaseId.ANY_PHASE;
12    }
13 }
```

O método **getPhaseID()** associa o listener a determinadas fases do processamento de uma requisição. Os métodos **beforePhase()** e **afterPhase()** são executados respectivamente antes e depois das fases associadas ao listener.

Para registrar o nosso Phase Listener devemos acrescentar algumas configurações no **faces-config.xml**.

```
1 <lifecycle>
2     <phase-listener>listeners.MeuPhaseListener</phase-listener>
3 </lifecycle>
```

11.3 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **Eventos** seguindo os passos vistos no exercício do capítulo 5.
2. Vamos montar um formulário de estados e cidades. Para isso, crie um pacote chamado **model** e adicione a seguinte classe para representar os estados.

```
1 package model;
2
3 public class Estado {
4     private String nome;
5
6     private String sigla;
7
8     private List<String> cidades = new ArrayList<String>();
9
10    // GETTERS AND SETTERS
11 }
```

3. Depois, crie um pacote **managedbeans** e adicione um Managed Bean para manipular os estados.

```
1 package managedbeans;
2
3 @ManagedBean
4 @SessionScoped
5 public class LocalidadeBean {
6
7     private String cidade;
8     private String siglaEstado;
9
10    private Estado estado = new Estado();
11    private List<Estado> estados = new ArrayList<Estado>();
12
13    public LocalidadeBean() {
14        Estado sp = new Estado();
15        sp.setSigla("SP");
16        sp.setNome("São Paulo");
17        sp.getCidades().add("São Paulo");
18        sp.getCidades().add("Campinas");
19
20        Estado rj = new Estado();
21        rj.setSigla("RJ");
22        rj.setNome("Rio de Janeiro");
23        rj.getCidades().add("Rio de Janeiro");
24        rj.getCidades().add("Niterói");
25
26        this.estados.add(sp);
27        this.estados.add(rj);
28    }
29
30    public void mudaEstado(ValueChangeEvent vce) {
31        this.siglaEstado = vce.getNewValue().toString();
32        for(Estado e : this.estados){
33            if(e.getSigla().equals(this.siglaEstado)){
34                this.estado = e;
35            }
36        }
37    }
38
39    // GETTERS AND SETTERS
40 }
```

4. Faça a tela de busca de estados e cidade. Adicione um arquivo chamado **busca-localidade.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>Busca Localidade</title>
9   </h:head>
10
11   <h:body>
12       <h1>Busca Localidade</h1>
13       <h:form>
14           <h:outputLabel value="Estado: " for="campo-estado" />
15           <h:selectOneMenu
16               id="campo-estado"
17               value="#{localidadeBean.siglaEstado}"
18               valueChangeListener="#{localidadeBean.mudaEstado}"
19               onchange="this.form.submit();">
20
21               <f:selectItems
22                   value="#{localidadeBean.estados}"
23                   var="e"
24                   itemLabel="#{e.nome}"
25                   itemValue="#{e.sigla}" />
26           </h:selectOneMenu>
27
28           <h:outputLabel value="Cidade: " for="campo-cidade" />
29           <h:selectOneMenu
30               id="campo-cidade"
31               value="#{localidadeBean.cidade}">
32
33               <f:selectItems
34                   value="#{localidadeBean.estado.cidades}" />
35           </h:selectOneMenu>
36       </h:form>
37   </h:body>
38 </html>

```




Capítulo 12

Ajax

Quando as aplicações possuem telas complexas com grande quantidade de conteúdo, não é interessante recarregar uma página inteira só para modificar um pequeno “pedaço” da tela pois isso deixará os usuários insatisfeitos.

Novamente, com o intuito de melhorar a interatividade entre as aplicações e os usuários, podemos aplicar o conceito do AJAX (Asynchronous Javascript And XML). Aplicando a ideia do AJAX obtemos duas capacidades muito uteis: a primeira é poder fazer requisições sem recarregar as páginas completamente e sim a parte delas que nos interessa; a segunda é poder realizar requisições sem pausar a navegação dos usuários.

Por exemplo, suponha uma página de listagem de fotos que possua paginação. Quando o usuário pressiona o link para a próxima página, não é necessário recarregar todo o conteúdo da tela, podemos recarregar apenas os itens na listagem.

Outro exemplo, suponha uma aplicação de Instant Message (gtalk, msn, ...). A listagem de contatos pode ser atualizada frequentemente sem os usuários pedirem e sem que eles tenham que para a navegação para essa atualização.

A versão 2 do JSF, diferentemente das anteriores, oferece suporte nativo a AJAX. Veremos como utilizar esse suporte.

12.1 Fazendo requisições AJAX

As requisições AJAX são realizadas quando determinados eventos definidos pela linguagem Javascript ocorrem. Esses eventos estão fortemente relacionados aos componentes visuais colocados nas telas. Precisamos indicar para o JSF quais componentes e eventos devem disparar requisições para o servidor. Para fazer isso, devemos utilizar a tag **f:ajax** (principal tag do suporte nativo do JSF para aplicar o conceito do AJAX).

```
1 <h:inputText>  
2   <f:ajax/>  
3 </h:inputText>
```

No exemplo acima, uma requisição AJAX será disparada quando o valor do campo for modificado. Isso porque a tag **f:ajax** assume o evento padrão do componente associado a ela. O componente **h:inputText** utiliza por padrão o evento **onchange**.

Por outra lado, podemos explicitar o evento que deve disparar as requisições AJAX deixando o código mais claro através do atributo **event**. Devemos tomar cuidado pois nem todos os eventos são aceitos por todos os componentes.

```
1 <h:inputText>
2   <f:ajax event="keyup"/>
3 </h:inputText>
```

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>
```

Novamente, se não escolhermos explicitamente o evento que vai disparar as requisições o JSF assumirá o padrão de cada componente. O padrão dos componentes **h:inputText** e **h:inputSecret** é **onchange**. O padrão do componente **h:commandButton** é **onclick**.

Mas, podemos explicitar o evento que deve disparar as requisições AJAX para um determinado grupo de componentes da mesma forma que fizemos anteriormente.

```
1 <f:ajax event="mouseout">
2   <h:inputText/>
3   <h:inputSecret/>
4   <h:commandButton value="OK"/>
5 </f:ajax>
```

12.2 Recarregando alguns “pedaços” das telas

Após realizar uma requisição AJAX, podemos pedir para o JSF redesenhar alguns “pedaços” da tela que está sendo mostrada para o usuário. Por exemplo, suponha uma listagem paginada de produtos, quando o usuário clica no botão que requisita através de AJAX a próxima página e a resposta chega, podemos mandar o JSF redesenhar a listagem e apenas a listagem com os dados que acabaram de chegar.

A tag **f:ajax** através do atributo **render** permite escolher os ids dos componentes que devem ser recarregados após uma requisição AJAX.

```
1 <h:commandButton value="Gera Número">
2   <f:ajax event="click" render="numero"/>
3 </h:commandButton>
4 <h:outputText id="numero" value="managedBean.numero"/>
```

Podemos redesenhar vários componentes, basta passar uma listagem de ids no valor do atributo **render**.

```
1 <h:commandButton value="Gera Números">
2   <f:ajax event="click" render="numero1 numero2"/>
3 </h:commandButton/>
4 <h:outputText id="numero1" value="managedBean.numero1"/>
5 <h:outputText id="numero2" value="managedBean.numero2"/>
```

12.3 Processando alguns “pedaços” das telas

Quando uma requisição AJAX é feita, podemos determinar quais componentes da tela devem ser avaliados pelo JSF. Por exemplo, quando enviamos um formulário, provavelmente, só é necessário avaliar os componentes que estão no próprio formulário.

Podemos definir quais componentes devem ser avaliados pelo JSF através do atributo **execute** passando uma lista de ids. Quando escolhemos um componente para ser avaliados os componentes dentro dele também serão.

```
1 <h:form id="formulario">
2   <h:inputText/>
3
4   <h:inputSecret/>
5
6   <h:commandButton value="Entrar">
7     <f:ajax event="click" render="message" execute="formulario"/>
8   </h:commandButton/>
9 </h:form>
10 <h:outputText id="message" value="#{loginBean.message}"/>
```

12.4 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 componente. Dessa forma, podemos trabalhar sem a necessidade de definir ids em alguns casos.

@all : refere-se a todos os componentes da tela.

@nome : 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 id do formulário.

```

1 <h:form>
2   <h:inputText/>
3
4   <h:inputSecret/>
5
6   <h:commandButton value="Entrar">
7     <f:ajax event="click" render="message" execute="@form"/>
8   </h:commandButton/>
9 </h:form>
10 <h:outputText id="message" value="#{loginBean.message}"/>

```

12.5 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **Ajax** seguindo os passos vistos no exercício do capítulo 5.
2. Vamos montar um formulário de cadastro de automóveis. Para isso, crie um pacote chamado **model** e adicione a seguinte classe para representar os automóveis.

```

1 package model;
2
3 public class Automovel {
4
5     private String nome;
6
7     private String marca;
8
9     // GETTERS AND SETTERS
10 }

```

3. Depois, crie um pacote **managedbeans** e adicione um Managed Bean para manipular os automóveis.

```

1 package managedbeans;
2
3 @ManagedBean
4 @SessionScoped
5 public class AutomovelBean {
6
7     private Automovel automovel = new Automovel();
8
9     private List<Automovel> automoveis = new ArrayList<Automovel>();
10
11     public void adiciona(){
12         this.automoveis.add(this.automovel);
13         this.automovel = new Automovel();
14     }
15
16     // GETTERS AND SETTERS
17 }

```

4. Faça a tela de listagem e cadastro de automóveis. Adicione um arquivo chamado **lista-cadastro-automovel.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>Automóveis</title>
9   </h:head>
10
11   <h:body>
12     <h1>Novo Automóvel</h1>
13     <h:form>
14       <h:panelGrid columns="2">
15         <h:outputLabel
16           value="Nome: "
17           for="campo-nome" />
18         <h:inputText
19           id="campo-nome"
20           value="#{automovelBean.automovel.nome}" />
21
22         <h:outputLabel
23           value="Marca: "
24           for="campo-marca" />
25         <h:inputText
26           id="campo-marca"
27           value="#{automovelBean.automovel.marca}" />
28
29         <h:commandButton
30           value="Cadastrar">
31           <f:ajax
32             event="click"
33             render="lista"
34             execute="@form"
35             listener="#{automovelBean.adiciona}" />
36         </h:commandButton>
37       </h:panelGrid>
38
39       <h1>Lista de Automóveis</h1>
40       <h:dataTable
41         id="lista"
42         value="#{automovelBean.automoveis}"
43         var="automovel">
44
45         <h:column>
46           <f:facet name="header">
47             <h:outputText value="Nome" />
48           </f:facet>
49           #{automovel.nome}
50         </h:column>
51
52         <h:column>
53           <f:facet name="header">
54             <h:outputText value="Marca" />
55           </f:facet>
56           #{automovel.marca}
57         </h:column>
58       </h:dataTable>
59     </h:form>
60   </h:body>
61 </html>
```



Capítulo 13

Projeto

Nos capítulos anteriores, vimos isoladamente recursos do JSF e do JPA. Agora, vamos mostrar em detalhes como esses recursos trabalham juntos e solidificar os conhecimentos obtidos. Além disso, mostraremos alguns padrões e conceitos relacionados ao desenvolvimento de aplicações web.

Como exemplo de aplicação desenvolveremos uma aplicação de cadastro de jogadores e seleções de futebol.

13.1 Modelo

Por onde começar o desenvolvimento de uma aplicação? Essa é uma questão recorrente. Um ótimo ponto de partida é desenvolver as entidades principais da aplicação. No nosso caso, vamos nos restringir às entidades **Selecao** e **Jogador**. Devemos estabelecer um relacionamento entre essas entidades já que um jogador atua em uma seleção.

13.2 Exercícios

1. Crie um projeto do tipo **Dynamic Web Project** chamado **K19-CopaDoMundo** seguindo os passos vistos no exercício do capítulo 5.
2. Faça um pacote chamado **modelo** e adicione as duas classes principais da nossa aplicação.

```
1 public class Selecao {  
2  
3     private String pais;  
4  
5     private String tecnico;  
6  
7     // GETTERS AND SETTERS  
8 }
```



```
1 public class Jogador {  
2  
3     private String nome;  
4  
5     private String posicao;  
6  
7     private Calendar nascimento = new GregorianCalendar();  
8  
9     private double altura;  
10  
11    private Selecao selecao;  
12  
13  
14    // GETTERS AND SETTERS  
15 }
```

13.3 Persistência - Mapaemento

Depois de definir algumas entidades podemos começar o processo de implementação da persistência da nossa aplicação. Vamos aplicar os recursos do JPA que aprendemos nos primeiros capítulos. Inicialmente, vamos definir o mapeamento das nossas entidades através das anotações adequadas.

13.4 Exercícios

3. Adicione as anotações do JPA nas classes de modelo.

```
1 @Entity  
2 public class Selecao {  
3  
4     @Id @GeneratedValue  
5     private Long id;  
6  
7     private String pais;  
8  
9     private String tecnico;  
10  
11    // GETTERS AND SETTERS  
12 }
```

```
1 @Entity
2 public class Jogador {
3
4     @Id @GeneratedValue
5     private Long id;
6
7     private String nome;
8
9     private String posicao;
10
11     private Calendar nascimento = new GregorianCalendar();
12
13     private double altura;
14
15     @ManyToOne
16     private Selecao selecao;
17
18
19     // GETTERS AND SETTERS
20 }
```

A anotação **@ManyToOne** é utilizada para indicar a cardinalidade do relacionamento entre jogadores e seleções.

13.5 Persistência - Configuração

Assim como nos capítulos anteriores, implantaremos a nossa aplicação no Glassfish que é um servidor de aplicação Java EE. Os servidores de aplicação Java EE já possuem uma implementação de JPA. Dessa forma, as aplicações não precisam se preocupar em obter um provedor de JPA.

Por outro lado, geralmente, os servidores de aplicação não são distribuídos com os principais drivers JDBC que normalmente as aplicação desejam utilizar. Portanto, devemos adicionar o driver JDBC que desejamos utilizar nas bibliotecas do Glassfish.

Depois disso, devemos configurar as propriedades do JPA através do arquivo **persistence.xml**.

13.6 Exercícios

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 **glassfishv3/glassfish/lib** também da sua Área de Trabalho.
5. Adicione uma pasta chamada **META-INF** na pasta **src** do projeto **K19-CopaDoMundo**.
6. Configure o JPA adicionando o arquivo **persistence.xml** na pasta **META-INF**.

```

1 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↵
4     ns/persistence/persistence_1_0.xsd"
5     version="1.0">
6
7   <persistence-unit name="copadomundo" transaction-type="RESOURCE_LOCAL">
8     <provider>org.hibernate.ejb.HibernatePersistence</provider>
9     <properties>
10       <property name="hibernate.dialect" value="org.hibernate.dialect.↵
11         MySQLInnoDBDialect" />
12       <property name="hibernate.hbm2ddl.auto" value="update" />
13       <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver↵
14         " />
15       <property name="javax.persistence.jdbc.user" value="root" />
16       <property name="javax.persistence.jdbc.password" value="root" />
17       <property name="javax.persistence.jdbc.url" value="jdbc:mysql://↵
18         localhost:3306/copadomundo" />
19     </properties>
20   </persistence-unit>
21 </persistence>

```

7. Abra um terminal; entre no cliente do MySQL Server; apague se existir a base de dados **copadomundo**; e crie uma base de dados nova chamada **copadomundo**.

Para entrar no MySQL Server:

```
mysql -u root -p
```

Para apagar uma base de dados:

```
DROP DATABASE copadomundo;
```

Para criar uma base de dados:

```
CREATE DATABASE copadomundo;
```

13.7 Persistência - Open Session in View

13.7.1 Gerenciando as fábricas de Entity Managers

Quando trabalhamos com JPA, devemos nos preocupar com a criação e destruição das fábricas de entity manager. Ao criar uma fábrica de Entity Manager, todas as configurações e anotações são processadas e armazenadas na memória. De fato, só é necessário realizar esse processo uma vez para cada execução. Além disso, como esse procedimento pode consumir significativamente os recursos da máquina então realizá-lo duas ou mais vezes na mesma execução seria apenas desperdício.

No contexto de uma aplicação web implementada com JSF, podemos criar uma fábrica de entity manager exatamente antes da primeira requisição à servlet do JSF e destruí-la exatamente antes do encerramento da aplicação. Dessa forma, conseguiremos garantir a existência da fábrica durante todo o período no qual a aplicação receberá requisições.

13.7.2 Filtros

Para implementar essa abordagem, podemos criar um filtro no Web Container associado à servlet do JSF. Dessa forma, antes da primeira requisição a essa servlet ele será iniciado e imediatamente antes da aplicação encerrar ele será desativado.

```
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("copadomundo");
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 }
23 }
```

Um filtro é registrado no Web Container através da anotação **@WebFilter**. Através dessa anotação definimos qual servlet está associada ao nosso filtro pelo nome da servlet.

O método **INIT()** é chamado para inicializar o filtro imediatamente antes da primeira requisição ser enviada para a servlet do JSF. Criamos a fábrica de entity manager nesse método.

O método **DESTROY()** é chamado para desativar o filtro imediatamente antes do encerramento da aplicação. Fechamos a fábrica de entity manager nesse método.

13.7.3 Gerenciando os Entity Managers

Provavelmente, a estratégia de gerenciamento de Entity Managers mais simples de entender e manter é adotar o padrão a Open Session in View. A ideia é associar o tempo de vida de um Entity Manager à duração do processamento de uma requisição. Ou seja, quando uma requisição é realizada, criamos um Entity Manager para ser utilizado no tratamento dessa requisição, quando esse processamento terminar, fechamos o Entity Manager.

Para implementar o padrão Open Session in View, podemos utilizar o mesmo filtro que gerencia a criação e o fechamento das fábricas de Entity Manager pois as requisições passam por esse filtro antes de chegar até a servlet do JSF. Além disso, antes da resposta ser enviada ao usuário, a execução passa pelo filtro novamente.

Na chegada de uma requisição, devemos criar um Entity Manager e adicioná-lo na requisição para que ele possa ser acessado pela aplicação durante o processamento dessa requisição. Imediatamente, antes da resposta ser enviada, devemos abrir e confirmar uma transação no Entity Manager para que as alterações decorrentes ao processamento da requisição sejam refletidas no banco de dados.

```
1 @WebFilter(servletNames={"Faces Servlet"})
2 public class JPAFilter implements Filter {
3
4     private EntityManagerFactory factory;
5
6     @Override
7     public void doFilter(ServletRequest request, ServletResponse response,
8         FilterChain chain) throws IOException, ServletException {
9
10         EntityManager entityManager = this.factory.createEntityManager();
11         request.setAttribute("entityManager", entityManager);
12
13         chain.doFilter(request, response);
14
15         try {
16             entityManager.getTransaction().begin();
17             entityManager.getTransaction().commit();
18         } catch (Exception e) {
19             entityManager.getTransaction().rollback();
20         } finally {
21             entityManager.close();
22         }
23     }
24
25     @Override
26     public void init(FilterConfig filterConfig) throws ServletException {
27         this.factory = Persistence.createEntityManagerFactory("copadomundo");
28     }
29
30     @Override
31     public void destroy() {
32         this.factory.close();
33     }
34 }
35
36 }
```

13.8 Exercícios

8. Faça um pacote chamado **filtros** e adicione nele uma classe chamada **JPAFilter**.

```
1 @WebFilter(servletNames={"Faces Servlet"})
2 public class JPAFilter implements Filter {
3
4     private EntityManagerFactory factory;
5
6     @Override
7     public void doFilter(ServletRequest request, ServletResponse response,
8         FilterChain chain) throws IOException, ServletException {
9
10        EntityManager entityManager = this.factory.createEntityManager();
11        request.setAttribute("entityManager", entityManager);
12
13        chain.doFilter(request, response);
14
15        try {
16            entityManager.getTransaction().begin();
17            entityManager.getTransaction().commit();
18        } catch (Exception e) {
19            entityManager.getTransaction().rollback();
20        } finally {
21            entityManager.close();
22        }
23    }
24
25    @Override
26    public void init(FilterConfig filterConfig) throws ServletException {
27        this.factory = Persistence.createEntityManagerFactory("copadomundo");
28    }
29
30    @Override
31    public void destroy() {
32        this.factory.close();
33    }
34
35
36 }
```

13.9 Persistência - Repositórios

Vamos deixar os repositórios para acessar as entidades da nossa aplicação preparados. Os repositórios precisam de Entity Managers para realizar as operações de persistência. Então, cada repositório terá um construtor para receber um Entity Manager como parâmetro.

Como o padrão Open Session in View foi adotado na nossa aplicação, o gerenciamento das transações não é uma tarefa dos repositórios. Mas, o funcionamento deles ainda é afetado pelo controle de transações.

Se um usuário faz uma requisição para cadastrar uma seleção ela só existirá no banco de dados quando o processamento voltar para o filtro que gerencia as transações. Nesse momento, a tela de resposta já teria sido montada pelo Facelets com os dados do banco de dados sem a nova seleção. Em outras palavras, o usuário veria uma listagem sem a seleção que ele acabou de cadastrar.

Para resolver este problema, o repositório de seleções pode enviar a nova seleção imediatamente para o banco de dados bem antes da montagem da tela de resposta. Dessa forma, essa tela de listagem de seleções mostrará inclusive a nova seleção. A implementação desse repositório deve utilizar o método **flush()** que envia imediatamente para o banco de dados as alterações realizadas dentro da transação corrente sem confirmá-las (a confirmação só ocorre na chamada do método **commit()**).

Para que esse processo funcione, devemos alterar o nosso filtro de gerenciamento transacional para que ele abra uma transação na chegada de uma nova requisição. Caso contrário, o método **flush()** não funcionará no repositório de seleções.

Os outros repositórios podem adotar a mesma estratégia para garantir que os usuários vejam dados atualizados.

13.10 Exercícios

9. Faça um pacote chamado **repositorios** e adicione nele uma classe chamada **SelecaoRepository**.

```
1 public class SelecaoRepository {
2     private EntityManager entityManager;
3
4     public SelecaoRepository(EntityManager entityManager) {
5         this.entityManager = entityManager;
6     }
7
8     public void adiciona(Selecao selecao) {
9         this.entityManager.persist(selecao);
10        this.entityManager.flush();
11    }
12
13    public List<Selecao> getSelecoes() {
14        Query query = this.entityManager
15            .createQuery("select s from Selecao as s");
16        return query.getResultList();
17    }
18 }
```

10. Analogamente crie um repositório de jogadores.

```
1 public class JogadorRepository {
2     private EntityManager entityManager;
3
4     public JogadorRepository(EntityManager entityManager) {
5         this.entityManager = entityManager;
6     }
7
8     public void adiciona(Jogador jogador) {
9         this.entityManager.persist(jogador);
10        this.entityManager.flush();
11    }
12
13    public List<Jogador> getJogadores() {
14        Query query = this.entityManager
15            .createQuery("select j from Jogador as j");
16        return query.getResultList();
17    }
18 }
```

11. Altere o filtro de gerenciamento de transações. Abra uma transação na chegada de uma requisição. Veja como deve ficar o método **doFilter()**.

```
1 EntityManager entityManager = this.factory.createEntityManager();
2 request.setAttribute("entityManager", entityManager);
3 entityManager.getTransaction().begin();
4
5 chain.doFilter(request, response);
6
7 try {
8     entityManager.getTransaction().commit();
9 } catch (Exception e) {
10     entityManager.getTransaction().rollback();
11 } finally {
12     entityManager.close();
13 }
```

13.11 Apresentação - Template

Vamos definir um template para as telas da nossa aplicação utilizando os recursos do Facelets. Aplicaremos algumas regras CSS para melhorar a parte visual das telas.

13.12 Exercícios

12. Na pasta **WebContent**, crie um arquivo com algumas regras CSS chamado **style.css**.


```
1 .logo{
2     vertical-align: middle;
3 }
4
5 .botao {
6     background-color: #064D83;
7     margin: 0 0 0 20px;
8     color: white;
9     text-decoration: none;
10    font-size: 20px;
11    line-height: 20px;
12    padding: 5px;
13    vertical-align: middle;
14 }
15
16 .botao:hover{
17     background-color: #cccccc;
18     color: #666666;
19 }
20
21 .formulario fieldset{
22     float: left;
23     margin: 0 0 20px 0;
24     border: 1px solid #333333;
25 }
26
27 .formulario fieldset legend{
28     color: #064D83;
29     font-weight: bold;
30 }
31
32 .botao-formulario{
33     background-color: #064D83;
34     color: #ffffff;
35     padding: 5px;
36     vertical-align: middle;
37     border: none;
38 }
39
40 .mensagem-erro{
41     color: #ff0000;
42 }
43
44 .titulo {
45     color: #064D83;
46     clear: both;
47 }
48
49 .tabela{
50     border: 1px solid #064D83;
51     border-collapse: collapse;
52 }
53
54 .tabela tr th{
55     background-color: #064D83;
56     color: #ffffff;
57 }
58
59 .tabela tr th,
60 .tabela tr td{
61     border: 1px solid #064D83;
62     padding: 2px 5px;
63 }
```

13. Copie o arquivo **k19-logo.png** da pasta **K19-Arquivos** da sua Área de Trabalho para a pasta **WebContent**.

14. Agora implemente um template utilizando Facelets. Crie um arquivo chamado **template.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:ui="http://java.sun.com/jsf/facelets"
6       xmlns:h="http://java.sun.com/jsf/html">
7
8 <h:head>
9   <title>Copa do Mundo</title>
10  <link rel="stylesheet" href="style.css" type="text/css"/>
11 </h:head>
12
13 <h:body>
14   <div id="header">
15     
16     <h:outputLink styleClass="botao" value="selecoes.xhtml">Selecoes</h:outputLink>
17     <h:outputLink styleClass="botao" value="jogadores.xhtml">Jogadores</h:outputLink>
18   <hr/>
19   </div>
20
21   <ui:insert name="conteudo"> Espaço para o conteúdo da tela </ui:insert>
22
23   <div id="footer" style="text-align: center">
24     <hr />
25     &copy; 2010 K19. Todos os direitos reservados.
26   </div>
27 </h:body>
28 </html>
```

15. Adicione duas telas, uma para seleções e outra para jogadores. Essas duas telas devem utilizar o template criado no exercício anterior. Os arquivos devem se chamar: **selecoes.xhtml** e **jogadores.xhtml** e ambos devem possuir o seguinte conteúdo.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2       xmlns:ui="http://java.sun.com/jsf/facelets"
3       xmlns:h="http://java.sun.com/jsf/html"
4       xmlns:f="http://java.sun.com/jsf/core">
5
6 <ui:composition template="template.xhtml">
7
8 </ui:composition>
9 </html>
```

16. Acesse as duas telas criadas anteriormente.

13.13 Cadastrando e Listando Seleções

Na tela de seleções, vamos adicionar um formulário para cadastrar novas seleções e uma tabela para apresentar as já cadastradas. Aplicaremos regras de validação específicas para garantir que nenhum dado incorreto seja armazenado no banco de dados. Além disso, utilizaremos os recursos nativos do JSF 2 para aplicar as técnicas de AJAX.

13.14 Exercícios

17. Para garantir uma melhor legibilidade da nossa aplicação, criaremos uma tela parcial com os campos e o botão referentes ao cadastramento de seleções. Adicione o arquivo **cadastro-selecao.xhtml** na pasta **WebContent** com o seguinte conteúdo.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:f="http://java.sun.com/jsf/core">
5
6 <fieldset>
7     <legend>Nova Seleção</legend>
8     <h:panelGrid columns="3">
9         <h:outputLabel value="País: " for="selecao-pais" />
10        <h:inputText
11            id="selecao-pais"
12            required="true"
13            value="#{selecaoBean.selecao.pais}">
14            <f:validateLength minimum="3" />
15        </h:inputText>
16        <h:message for="selecao-pais" styleClass="mensagem-erro" />
17
18        <h:outputLabel value="Técnico: " for="selecao-tecnico" />
19        <h:inputText
20            id="selecao-tecnico"
21            required="true"
22            value="#{selecaoBean.selecao.tecnico}">
23            <f:validateLength minimum="6" />
24        </h:inputText>
25        <h:message for="selecao-tecnico" styleClass="mensagem-erro" />
26
27        <h:commandButton value="Cadastrar" styleClass="botao-formulario">
28            <f:ajax
29                event="click"
30                execute="@form"
31                listener="#{selecaoBean.adiciona}"
32                render="@form" />
33        </h:commandButton>
34    </h:panelGrid>
35 </fieldset>
36 </html>
```

18. Analogamente ao exercício anterior, vamos criar uma tela parcial para apresentar as seleções dentro de uma tabela. Adicione o arquivo **listagem-selecoes.xhtml** na pasta **WebContent** com o seguinte conteúdo.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core">
5
6 <h3 class="titulo">Listagem de Seleções</h3>
7
8 <h:dataTable
9   id="tabela"
10  value="#{selecaoBean.selecoes}"
11  var="selecao"
12  styleClass="tabela">
13   <h:column>
14     <f:facet name="header">
15       <h:outputText value="Id" />
16     </f:facet>
17     #{selecao.id}
18   </h:column>
19   <h:column>
20     <f:facet name="header">
21       <h:outputText value="País" />
22     </f:facet>
23     #{selecao.pais}
24   </h:column>
25   <h:column>
26     <f:facet name="header">
27       <h:outputText value="Técnico" />
28     </f:facet>
29     #{selecao.tecnico}
30   </h:column>
31 </h:dataTable>
32 </html>
```

19. O próximo passo é montar a tela principal de seleções agrupando as telas parciais criadas anteriormente. Veja como deve ficar o arquivo **selecoes.xhtml**.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core">
5
6 <ui:composition template="/template.xhtml">
7   <ui:define name="conteudo">
8     <h:form styleClass="formulario">
9       <ui:include src="/cadastro-selecao.xhtml"/>
10      <ui:include src="/listagem-selecoes.xhtml"/>
11    </h:form>
12  </ui:define>
13 </ui:composition>
14 </html>
```

20. Implemente um Managed Bean para dar suporte às funcionalidades da tela de seleções. Adicione um pacote chamado **managedbeans**.

```

1 @ManagedBean
2 public class SelecaoBean {
3
4     @ManagedProperty(value = "#{entityManager}")
5     private EntityManager entityManager;
6
7     private Selecao selecao = new Selecao();
8
9     public void adiciona() {
10         SelecaoRepository repository = new SelecaoRepository(this.entityManager);
11         repository.adiciona(this.selecao);
12         this.selecao = new Selecao();
13     }
14
15     public List<Selecao> getSelecoes() {
16         SelecaoRepository repository = new SelecaoRepository(this.entityManager);
17         return repository.getSelecoes();
18     }
19
20     // GETTERS AND SETTERS
21 }

```

13.15 Mensagens de Erro

Vamos personalizar as mensagens de erro criando um arquivo de mensagens em português. Devemos registrar esse arquivo no **faces-config.xml**.

13.16 Exercícios

21. Crie um pacote na pasta **src** chamado **resources**. Adicione nesse pacote um arquivo chamado **Messages.properties** com o seguinte conteúdo:

```

1 javax.faces.component.UIInput.REQUIRED = Campo obrigatório
2 javax.faces.validator.LengthValidator.MINIMUM = O número mínimo de caracteres é "{0}".

```

22. Adicione o seguinte trecho no arquivo **faces-config.xml**:

```

1 <application>
2   <message-bundle>resources.Messages</message-bundle>
3 </application>

```

13.17 Removendo Seleções

Vamos acrescentar a funcionalidade de remover seleções utilizando o suporte de AJAX do JSF 2.

13.18 Exercícios

23. Acrescente uma coluna na tabela que apresenta das seleções alterando o arquivo **listagem-selecoes.xhtml**.

```
1 <h:column>
2   <f:facet name="header">
3     <h:outputText value="Remover" />
4   </f:facet>
5   <f:ajax
6     event="click"
7     render="@form"
8     listener="#{selecaoBean.remove(selecao)}">
9     <h:commandLink>Remover</h:commandLink>
10  </f:ajax>
11 </h:column>
```

24. Implemente um método para remover seleções no repositório de seleções, **SelecaoRepository**.

```
1 public void remove(Selecao selecao) {
2     this.entityManager.remove(selecao);
3     this.entityManager.flush();
4 }
```

25. Adicione um método para remover seleções no Managed Bean **SelecaoBean**.

```
1 public void remove(Selecao selecao) {
2     SelecaoRepository repository = new SelecaoRepository(this.entityManager);
3     repository.remove(selecao);
4 }
```

13.19 Otimizando o número de consultas

Os **getters** dos Managed Beans são chamados diversas vezes pelo JSF durante o processamento de uma requisição. Por exemplo, o método **getSelecoes()** da classe **SelecaoBean** é chamado cerca de nove vezes durante o processamento da requisição à página principal de seleções. Esse método chama o **getSelecoes()** do repositório que por sua vez faz uma consulta no banco de dados, ou seja, são realizadas cerca de nove consultas iguais para gerar a tela principal de seleções.

Podemos diminuir o número de consultas fazendo os Managed Beans armazenarem o resultado de uma consulta para utilizá-lo o maior número possível de vezes. Apenas devemos tomar cuidado para não manter informações desatualizadas nos Managed Beans. Por exemplo, quando uma seleção for adicionada ou removida devemos descartar qualquer resultado armazenado no Managed Bean **SelecaoBean**.

13.20 Exercícios

26. Imprima uma mensagem no método `getSelecoes()` do Managed Bean **SelecaoBean** para verificar a quantidade de vezes que ele é chamado quando o usuário acessa a url:

`http://localhost:8080/K19-CopaDoMundo/selecoes.xhtml`.

```
1 public List<Selecao> getSelecoes() {  
2     System.out.println("CHAMANDO O REPOSITORIO");  
3     SelecaoRepository repository = new SelecaoRepository(this.entityManager);  
4     return repository.getSelecoes();  
5 }
```

27. Verifique quantas vezes a mensagem é impressa no console do eclipse acessando a url:

`http://localhost:8080/K19-CopaDoMundo/selecoes.xhtml`

28. Altere a classe **SelecaoBean** para esse Managed Bean guardar os resultados das consultas feitas nos repositórios.

```
1 @ManagedBean  
2 public class SelecaoBean {  
3  
4     @ManagedProperty(value = "#{entityManager}")  
5     private EntityManager entityManager;  
6  
7     private Selecao selecao = new Selecao();  
8  
9     private List<Selecao> selecoes;  
10  
11     public void adiciona() {  
12         SelecaoRepository repository = new SelecaoRepository(this.entityManager);  
13         repository.adiciona(this.selecao);  
14         this.selecao = new Selecao();  
15         this.selecoes = null;  
16     }  
17  
18     public void remove(Selecao selecao) {  
19         SelecaoRepository repository = new SelecaoRepository(this.entityManager);  
20         repository.remove(selecao);  
21         this.selecoes = null;  
22     }  
23  
24     public List<Selecao> getSelecoes() {  
25         if (this.selecoes == null) {  
26             System.out.println("CHAMANDO O REPOSITORIO");  
27             SelecaoRepository repository = new SelecaoRepository(  
28                 this.entityManager);  
29             this.selecoes = repository.getSelecoes();  
30         }  
31         return this.selecoes;  
32     }  
33  
34     // GETTERS AND SETTERS  
35 }
```

29. Verifique quantas vezes a mensagem é impressa acessando novamente a url:

`http://localhost:8080/K19-CopaDoMundo/selecoes.xhtml`

13.21 Cadastrando, Listando e Removendo Jogadores

Na tela de jogadores, vamos adicionar um formulário para cadastrar novos jogadores e uma tabela para apresentar os já cadastrados. Aplicaremos regras de validação específicas para garantir que nenhum dado incorreto seja armazenado no banco de dados. Além disso, utilizaremos os recursos nativos do JSF 2 para aplicar as técnicas de AJAX.

13.22 Exercícios

30. Para garantir uma melhor legibilidade da nossa aplicação, criaremos uma tela parcial com os campos e o botão referentes ao cadastramento de jogadores. Adicione o arquivo **cadastro-jogador.xhtml** na pasta **WebContent** com o seguinte conteúdo.


```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core">
5
6 <fieldset>
7   <legend>Novo Jogador</legend>
8   <h:panelGrid columns="3">
9     <h:outputLabel value="Nome: " for="jogador-nome" />
10    <h:inputText
11      id="jogador-nome"
12      required="true"
13      value="#{jogadorBean.jogador.nome}">
14      <f:validateLength minimum="6" />
15    </h:inputText>
16    <h:message for="jogador-nome" styleClass="mensagem-erro" />
17
18    <h:outputLabel value="Posição: " for="jogador-posicao" />
19    <h:inputText
20      id="jogador-posicao"
21      required="true"
22      value="#{jogadorBean.jogador.posicao}">
23    </h:inputText>
24    <h:message for="jogador-posicao" styleClass="mensagem-erro" />
25
26    <h:outputLabel value="Data de Nascimento: " for="jogador-nascimento" />
27    <h:inputText
28      id="jogador-nascimento"
29      required="true"
30      value="#{jogadorBean.jogador.nascimento.time}">
31      <f:convertDateTime pattern="dd/MM/yyyy" />
32    </h:inputText>
33    <h:message for="jogador-nascimento" styleClass="mensagem-erro" />
34
35    <h:outputLabel value="Altura (m): " for="jogador-altura" />
36    <h:inputText
37      id="jogador-altura"
38      required="true"
39      value="#{jogadorBean.jogador.altura}">
40    </h:inputText>
41    <h:message for="jogador-altura" styleClass="mensagem-erro" />
42
43    <h:outputLabel value="Seleção: " for="jogador-selecao" />
44    <h:selectOneMenu id="jogador-selecao" value="#{jogadorBean.selecaoID}">
45      <f:selectItems
46        value="#{selecaoBean.selecoes}"
47        var="selecao"
48        itemLabel="#{selecao.pais}"
49        itemValue="#{selecao.id}" />
50    </h:selectOneMenu>
51    <h:message for="jogador-selecao" styleClass="mensagem-erro" />
52
53    <h:commandButton value="Cadastrar" styleClass="botao-formulario">
54      <f:ajax
55        event="click"
56        execute="@form"
57        listener="#{jogadorBean.adiciona}"
58        render="@form" />
59    </h:commandButton>
60  </h:panelGrid>
61 </fieldset>
62 </html>

```

31. Analogamente ao exercício anterior, vamos criar uma tela parcial para apresentar os jogadores dentro de uma tabela. Adicione o arquivo **listagem-jogadores.xhtml** na pasta **WebContent** com o seguinte conteúdo.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core">
5
6 <h3 class="titulo">Listagem de Jogadores</h3>
7
8 <h:dataTable
9   id="tabela"
10  value="#{jogadorBean.jogadores}"
11  var="jogador"
12  styleClass="tabela">
13   <h:column>
14     <f:facet name="header">
15       <h:outputText value="Id" />
16     </f:facet>
17     #{jogador.id}
18   </h:column>
19   <h:column>
20     <f:facet name="header">
21       <h:outputText value="Nome" />
22     </f:facet>
23     #{jogador.nome}
24   </h:column>
25
26   <h:column>
27     <f:facet name="header">
28       <h:outputText value="Posição" />
29     </f:facet>
30     #{jogador.posicao}
31   </h:column>
32
33   <h:column>
34     <f:facet name="header">
35       <h:outputText value="Nascimento" />
36     </f:facet>
37
38     <h:outputText value="#{jogador.posicao}">
39       <f:convertDateTime pattern="dd/MM/yyyy"/>
40     </h:outputText>
41   </h:column>
42
43   <h:column>
44     <f:facet name="header">
45       <h:outputText value="Altura" />
46     </f:facet>
47     #{jogador.altura}
48   </h:column>
49
50   <h:column>
51     <f:facet name="header">
52       <h:outputText value="Seleção" />
53     </f:facet>
54     #{jogador.selecao.pais}
55   </h:column>
56
57   <h:column>
58     <f:facet name="header">
59       <h:outputText value="Remover" />
60     </f:facet>
61     <f:ajax
62       event="click"
63       render="@form"
64       listener="#{jogadorBean.remove(jogador)}">
65       <h:commandLink>Remover</h:commandLink>
66     </f:ajax>
67   </h:column>
68 </h:dataTable>
69 </html>
```

32. O próximo passo é montar a tela principal de jogadores agrupando as telas parciais criadas anteriormente. Veja como deve ficar o arquivo **jogadores.xhtml**.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:f="http://java.sun.com/jsf/core">
5
6 <ui:composition template="/template.xhtml">
7     <ui:define name="conteudo">
8         <h:form styleClass="formulario">
9             <ui:include src="/cadastro-jogador.xhtml"/>
10            <ui:include src="/listagem-jogadores.xhtml"/>
11        </h:form>
12    </ui:define>
13 </ui:composition>
14 </html>
```

33. Implemente um método para remover jogadores no repositório de jogadores, **Jogador-Repository**.

```
1 public void remove(Jogador jogador) {
2     this.entityManager.remove(jogador);
3     this.entityManager.flush();
4 }
```

34. Adicione um método no repositório de seleções para buscar por id.

```
1 public Selecao procura(Long id) {
2     return this.entityManager.find(Selecao.class, id);
3 }
```

35. Crie um Managed Bean para trabalhar com a tela de jogadores.

```
1 @ManagedBean
2 public class JogadorBean {
3
4     @ManagedProperty(value = "#{entityManager}")
5     private EntityManager entityManager;
6
7     private Jogador jogador = new Jogador();
8
9     private Long selecaoID;
10
11     private List<Jogador> jogadores;
12
13     public void adiciona() {
14         SelecaoRepository selecaoRepository =
15             new SelecaoRepository(this.entityManager);
16         Selecao selecao = selecaoRepository.procura(this.selecaoID);
17         this.jogador.setSelecao(selecao);
18
19         JogadorRepository jogadorRepository =
20             new JogadorRepository(this.entityManager);
21         jogadorRepository.adiciona(this.jogador);
22
23         this.jogador = new Jogador();
24         this.jogadores = null;
25     }
26
27     public void remove(Jogador jogador) {
28         JogadorRepository repository = new JogadorRepository(this.entityManager);
29         repository.remove(jogador);
30         this.jogadores = null;
31     }
32
33     public List<Jogador> getJogadores() {
34         if (this.jogadores == null) {
35             System.out.println("CHAMANDO O REPOSITORIO");
36             JogadorRepository repository = new JogadorRepository(
37                 this.entityManager);
38             this.jogadores = repository.getJogadores();
39         }
40         return this.jogadores;
41     }
42
43     // GETTERS AND SETTERS
44 }
```

13.23 Removendo Seleções com Jogadores

Se uma seleção possui jogadores ela não poderá ser removida pois teríamos dados inconsistentes no banco de dados. Em outras palavras, teríamos jogadores vinculados com seleções que já teriam sido removidas. Nesse caso, uma possibilidade é informar ao usuário que ele só pode remover seleções sem jogadores. Outra possibilidade é remover a seleção e os jogadores quando o usuário clicar no link para remover uma seleção. Na verdade, a maneira de proceder depende das regras da aplicação. Vamos supor que a regra da nossa aplicação é remover tanto a seleção quanto os jogadores. Devemos alterar o repositório de seleções para aplicar essa regra.

13.24 Exercícios

36. Tente remover pela interface web uma seleção que possua jogadores.

37. Altere a classe **SelecaoRepository** para que na remoção de seleções os jogadores correspondentes também sejam removidos. A modificação deve ser feita no método **remove()**.

```
1 public void remove(Selecao selecao) {  
2     this.entityManager.remove(selecao);  
3  
4     Query query = this.entityManager  
5         .createQuery("select j from Jogador as j where j.selecao = :selecao");  
6     query.setParameter("selecao", selecao);  
7  
8     List<Jogador> jogadores = query.getResultList();  
9     for (Jogador jogador : jogadores) {  
10         this.entityManager.remove(jogador);  
11     }  
12  
13     this.entityManager.flush();  
14 }
```

13.25 Controle de Acesso

Na maioria dos casos, as aplicações devem controlar o acesso dos usuários. Vamos implementar um mecanismo de autenticação na nossa aplicação utilizando filtro. As requisições feitas pelos usuários passarão por esse filtro antes de chegar ao controle do JSF. A função do filtro é verificar se o usuário está logado ou não. Se estiver logado o filtro autoriza o acesso. Caso contrário, o filtro redirecionará o usuário para a tela de login.

13.26 Exercícios

38. Adicione no pacote filtros a seguinte classe:

```
1 @WebFilter(servletNames = { "Faces Servlet" })
2 public class LoginFilter 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("autenticado") != 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
29 }
```

39. Crie a tela de login adicionando um arquivo chamado **login.xhtml** na pasta **WebContent**.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:f="http://java.sun.com/jsf/core">
5
6 <ui:composition template="/template.xhtml">
7     <ui:define name="conteudo">
8         <h:form styleClass="formulario">
9             <h:outputLabel value="Usuário: " for="usuario"/>
10            <h:inputText id="usuario" value="#{loginBean.usuario}" />
11            <h:outputLabel value="Senha: " for="senha"/>
12            <h:inputSecret id="senha" value="#{loginBean.senha}" />
13            <h:commandButton value="Entrar" action="#{loginBean.entrar}" />
14        </h:form>
15    </ui:define>
16 </ui:composition>
17 </html>
```

40. Implemente o Managed Bean que autentica os usuários. Adicione uma classe no pacote **managedbeans** chamada **LoginBean**.

```

1  @ManagedBean
2  public class LoginBean {
3
4      private String usuario;
5      private String senha;
6
7      public String entrar() {
8          if ("k19".equals(this.usuario) && "k19".equals(this.senha)) {
9              HttpSession session = (HttpSession) FacesContext
10                  .getCurrentInstance().getExternalContext()
11                  .getSession(false);
12              session.setAttribute("autenticado", true);
13              return "/selecoes";
14          } else {
15              return "/login";
16          }
17      }
18
19      public String sair() {
20          HttpSession session = (HttpSession) FacesContext.getCurrentInstance()
21              .getExternalContext().getSession(false);
22          session.removeAttribute("autenticado");
23          return "/ate-logo";
24      }
25
26      // GETTERS AND SETTERS
27  }

```

41. Crie uma tela de despedida que será utilizada quando um usuário sair da aplicação. Adicione um arquivo chamado **ate-logo.xhtml** na pasta **WebContent**.

```

1  <html xmlns="http://www.w3.org/1999/xhtml"
2      xmlns:ui="http://java.sun.com/jsf/facelets"
3      xmlns:h="http://java.sun.com/jsf/html"
4      xmlns:f="http://java.sun.com/jsf/core">
5
6  <ui:composition template="/template.xhtml">
7      <ui:define name="conteudo">
8          <h3 class="titulo">Até Logo</h3>
9      </ui:define>
10 </ui:composition>
11 </html>

```

42. Modifique o template das telas para acrescentar um botão de sair.

```
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
8 <h:head>
9   <title>Copa do Mundo</title>
10  <link rel="stylesheet" href="style.css" type="text/css"/>
11 </h:head>
12
13 <h:body>
14   <div id="header">
15     
16     <h:outputLink styleClass="botao" value="selecoes.xhtml">Selecoes</h:outputLink>
17     <h:outputLink styleClass="botao" value="jogadores.xhtml">Jogadores</h:outputLink>
18     <h:form style="display: inline;">
19       <h:commandLink styleClass="botao" action="#{loginBean.sair}">Sair</h:commandLink>
20     </h:form>
21     <hr />
22   </div>
23
24   <ui:insert name="conteudo"> Espaço para o conteúdo da tela </ui:insert>
25
26   <div id="footer" style="text-align: center">
27     <hr />
28     &copy; 2010 K19. Todos os direitos reservados.
29   </div>
30 </h:body>
31 </html>
```

13.27 Ordem dos filtros

Em alguns casos a ordem de execução dos filtros afeta o funcionamento da aplicação. No caso da nossa aplicação, a ordem correta de execução dos filtros é primeiro o filtro de autenticação e depois o de controle transacional.

Não podemos definir a ordem de execução de um conjunto de filtros através de anotações. Podemos determinar essa sequência se os filtros forem registrados no arquivo de configurações do Web Container, **web.xml**. A ordem de execução dos filtros é a ordem na qual eles aparecem no arquivo de configuração.

13.28 Exercícios

43. Remova a anotação **@WebFilter** das classes **JPAFilter** e **LoginFilter** que estão no pacote **filtros**.
44. Adicione o seguinte trecho de configuração no arquivo **web.xml**.


```
1 <filter>
2   <filter-name>LoginFilter</filter-name>
3   <filter-class>jpa.LoginFilter</filter-class>
4 </filter>
5 <filter-mapping>
6   <filter-name>LoginFilter</filter-name>
7   <servlet-name>Faces Servlet</servlet-name>
8 </filter-mapping>
9 <filter>
10  <filter-name>JPAFilter</filter-name>
11  <filter-class>jpa.JPAFilter</filter-class>
12 </filter>
13 <filter-mapping>
14   <filter-name>JPAFilter</filter-name>
15   <servlet-name>Faces Servlet</servlet-name>
16 </filter-mapping>
```

13.29 Controle de Erro

Quando uma exception ocorre durante o processamento de uma requisição, o filtro de controle transacional dispara um rollback para desfazer o que já havia sido alterado no banco de dados. Porém, o usuário não é informado sobre o problema.

Podemos configurar uma página de erro padrão para ser utilizada toda vez que um erro ocorrer. O filtro de controle transacional deve lançar uma **ServletException** após disparar o rollback para informar o Web Container que houve uma falha no processamento da requisição. Depois, devemos configurar uma página de erro padrão no Web Container.

13.30 Exercícios

45. Modifique o filtro de controle transacional para que ele informe o Web Container através de uma exception que houve um problema no processamento de uma requisição.

```
1 public class JPAFilter implements Filter {
2
3     private EntityManagerFactory factory;
4
5     @Override
6     public void doFilter(ServletRequest request, ServletResponse response,
7         FilterChain chain) throws IOException, ServletException {
8
9         EntityManager entityManager = this.factory.createEntityManager();
10        request.setAttribute("entityManager", entityManager);
11        entityManager.getTransaction().begin();
12
13        chain.doFilter(request, response);
14
15        try {
16            entityManager.getTransaction().commit();
17        } catch (Exception e) {
18            entityManager.getTransaction().rollback();
19            throw new ServletException(e);
20        } finally {
21            entityManager.close();
22        }
23    }
24
25    @Override
26    public void init(FilterConfig filterConfig) throws ServletException {
27        this.factory = Persistence.createEntityManagerFactory("copadomundo");
28    }
29
30    @Override
31    public void destroy() {
32        this.factory.close();
33    }
34 }
35 }
```

46. Configure a página de erro no arquivo **web.xml**. Acrescente o seguinte trecho.

```
1 <error-page>
2     <exception-type>java.lang.Exception</exception-type>
3     <location>/erro.xhtml</location>
4 </error-page>
```

47. Crie a página de erro adicionando um arquivo chamado **erro.xhtml** na pasta **WebContent** com o seguinte conteúdo.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:f="http://java.sun.com/jsf/core">
5
6     <ui:composition template="/template.xhtml">
7         <ui:define name="conteudo">
8             <h3 class="titulo">Erro Interno</h3>
9         </ui:define>
10    </ui:composition>
11 </html>
```

48. Vamos criar um Managed Bean que sempre gera um problema para testar a página de erro. Adicione a classe **ErroBean** no pacote **managedbeans** com o seguinte conteúdo.

```

1 @ManagedBean
2 public class ErroBean {
3     public void erro(){
4         throw new NullPointerException();
5     }
6 }

```

49. Modifique o template para adicionar um botão que dispara o Managed Bean que sempre gera um erro.

```

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
8 <h:head>
9     <title>Copa do Mundo</title>
10    <link rel="stylesheet" href="style.css" type="text/css"/>
11 </h:head>
12
13 <h:body>
14     <div id="header">
15         
16         <h:outputLink styleClass="botao" value="selecoes.xhtml">Selecoes</h:outputLink>
17         <h:outputLink styleClass="botao" value="jogadores.xhtml">Jogadores</h:outputLink>
18         <h:form style="display: inline;">
19             <h:commandLink styleClass="botao" action="#{loginBean.sair}">Sair</h:commandLink>
20             <h:commandLink styleClass="botao" action="#{erroBean.erro}">Erro</h:commandLink>
21         </h:form>
22         <hr/>
23     </div>
24
25     <ui:insert name="conteudo"> Espaço para o conteúdo da tela </ui:insert>
26
27     <div id="footer" style="text-align: center">
28         <hr />
29         &copy; 2010 K19. Todos os direitos reservados.
30     </div>
31 </h:body>
32 </html>

```

13.31 Enviando email

Quando um erro ocorre na nossa aplicação, podemos permitir que o usuário envie uma email para os administradores do sistema. Devemos utilizar a especificação **JavaMail** para que a nossa aplicação JSF tenha a capacidade de enviar mensagens por email.

13.32 Exercícios

50. Altere a tela de erro adicionando um formulário para o usuário escrever uma mensagem para os administradores da aplicação.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core">
5
6 <ui:composition template="/template.xhtml">
7   <ui:define name="conteudo">
8     <h3 class="titulo">Erro Interno</h3>
9     <p>Envie uma mensagem para os administradores do sistema.</p>
10    <h:form styleClass="formulario">
11
12      <h:panelGrid columns="3">
13        <h:outputLabel value="Mensagem: " for="campo-mensagem" />
14        <h:inputTextarea
15          id="campo-mensagem"
16          value="#{emailBean.mensagem}"
17          required="true"
18          rows="10"
19          cols="30"/>
20        <h:message for="campo-mensagem" />
21
22        <h:commandButton value="Enviar" action="#{emailBean.envia}" />
23      </h:panelGrid>
24    </h:form>
25  </ui:define>
26 </ui:composition>
27 </html>
```

51. Crie um Managed Bean que envie as mensagens por email utilizando a especificação JavaMail. Observação, utilize usuários, senhas e emails válidos do gmail para este exercício.

```
1 @ManagedBean
2 public class EmailBean {
3
4     private String mensagem;
5
6     private Properties properties = new Properties();
7
8     private Authenticator authenticator;
9
10    public EmailBean() {
11        this.properties.put("mail.smtp.auth", true);
12        this.properties.put("mail.smtp.port", "465");
13        this.properties.put("mail.host", "smtp.gmail.com");
14
15        this.properties.put("mail.smtp.socketFactory.class",
16            "javax.net.ssl.SSLSocketFactory");
17        this.properties.put("mail.smtp.socketFactory.fallback", "false");
18
19        this.authenticator = new Authenticator() {
20            protected PasswordAuthentication getPasswordAuthentication() {
21                return new PasswordAuthentication("USUÁRIO DO GMAIL", "SENHA DO USUÁRIO↵
22                ");
23            };
24        };
25    }
26
27    public String envia() throws AddressException, MessagingException {
28        Session session = Session.getInstance(this.properties,
29            this.authenticator);
30        MimeMessage message = new MimeMessage(session);
31
32        message.setFrom(new InternetAddress("EMAIL DE ORIGEM"));
33        message.setRecipients(Message.RecipientType.TO, "EMAIL DE DESTINO");
34
35        message.setSentDate(new Date());
36
37        message.setSubject("Copa do Mundo - Erro");
38        message.setContent(this.mensagem, "text/plain");
39
40        Transport.send(message);
41
42        return "/selecoes";
43    }
44
45    // GETTERS AND SETTERS
46 }
```